Intel® oneAPI DPC++/C++ Compiler Developer Guide and Reference

Disclaimer and Legal Information

Contents

Part VI: Compatibility and Portability

Notices and Disclaimers

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Intel® oneAPI DPC++/C++ Compiler Developer Guide and Reference

This document is for version 2022.2 of the compilers.

This guide provides information about the Intel® oneAPI DPC++/C++ Compiler (`icx, icpx, dpcpp, dpcpp-cl`) and runtime environment.

The Intel® oneAPI DPC++/C++ Compiler is available as part of the Intel® oneAPI Base Toolkit, Intel® oneAPI HPC Toolkit, Intel® oneAPI IoT Toolkit, or as a standalone compiler.

Refer to the Intel® oneAPI DPC++/C++ Compiler product page and the Release Notes for more information about features, specifications, and downloads.

Use this guide to learn about:

- **Compiler Setup**: How to invoke the compiler on the command line or from within an IDE.
- **Compiler Options**: Information about options you can use to affect optimization, code generation, and more.
- **Pragmas**: Information about directives to provide the compiler with instructions for specific tasks, including splitting large loops into smaller ones, enabling or disabling optimization for code, or offloading computation to the target.
- **OpenMP* Support**: Compiler support for OpenMP 5.0 Version TR4 features and some OpenMP Version 5.1 features.

For more information, refer to Introducing the Intel® oneAPI DPC++/C++ Compiler.

For information about Intel intrinsics, visit Intel® Intrinsics Guide.

Notices and Important Information

- To use Microsoft Visual C++ (MSVC) compatible options with SYCL, use `dpcpp-cl`.
- In this document, you may see features labeled as experimental. An experimental feature is one that requires further testing and possible refinement. Depending on testing results, such features may be fully defined and implemented or they may be removed in a future release.
- The Intel® oneAPI DPC++/C++ Compiler (`icx, icpx, dpcpp, dpcpp-cl`) does not support macOS*. For macOS or Xcode* support use Intel® C++ Compiler Classic. For more information, visit the Intel® C++ Compiler Classic Developer Guide and Reference.

Using the Compiler Documentation

- **Context Sensitive/F1 Help**: To use the Context Sensitive/F1 Help feature, visit the Download Documentation: Intel® Compiler (Current and Previous) page and follow the provided instructions.
- **Previous Versions of the Developer Guide and Reference**: Visit the Download Documentation: Intel® Compiler (Current and Previous) page to download PDF or FAR HTML versions of previous compiler documentation.

> **NOTE** When searching HTML files, use a Google Chrome* or Internet Explorer* browser to view your downloaded copy of the Developer Guide and Reference.
> If you use Mozilla Firefox*, you may encounter an issue where the **Search** tab does not work. As a workaround, you can use the **Contents** and **Index** tabs or a third-party search tool to find your content.

Intel® oneAPI DPC++/C++ Compiler Introduction

Part I

Using the Intel® oneAPI DPC++/C++ Compiler, you can compile and generate applications that can run on Intel® 64 architecture. You can also create programs for the IA-32 architecture on Windows* and Linux*.

NOTE IA-32 applications do not apply for SYCL.

Intel® 64 architecture applications can run on the following:

- Windows operating systems for Intel® 64 architecture-based systems.
- Linux operating systems for Intel® 64 architecture-based systems.

IA-32 architecture applications can run on the following:

- Supported Windows operating systems
- Supported Linux operating systems

Unless specified otherwise, assume the information in this document applies to all supported architectures and all operating systems.

You can use the compiler in the command-line or in a supported Integrated Development Environment (IDE):

- Microsoft Visual Studio* (Windows only)
- Eclipse*/CDT (Linux only)

See the Release Notes for complete information on supported architectures, operating systems, and IDEs for this release.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Feature Requirements

To use these tools and features, you need licensed versions of the tools and you must have an appropriately supported version of the product edition. For more information, check the product release notes.

NOTE Some features may require additional product installation.

The following table shows components (tools) and where to find additional information on them.

Component	More Information
Intel® oneAPI DPC++/C++ Compiler Intel® Advisor	More information on tools and features can be found on the Intel® Developer Zone and the Software Development Tools pages.

Component	More Information
Intel® Inspector	
Intel® Trace Analyzer and Collector	
Intel® VTune™ Profiler	

The following table lists dependent features and their corresponding required products. For certain compiler options, the compilation may fail if the option is specified but the required product is not installed. In this case, remove the option from the command line and recompile.

Feature Requirements

Feature	Requirement
Intel® oneAPI Threading Building Blocks (oneTBB)	The `-tbb`, `-qtbb`, and `/Qtbb` options require a oneTBB install.
Intel® oneAPI Math Kernel Library (oneMKL)	The `-qmkl`, `-mkl`, and `/Qmkl` options require a oneMKL install.
Intel® oneAPI Data Analytics Library (oneDAL)	The `-daal`, `-qdaal`, and `/Qdaal` options require a oneDAL install.
Intel® Integrated Performance Primitives (Intel® IPP)	The `-ipp`, `-qipp`, and `/Qipp` options require an Intel® IPP install.
Intel® Integrated Performance Primitives Cryptography (Intel® IPP Cryptography)	Use `crypto` to link to the Intel® IPP Cryptography library.
Thread Checking	Intel® Inspector
Trace Analyzing and Collecting	Intel® Trace Analyzer and Collector Compiler options related to this feature may require a set-up script. For further information, see the product documentation.

Refer to the Release Notes for detailed information about system requirements, late changes to the products, supported architectures, operating systems, and Integrated Development Environments (IDEs).

Get Help and Support

Intel® Software Documentation

You can find product documentation for many released products at the Explore Our Documentation page.

Product Website and Support

To find product information, register your product, or contact Intel, visit the Get Help page. At the support site, you will find comprehensive product information, including:

- Links to Get Started, Documentation, Individual Support, and Registration.
- Links to information such as white papers, articles, and user forums.
- Links to product information.
- Links to news and events.

Online Service Center

For more information about the Online Service Center visit the Support page.

NOTE To access support, you must register your product at the Intel Registration Center.

Release Notes

For detailed information on system requirements, late changes to the products, supported architectures, operating systems, and Integrated Development Environments (IDE) see the Release Notes for the product.

Forums

You can find helpful information in the Intel Software user forums. You can also submit questions to the forums. To see the list of the available forums, go to the Software Development Tools forum for general information, or visit a specific forum for:

- Intel® C++ Compilers
- Intel® oneAPI Data Parallel C++

Related Information

Recommended Additional Reading

You are strongly encouraged to read the following books for in-depth understanding of threading. Each book discusses general concepts of parallel programming by explaining a particular programming technology:

- For information on Intel® Threading Building Blocks (Intel® TBB): Reinders, James. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly, July 2007
- For information on OpenMP technology: Chapman, Barbara, Gabriele Jost, Ruud van der Pas, and David J. Kuck (foreword). *Using OpenMP: Portable Shared Memory Parallel Programming*. MIT Press, October 2007
- For information on Microsoft Win32 Threading (for Windows users): Akhter, Shameem, and Jason Roberts. *Multi-Core Programming: Increasing Performance through Software Multithreading*, Intel Press, April 2006

Intel does not endorse these books or recommend them over other books on the same subjects.

Additional Product Information

For additional technical product information including white papers, forums, and documentation, visit https://software.intel.com/content/www/us/en/develop/tools.html

Additional Language Information

- For information on SYCL and for programming of heterogeneous systems using C++ and SYCL, visit https://link.springer.com/book/10.1007/978-1-4842-5574-2 for a book download.
- For information about the C++ standards, visit http://www.isocpp.org/.
- For information about the C standards, visit http://www.open-std.org/jtc1/sc22/wg14/.
- For information about the OpenMP standards, visit http://www.openmp.org/

Notational Conventions

Information in this documentation applies to all supported operating systems and architectures unless otherwise specified. This documentation uses the following conventions:

Notational Conventions

THIS TYPE	Indicates language keywords.
this type	Indicates command-line or option arguments.
`This type`	Indicates a code example.
`This type`	Indicates what you type as input.
This type	Indicates menu names, menu items, button names, dialog window names, and other user-interface items.
File > Open	Menu names and menu items joined by a greater than (>) sign to indicate a sequence of actions. For example, **Click File > Open** indicates that in the **File** menu, you would click **Open** to perform this action.
{value | value}	Indicates a choice of items or values. You can usually only choose one of the values in the braces.
[*item*]	Indicates items that are optional.
item [, *item*]...	Indicates that the item preceding the ellipsis (...) can be repeated.
Intel® C++	This term refers to the name of the common compiler language supported by the Intel® oneAPI DPC++/C++ Compiler.
compiler or the compiler	These terms are used when information is not limited to only one specific compiler, or when it is not necessary to indicate a specific compiler.
Windows or Windows operating system	These terms refer to all supported Microsoft Windows operating systems.
Linux or Linux operating system	These terms refer to all supported Linux operating systems.
*	An asterisk at the end of a word or name indicates it is a third-party product trademark.
compiler option	This term refers to Linux or Windows options, which are used by the compiler to compile applications.

Additional Conventions Used for Compiler Options

compiler option name shortcuts	The following conventions are used as shortcuts when referencing compiler option names in descriptions: • No initial – or / This shortcut is used for option names that are the same for Linux and Windows except for the initial character.

	For example, `Fa` denotes: • Linux: `-Fa` • Windows: `/Fa` • [Q]option-name This shortcut is used for option names that only differ because the Windows form starts with a Q. For example, [Q]ipo denotes: • Linux: `-ipo` • Windows: `/Qipo` • [q or Q]option-name This shortcut is used for option names that only differ because the Linux form starts with a q and the Windows form starts with a Q. For example, [q or Q]opt-report denotes: • Linux: `-qopt-report` • Windows: `/Qopt-report` More dissimilar compiler option names are shown in full.
`/option` or `-option`	A slash before an option name indicates the option is available on Windows. A dash before an option name indicates the option is available on Linux systems. For example: • Linux : `-help` • Windows: `/help` **NOTE** If an option is available on all supported operating systems, no slash or dash appears in the general description of the option. The slash and dash will only appear where the option syntax is described.
`/option:argument` or `-option=argument`	Indicates that an option requires an argument (parameter).
`/option:`*`keyword`* or `-option=`*`keyword`*	Indicates that an option requires one of the *`keyword`* values.
`/option`[`:`*`keyword`*] or `-option`[`=`*`keyword`*]	Indicates that the option can be used alone or with an optional keyword.
`option`[*n*] or `option`[:*n*] or `option`[=*n*]	Indicates that the option can be used alone or with an optional value. For example, in `-unroll`[=*n*], the *n* can be omitted or a valid value can be specified for *n*.

`option`[-]	Indicates that a trailing hyphen disables the option. For example, `/Qglobal_hoist-` disables the Windows option `/Qglobal_hoist`.
`[no]option` or `[no-]option`	Indicates that `no` or `no-` preceding an option disables the option. For example: In the Linux option `-[no-]global_hoist`, `-global_hoist` enables the option, while `-no-global_hoist` disables it. In some options, the `no` appears later in the option name. For example, `-fno-common` disables the `-fcommon` option.

Part II

Compiler Setup

You can use the Intel® oneAPI DPC++/C++ Compiler from the command line, Eclipse, or Microsoft Visual Studio.

These IDEs are described in further detail in their corresponding sections.

See Also

Use the Command Line
Use Eclipse
Use Microsoft Visual Studio

Use the Command Line

This section provides information about the Command Line Interface (CLI).

Specify the Location of Compiler Components

Before you invoke the compiler, you may need to set certain environment variables that define the location of compiler-related components. The compiler includes environment configuration scripts to configure your build and development environment variables:

- On Linux, the file is a shell script called `setvars.sh`.
- On Windows, the file is a batch file called `setvars.bat`.

> **NOTE** The Intel oneAPI DPC++/C++ Compiler is designed and tested only for use on 64-bit Linux and Windows operating systems, 32-bit operating systems are not supported. Additionally, the macOS operating system is not supported by the compiler.

Linux

Set the environment variables before using the compiler by sourcing the shell script `setvars.sh`. Depending on the shell, you can use the `source` command or a `.` (dot) to source the shell script, according to the following rules for a `.sh` script:

Using source:

```
source /<install-dir>/setvars.sh <arg1> <arg2> … <argn>
```

Example:

```
source /opt/intel/oneapi/setvars.sh intel64
```

Using `.` (dot):

```
. /<install-dir>/setvars.sh <arg1> <arg2> … <argn>
```

Example:

```
. /opt/intel/oneapi/setvars.sh intel64
```

Use `source /<install-dir>/setvars.sh --help` for more `setvars` usage information.

The compiler environment script file accepts an optional target architecture argument `<arg>`:

- `intel64`: Generate code and use libraries for Intel® 64 architecture-based targets.
- `ia32`: Generate code and use libraries for IA-32 architecture-based targets.

If you want the `setvars.sh` script to run automatically in all of your terminal sessions, add the source `setvars.sh` command to your startup file. For example, inside your `.bash_profile` entry for Intel® 64 architecture targets.

If the proper environment variables are not set, errors similar to the following may appear when attempting to execute a compiled program:

```
./a.out: error while loading shared libraries:
libimf.so: cannot open shared object file: No such file or directory
```

Windows

Under normal circumstances, you do not need to run the `setvars.bat` batch file. The terminal shortcuts in the Windows **Start** menu, **Intel oneAPI command prompt for <target architecture> for Visual Studio <year>**, set these variables automatically.

For additional information, see Use the Command Line on Windows.

You need to run the `setvars` batch file if a command line is opened without using one of the provided **Command Prompt** menu items in the **Start** menu, or if you want to use the compiler from a script of your own.

The `setvars` batch file inserts DLL directories used by the compiler and libraries at the beginning of the existing `Path`. Because these directories appear first, they are searched before any directories that were part of the original `Path` provided by Windows (and other applications). This is especially important if the original `Path` includes directories with files that have the same names as those added by the compiler and libraries.

The `setvars` batch file takes multiple optional arguments; the following two arguments are recognized for compiler and library initialization:

```
<install-dir>\setvars.bat [<arg1>] [<arg2>]
```

Where *<arg1>* is optional and can be one of the following:

- `intel64`: Generate code and use libraries for Intel® 64 architecture (host and target).
- `ia32`: Generate code and use libraries for IA-32 architecture (host and target).

The *<arg2>* is optional. If specified, it is one of the following:

- `vs2022`: Microsoft Visual Studio 2022
- `vs2019`: Microsoft Visual Studio 2019
- `vs2017`: Microsoft Visual Studio 2017.

> **NOTE** Support for Microsoft Visual Studio 2017 is deprecated as of the Intel® oneAPI 2022.1 release and will be removed in a future release.

If *<arg1>* is not specified, the script uses the `intel64` argument by default. If *<arg2>* is not specified, the script uses the highest installed version of Microsoft Visual Studio detected during the installation procedure.

See Also
oneAPI Development Environment Setup

Configure Your CPU or GPU System

Invoke the Compiler

Requirements Before Using the Command Line

You may need to set certain environment variables before using the command line. For more information, see Specify the Location of Compiler Components.

Different Compilers and Drivers

The table below provides the different compiler front-end and driver information.

Compiler	Notes	Linux Driver	Windows Driver
Intel® DPC++ Compiler	A C++ and SYCL compiler with a Clang front-end. To use Microsoft Visual C++ (MSVC) compatible options, use `dpcpp-cl`.	`dpcpp`	`dpcpp` (`clang` compatible) `dpcpp-cl` (`clang-cl` compatible)
Intel® C++ Compiler	A C++ compiler with a Clang front-end, that supports an OpenMP offload.	`icx` for C `icpx` for C++	`icx`

Use the Compiler from the Command Line

Use the compiler with the OS/language specific invocations below.

Linux

Invoke the compiler using `icx`/`icpx` or `dpcpp` to compile C/C++/DPC++ source files.

- When you invoke the compiler with `dpcpp` the compiler builds C++ source files with SYCL using SYCL libraries and SYCL include files. If you use `dpcpp` with a C source file, it is compiled as a SYCL file. Use `dpcpp` to link SYCL object files.
- When you invoke the compiler with `icx` the compiler builds C source files using LLVM C libraries and C include files. If you use `icx` with a C++ source file, it is compiled as a C++ file. Use `icx` to link C object files.
- When you invoke the compiler with `icpx` the compiler builds C++ source files using C++ libraries and C++ include files. If you use `icpx` with a C source file, it is compiled as an C++ file. Use `icpx` to link C++ object files.

The `icx`, `icpx`, or `dpcpp` command:

- Compiles and links the input source file(s).
- Produces one executable file, `a.out`, in the current directory.

Windows

You can invoke the compiler on the command line using `icx` or `dpcpp-cl`. This command:

- Compiles and links the input source file(s).
- Produces object file(s) and assigns the names of the respective source file(s), but with a `.obj` extension.
- Produces one executable file and assigns it the name of the first input file on the command line, but with a `.exe` extension.
- Places all the files in the current directory.

When compilation occurs with the compiler, many tools may be called to complete the task that may reproduce diagnostics unique to the given tool. For instance, the linker may return a message if it cannot resolve a global reference.

Command Line Syntax

The syntax to invoke the compiler is:

Linux

```
icpx [option] file1 [file2...]
```

Windows

```
icx [option] file1 [file2...]
```

Argument	Description
`option`	Indicates one or more command line options. On Linux systems, the compiler recognizes one or more letters preceded by a hyphen (`-`). On Windows, options are preceded by a hyphen (`-`) or slash (`/`). This includes linker options. Options are not required when invoking the compiler. The default behavior of the compiler implies that some options are ON by default when invoking compiler.
`file1, file2...`	Indicates one or more files to be processed by the compiler.
`/link` (Windows)	All options following `/link` are passed to the linker. Compiler options must precede `link` if they are not to be passed to the linker.

Other Methods for Using the Command Line to Invoke the Compiler

- **Using makefiles from the Command Line:** Use makefiles to specify a number of files with various paths and to save this information for multiple compilations. For more information on using makefiles, see Use Makefiles to Compile Your Application.
- **Using a Batch File from the Command Line:** Create and use a `.bat` file to execute the compiler with a desired set of options instead of retyping the command each time you need to recompile.

See Also

Specify the Location of Compiler Components
Understand File Extensions
Use Eclipse
Use Microsoft Visual Studio
Use Makefiles to Compile Your Application
watch compiler option

Use the Command Line on Windows

The compiler provides a shortcut to access the command line with the appropriate environment variables already set.

To invoke the compiler from the command line:

1. Open the Windows **Start** menu.
2. Scroll down the list of apps (programs) in the **Start** menu and find the **Intel oneAPI 2021** folder.
3. Left click on the folder name and select your component. The command prompts shown are dependent on the versions of Microsoft Visual Studio you have installed on your machine.
4. Right click on the command prompt icon to pin it to your taskbar. This step is optional.
5. The command line opens.

You can use any command recognized by the Windows command prompt, plus some additional commands.

Because the command line runs within the context of Windows, you can easily switch between the command line and other applications for Windows or have multiple instances of the command line open simultaneously.

When you are finished working in a command line, use the **exit** command to close and end the session.

File Extensions

Input File Extensions

The Intel® oneAPI DPC++/C++ Compiler recognizes input files with the extensions listed in the following table:

File Name	Interpretation	Action
`file.c`	C source file	Passed to compiler
`file.C` `file.CC` `file.cc` `file.cpp` `file.cxx`	C++ source file	Passed to compiler
`file.lib` (Windows) `file.a` `file.so` (Linux)	Library file	Passed to linker
`file.i`	Preprocessed file	Passed to compiler
`file.obj` (Windows) `file.o` (Linux)	Object file	Passed to linker
`file.asm` (Windows) `file.s` (Linux) `file.S` (Linux)	Assembly file	Passed to assembler

Output File Extensions

The Intel® oneAPI DPC++/C++ Compiler produces output files with the extensions listed in the following table:

File Name	Description
`file.i`	Preprocessed file: Produced with the `-E` option.
`file.o` (Linux) `file.obj` (Windows)	Object file: Produced with the `-c` (Linux and Windows) object. The `/Fo` (Windows) or `-o` (Linux) option allows you to rename the output object file.
`file.s` (Linux) `file.asm` (Windows)	Assembly language file: Produced with the `-S` option. The `/Fa` (Windows) or `-s` (Linux) option allows you to rename the output assembly file.
`a.out` (Linux) `file.exe` (Windows)	Executable file: Produced by the default compilation. The `/Fe` (Windows) or `-o` (Linux) option allows you to rename the output executable file.

See Also

Invoke the Compiler
Specify Compiler Files

Use Makefiles for Compilation

This topic describes the use of makefiles to compile your application. You can use makefiles to specify a number of files with various paths, and to save this information for multiple compilations.

Use Makefiles to Store Information for Compilation on Linux

To run `make` from the command line using the compiler, make sure that `/usr/bin` and `/usr/local/bin` are in your *PATH* environment variable.

If you use the C shell, you can edit your `.cshrc` file and add the following:

```
setenv PATH /usr/bin:/usr/local/bin:$PATH
```

To use the compiler, your makefile must include the setting `CC=icx`, `CC=icpx`, or `CC=dpcpp`. Use the same setting on the command line to instruct the makefile to use the compiler. If your makefile is written for GCC, you need to change the command line options that are not recognized by the compiler. Run `make`, using the following syntax:

```
make -f yourmakefile
```

Where `-f` is the `make` command option to specify a particular makefile name.

Use Makefiles to Store Information for Compilation on Windows

To use a makefile to compile your source files, use the `nmake` command with the following syntax:

```
nmake /f [makefile_name.mak] CPP=[compiler_name] [LINK32=[linker_name]
```

Example:

```
nmake /f your_project.mak CPP=icx LINK32=link
```

NOTE If you have `link/xilink` specific options that are not accepted by `dpcpp-cl`, ensure any linker specific options are placed after the `/link` option. For example: `dpcpp test.obj <compiler options> /link <linker options>`

Argument	Description
`/f`	The `nmake` option to specify a makefile.
`your_project.mak`	The makefile used to generate object and executable files.
`CPP`	The preprocessor/compiler that generates object and executable files. (The name of this macro may be different for your makefile.)
`LINK32`	The linker that is used.

The `nmake` command creates object files (`.obj`) and executable files () from the information specified in the `your_project.mak` makefile.

See Also

Modify Your makefile (Linux)
Modify Your makefile (Windows)

Use CMake with the Intel® oneAPI DPC++/C++ Compiler

Linux

Using CMake with the compiler on Linux is supported. When you are using CMake, the compiler is enabled using the `icx` (variant) binary. You may need to set your `CC/CXX` or `CMAKE_C_COMPILER / CMAKE_CXX_COMPILER` string to `icx/icpx`. For example:

```
cmake -DCMAKE_C_COMPILER=icx -DCMAKE_CXX_COMPILER=icpx …
```

Windows

Using CMake with the compiler on Windows is supported. When you are using CMake, the compiler is enabled using the `icx` (variant) binary. You may need to set your `CC/CXX` or `CMAKE_C_COMPILER / CMAKE_CXX_COMPILER` string to `icx`. The supported generator in the Windows environment is Ninja. For example:

```
cmake -DCMAKE_C_COMPILER=icx -DCMAKE_CXX_COMPILER=icx -GNinja …
```

NOTE If your Microsoft Visual Studio default CMake version is older than 3.21, you need to install CMake 3.21 (or above) and update Microsoft Visual Studio with the new CMake executable. Edit the `CMakeSettings.json` file for this update.

Support

Use the following steps to enable the compiler for your project:

1. Add the following snippets to your project's `CMakeLists.txt`:

```
cmake_minimum_required(VERSION 3.21.0)
```

And:

```
find_package(IntelDPCPP REQUIRED)
```

The second snippet enables the `dpcpp` compiler. The heterogeneous compilation configuration package (`IntelDPCPPConfig.cmake`) is shipped with the compiler. The package directory is found in the parent directory of the `icx` bin directory.

2. Select the appropriate compilers for C or C++. See the Linux and Windows sections above for specific settings.

3. Run CMake and build your applications as normal.

4. The heterogeneous compilation configuration package exposes other variables that may be required. Refer to the package for more information.

Use Compiler Options

A compiler option is a case-sensitive, command line expression used to change the compiler's default operation. Compiler options are not required to compile your program, but they can control different aspects of your application, such as:

- Code generation
- Optimization
- Output file (type, name, location)
- Linking properties
- Size of the executable
- Speed of the executable

Linux

When you specify compiler options on the command line, the following syntax applies:

```
[invocation] [option] [@response_file] file1 [file2...]
```

The *invocation* is `icx`, `icpx`, or `dpcpp`.

The *option* represents zero or more compiler options and the *file* is any of the following:

- C or C++ source file (`.C`, `.c`, `.cc`, `.cpp`, `.cxx`, `.c++`, `.i`, `.ii`)
- Assembly file (`.s`, `.S`)
- Object file (`.o`)
- Static library (`.a`)

When compiling C language sources, invoke the compiler with icx. When compiling C++ language sources or a combination of C and C++, invoke the compiler with icpx. When compiling SYCL-based sources, invoke the compiler with `dpcpp`.

Windows

When you specify compiler options on the command line, the following syntax applies:

```
[invocation] [option] [@response_file] file1 [file2 ...] [/link linker_option]
```

The *invocation* is `icx` or `dpcpp-cl`.

The *option* represents zero or more compiler options, the *linker_option* represents zero or more linker options, and the *file* is any of the following:

- C or C++ source file (`.c`, `.cc`, `.ccp`, `.cxx`, `.i`)
- Assembly file (`.asm`)
- Object (`.obj`)
- Static library (`.lib`)

The optional *response_file* is a text file that lists the compiler options you want to include during compilation. See Use Response Files for additional information.

Default Operation

The compiler invokes many options by default. In this example, the compiler includes the option `O2` (and other default options) in the compilation. Using C++ as an example:

Linux

```
icpx main.c
```

Windows

```
icx main.c
```

Each time you invoke the compiler, options listed in the corresponding configuration file override any competing default options. For example, if your configuration file includes the `O3` option, the compiler uses `O3` rather than the default `O2` option. Use the configuration file to list the options for the compiler to use for every compilation. See Using Configuration Files.

> **NOTE** The default `.cfg` files are not valid for the compiler. You can use the `-config<name>` option instead of a default `.cfg` file. `<name>` can be a configuration file that is in the `bin` directory, or you can use the full path your selected `.cfg` file.

Options specified in the command line environment variable override any competing default options and options listed in the configuration file.

Finally, options used on the command line override any competing options that may be specified elsewhere (default options, options in the configuration file, and options specified in the command line environment variable). If you specify the option `O1` this option setting takes precedence over competing option defaults and competing options in the configuration files, in addition to the competing options in the command line environment variable.

Certain `#pragma` statements in your source code can override competing options specified on the command line. If a function in your code is preceded by `#pragma optimize("", off)`, then optimization for that function is turned off. The override is valid even when the `O2` optimization is on by default, the `O3` is listed in the configuration file, and the `O1` is specified on the command line for the rest of the program.

Use Competing Options

The compiler reads command line options from left to right. If your compilation includes competing options, then the compiler uses the one furthest to the right. Using C++ as an example:

Linux

```
icpx -xSSSE3 main.c file1.c -xSSE4.2 file2.c
```

Windows

```
icx /QxSSSE3 main.c file1.c /QxSSE4.2 file2.c
```

You can compile for SYCL by adding `-fsycl` after `icpx`.

The compiler sees `[Q]xSSSE3` or `O1` and `[Q]xSSE4.2` or `O2` as two forms of the same option, where only one form can be used. Since `[Q]xSSE4.2` or `O2` are last (furthest to the right), they are used.

All options specified on the command line are used to compile each file. The compiler does not compile individual files with specific options.

A rare exception to this rule is the `-x` *type* option on Linux. Using C++ as an example:

Linux

```
icpx -x c file1 -x c++ file2 -x assembler file3
```

The *type* argument identifies each file type for the compiler.

Use Options with Arguments

Compiler options can be as simple as a single letter, such as the option `E`. Many options accept or require arguments. The `O` option, for example, accepts a single-value argument that the compiler uses to determine the degree of optimization. Other options require at least one argument and can accept multiple arguments. For most options that accept arguments, the compiler warns you if your option and argument are not recognized. If you specify `O9`, the compiler issues a warning, then ignores the unrecognized option `O9`, and proceeds with the compilation.

The `O` option does not require an argument, but there are other options that must include an argument. The `I` option requires an argument that identifies the directory to add to the include file search path. If you use this option without an argument, the compiler will not finish its compilation.

Other Forms of Options

You can toggle some options on or off by using the negation convention. For example, the `[Q]ipo` option (and many others) includes negation forms, `-no-ipo` (Linux) and `/Qipo-` (Windows), to change the state of the option.

Option Categories

When you invoke the Intel oneAPI DPC++/C++ Compiler and specify a compiler option, you have a wide range of choices to influence the compiler's default operation. Intel oneAPI DPC++/C++ Compiler options typically correspond to one or more of the following categories:

- Advanced Optimization
- Code Generation
- Compatibility
- Compiler Diagnostics
- Component Control (Not available for device compilation.)
- Data
- Floating Point
- Inlining
- Interprocedural Optimizations (IPO)
- Language
- Linking/Linker
- Miscellaneous
- Offload Compilation, OpenMP, and Parallel Processing
- OpenMP and Parallel Processing
- Optimization
- Optimization Report
- Output
- Preprocessor

See Also

qopt-report, Qopt-report
Use Configuration Files

Specify Compiler Files

Specify Include Files

The compiler searches the default system areas for include files and items specified by the `I` compiler option. The compiler searches directories for include files in the following order:

1. Directories specified by the `I` option.
2. Directories specified in the environment variables.
3. Default include directories.

Use the `-nostdinc` (Linux) or `X` (Windows) option to remove the default directories from the include file search path.

For example, to direct the compiler to search the path `/alt/include` instead of the default path, use the following:

Linux

```
icpx -nostdinc -I/alt/include prog1.cpp
```

Windows

```
icx /X /I\alt\include prog1.cpp
```

Specify Assembly Files

You can use the `-S` and `-o` options (Linux) or `/Fa` option (Windows) to specify an alternate name for an assembly file. The compiler generates an assembly file named `myasm.s` (Linux) or `myasm.asm` (Windows):

Linux

```
icpx -S -o myasm.s x.cpp
```

Windows

```
icx /Famyasm x.cpp
```

Specify Object Files

You can use the `-c` and `-o` options (Linux) or `/Fo` option (Windows) to specify an alternate name for an object file. In this example, the compiler generates an object file name `myobj.o` (Linux) or `myobj.obj` (Windows):

Linux

```
icpx -c -o myobj.o x.cpp
```

Windows

```
icx /Fomyobj x.cpp
```

See Also

`-c` compiler option
`/Fa` compiler option
`/Fo` compiler option
`I` compiler option
`-o` compiler option
`-S` compiler option
`X` compiler option

Supported Environment Variables

Convert Projects to Use a Selected Compiler

You can use the command-line interface `ICProjConvert<version>.exe` to transform your Intel® C++ projects into Microsoft Visual C++ projects, or vice versa. The syntax is:

```
ICProjConvert<version>.exe <sln_file | prj_files> </VC[:"VCtoolset name"] | /IC[:"ICtoolset
name"]> [/q] [/nologo] [/msvc] [/s] [/f]
```

Where:

Parameter	Description
`version`	The `ICProjConvert` version number. Values are: 191 or 192.
`sln_file`	A path to the solution file, which should be modified to use a specified project system.
`prj_files`	A space separated list of project files (or wildcard), which should be modified to use specified project system.
`/VC`	Convert to use the Microsoft Visual C++ project system.
VCtoolset name	The possible values are `v141` (Microsoft Visual Studio 2017), `v142` (Microsoft Visual Studio 2019), or `v143` (Microsoft Visual Studio 2022). **NOTE** Support for Microsoft Visual Studio 2017 is deprecated as of the Intel® oneAPI 2022.1 release, and will be removed in a future release.
`/IC`	Convert to use the Intel® C++ project system.
ICtoolset name	Such as `Intel C++ Compiler 2021.1` Depending on the integration version, the supported name values may be different.
`/q`	Starts quiet mode, all information messages (except errors) are hidden.
`/nologo`	Suppresses the startup banner.
`/msvc`	Sets the compiler to Microsoft Visual C++.
`/s`	Searches the project files through all subdirectories.
`/f`	Forces an update to the project even if it has an unsupported type or unsupported properties.
`/?` or `/h`	Shows help.

Example

To convert all Intel® C++ project files to use Microsoft Visual C++ in your current directory and its subdirectories, use the command:

```
ICProjConvert<version>.exe *.icproj /s /VC
```

NOTE If you uninstall the Intel® oneAPI DPC++/C++ Compiler, `ICProjConvert<version>.exe` remains in the folder `Program Files (x86)\Common Files\Intel\shared files\ia32\Bin`. You can use it to transform Intel® C++ projects back into Microsoft Visual C++.

Use Eclipse

The Intel® oneAPI DPC++/C++ Compiler for Linux provides integrations for the compiler to Eclipse and C/C++ Development Tooling (CDT) that let you develop, build, and debug your Intel oneAPI DPC++/C++ Compiler projects in an integrated development environment (IDE).

Eclipse is an open source software development project dedicated to providing a robust, full-featured, commercial-quality, industry platform for the development of highly integrated tools. It is an extensible, open source integrated development environment (IDE). CDT is a complete C/C++ IDE for the Eclipse platform, which allows you to develop, build, and run projects in a visual, interactive environment. CDT is layered on Eclipse and provides a C/C++ development environment perspective.

NOTE
Eclipse and CDT are not bundled with the Intel® oneAPI DPC++/C++ Compiler. They must be obtained separately.

If you used `sudo sh ./<installer>.sh` to install the Intel® oneAPI toolkits, use `sudo ./eclipse` to open Eclipse as a root user.

If you used `sh ./<installer>.sh` to install the Intel® oneAPI toolkits, use `./eclipse` to open Eclipse as a current user.

If you attempt to open Eclipse as a current user after installing as a root user, the integration will not be available.

Add the Compiler to Eclipse

This step is needed only if you are manually installing the Intel® oneAPI DPC++/C++ Compiler plug-in for Eclipse.

To add the Intel oneAPI DPC++/C++ Compiler product extension to your Eclipse configuration:

1. Start Eclipse.
2. Select **Help** > **Install New Software**.
3. Next to the **Work with** field, click the **Add** button. The **Add Repository** dialog box opens.
4. Click the **Archive** button and browse to the `<install_dir>/compiler/<version>/linux/ide_support` directory. Select the `.zip` file that starts with `com.intel.compiler` for C/C++ or `com.intel.dpcpp.compiler` for DPC++, then click **OK**.
5. Select **Intel® Software Development Tools** > **Intel® C++ Compiler Integration** for C/C++ or **Intel® oneAPI DPC++ Compiler Integration** > **Intel® oneAPI DPC++ Compiler Integration** for DPC++, then click **OK**.
6. Follow the installation instructions.
7. When asked if you want to restart Eclipse, select **Yes**.

When Eclipse restarts, you can create and work with CDT projects that use the Intel oneAPI DPC++/C++ Compiler.

Multi-version Compiler Support

You can select different versions of the Intel® oneAPI DPC++/C++ Compiler for compiling projects with the Eclipse Integrated Development Environment (IDE). For a list of the currently supported compiler versions by platform, refer to the Release Notes.

If multiple versions of the compiler are installed on the system, Eclipse uses the latest version by default. To select the version of the compiler to build your project:

1. Right click the project and open **Properties**.
2. In the properties dialog box, select **C/C++ Build** > **Settings**.
3. Select the **Intel(R) oneAPI DPC++ Compiler** for a DPC++ project, or the **Intel® C++ Compiler** for a C++ project tab.
4. Select the row with the desired compiler version.
5. Click **Use Selected**. Alternatively, click **Use Latest** to select the latest version of compiler.
6. Click **Apply**.

The corresponding compiler environment is configured automatically for your project.

Use **Settings** and **Tool Chain Editor** to select tools to be used within the toolchain, or set distinct project properties, like compiler options, to be used with different versions of the compiler.

For any project, you can set the compiler environment by specifying it within Eclipse; this overrides any other environment specifications for the compiler.

Use Cheat Sheets

The Intel® oneAPI DPC++/C++ Compiler integration includes several Eclipse* cheat sheets that can guide you through various compilation and debugging tasks.

To view a list of available cheat sheets and select one:

1. Select **Help** > **Cheat Sheets**.
   The **Cheat Sheet Selection** dialog box opens, displaying a list of available cheat sheets.
2. Select a cheat sheet. Cheat sheets located outside of the Eclipse* integration can be entered in the **Select a cheat sheet from a file** or **Enter the URL of a cheat sheet**.
   Intel cheat sheets are located under **Intel(R) C++ Compiler**. A description of the cheat sheet appears in the lower pane.
3. To open a cheat sheet, click **OK**.

The **Cheat Sheets** view opens in the Eclipse window.

Create a Simple Eclipse Project

The sections below show you how to create a simple project using Eclipse.

Create a New Eclipse Project

To create an Eclipse project:

1. Select **File** > **New** > **Project...** The **New Project** wizard opens.
2. Expand the **C/C++ Project** tab and select the appropriate project type. Click **Next** to continue.
3. For **Project name**, enter hello_world. Deselect the **Use default location** to specify a directory for the new project.
4. In the **Project Type** list, expand the **Executable** project type and select **Hello World C++ Project** for C++ or **Hello World DPC++ Project** for DPC++.
5. In the **Toolchains** list, select **Intel(R) oneAPI DPC++ Compiler** for a DPC++ project, or the **Intel C ++ Compiler** for a C++ project. Click **Next**.

NOTE

- If you need to see the toolchains for the compilers that are not locally installed, uncheck **Show project types and toolchains only if they are supported on the platform**. You are only able to view and configure these toolchains if the proper compilers are installed.
- If you have multiple versions of the compiler installed, they appear in the project's properties under **C/C++ Build** > **Settings** on the **Intel(R) oneAPI DPC++ Compiler** tab for a DPC++ project, or the **Intel C++ Compiler** tab for a C++ project.

6. The **Basic Settings** page allows specifying template information, including **Author** and **Copyright notice**, which appear as a comment at the top of the generated source file. After entering desired fields, click **Next**.
7. The **Select Configurations** page allows specifying deployment platforms and configurations. By default, a **Debug** and **Release** configuration is created for the selected toolchain. Select no (**Deselect all**), multiple, or all (**Select all**) configurations. To edit project properties, click the **Advanced settings** button. Click **Finish** to create the `hello_world` project. Configurations can be created after the project is created by selecting **Project** > **Properties**.
8. If the view is not the **C/C++ Development Perspective** (default), an **Open Associated Perspective** dialog box opens. In the **C/C++ Perspective**, click **Yes** to proceed.

An entry for your `hello_world` project appears in the **Project Explorer** view.

Add a C Source File

To add a source file to the `hello_world` project:

1. Select the `hello_world` project in the **Project Explorer** view.
2. Select **File** > **New** > **Source File**. The **New Source File** dialog box opens. The dialog box automatically populates the source folder for the source file to be created. You can change this by entering a new location or selecting **Browse**.
3. Enter `new_source_file.c` in the **Source File** field.
4. Select a **Template** from the drop-down list or **Configure** a new template.
5. Click **Finish** to add the file to the `hello_world` project.
6. In the **Editor** view, add your code for `new_source_file.c`.
7. When your code is complete, **Save** your file.

Set Options for a Project or File

You can specify compiler, linker, and archiver options at the project and source file level. Follow these steps to set options for a project or file:

1. Right-click a project or source file in the **Project Explorer**.
2. Select **Properties**. The property pages dialog box opens.
3. Select **C/C++ Build** > **Settings**.
4. Select the **Tool Settings** tab and click an option category for **Intel C Compiler**, **Intel C++ Compiler**, or **Intel C++ Linker** for a C++ project, or select **Intel® oneAPI DPC++ Compiler** or **Intel® oneAPI DPC++ Linker** for a DPC+++ project.
5. Set the options to apply to the project or file.

NOTE

- Some properties use check boxes, drop-down boxes, or dialog boxes to specify compiler options. For a description on options properties, hover over the option to display a tooltip. After setting the desired options in command line syntax, select **Apply**.
- To specify an option that is not available from the **Properties** dialog, use **C/C++ Build Settings** > **Settings** > **<Compiler>** > **Command Line**. Enter the command line options in the **Additional Options** field using command-line syntax and select **Apply**.
- You can specify option settings for one or more configurations by using the **Configuration** drop-down menu.

6. Click **Apply and Close**.

The compiler applies the selected options, using the selected configurations, when building. To restore default settings to all properties for the selected configuration, click the **Restore Defaults** button on any property page.

Exclude Source Files from a Build

To exclude a source file from a build:

1. Right-click a file or folder in the **Project Explorer**.
2. Select **Resource Configurations** > **Exclude from build**. The **Exclude from build** dialog box opens.
3. Select one or more build configurations to exclude the file or folder from.
4. Click **OK**.

The compiler excludes that file or folder when it builds using the selected project configuration.

Build a Project

To build your project:

1. Select the `hello_world` project in the **Project Explorer** view.
2. Select **Project** > **Build Project**.

See the **Build** results in the **Console** view.

For a C/C++ project, use:

```
**** Build of configuration Debug for project hello_world ****
make all
Building file: ../src/hello_world.cpp
Invoking: Intel C++ Compiler
icpx -g -O0 -MMD -MP -MF"src/hello_world.d" -MT"src/hello_world.d" -c -o "src/hello_world.o"
"../src/hello_world.cpp"
Finished building: ../src/hello_world.cpp

Building target: hello_world
Invoking: Intel C++ Linker
icpx -O0 -o "hello_world"  ./src/hello_world.o
Finished building target: hello_world

Build Finished. 0 errors, 0 warnings.
```

For a DPC++ project, use:

```
**** Build of configuration Debug for project DPCPPhelloworld ****
make all
Building file: ../main.cpp
Invoking: Intel(R) oneAPI DPC++ Compiler
dpcpp -g -Wall -O0 -I/home/sys_idebuilder/eclipse-workspace/DPCPPhelloworld -MMD -MP -c -o
```

```
"main.o" "../main.cpp"
Finished building: ../main.cpp

Building target: DPCPPhelloworld
Invoking: Linker
dpcpp  -o "DPCPPhelloworld"  ./main.o   -lsycl -lOpenCL
Finished building target: DPCPPhelloworld

Build Finished. 0 errors, 0 warnings.
```

Detailed descriptions of errors, warnings, and other output can be viewed by selecting the **Problems** tab.

Run a Project

After building a project, you can run your project by following these steps:

1. Select the `hello_world` project in the **Project Explorer** view.
2. Select **Run As** > **Local C/C++ Application**.

When the executable runs, the output appears in the **Console** view.

Error Parser

The Error Parser (selected by default) lets you track compile-time errors in Eclipse. To confirm that the Error Parser is active:

1. Select the `hello_world` project in the **Project Explorer** view.
2. Select **Project** > **Properties**.
3. In the **Properties** dialog box, select **C/C++ Build** > **Settings**.
4. Click the **Error Parsers** tab. Make sure that **Intel C++ Error Parser** is checked, and **CDT Visual C Error Parser** or **Microsoft Visual C Error Parser** are not checked.
5. Click **OK** to update your choices, if you have changed any settings.

Use the Error Parser

The Error Parser automatically detects and manages the diagnostics generated by the Intel® oneAPI DPC++/C++ Compiler.

If an error occurring in the `hello_world.c` program is compiled, for example, `#include <xstdio.h>`, the error is reported in the **Problems** view next to an error marker.

You can double-click each error in the **Problems** view to highlight the source line, which causes the error in the **Editor** view.

Correct the error, then rebuild your project.

Makefiles

This section provides information about makefile project types and exporting makefiles.

Project Types and Makefiles

When you create a new project in Eclipse*, there are **Executable**, **Shared Library**, **Static Library**, or **Makefile** project types available for your selection.

- Select **Makefile Project** if the project already includes a makefile.
- Use **Executable**, **Shared Library**, or **Static Library Project** to build a makefile using options assigned from property pages specific to the Intel® oneAPI DPC++/C++ Compiler.

Export Makefiles

Eclipse can build a makefile that includes Intel® oneAPI DPC++/C++ Compiler options for created **Executables**, **Shared Libraries**, or **Static Library** Projects. When the project is complete, export the makefile and project source files to another directory, and then build the project from the command line using `make`.

To export the makefile:

1. Select the project in the Eclipse **Project Explorer** view.
2. Select **File** > **Export** to launch the **Export Wizard**. The **Export** dialog box opens, showing the **Select** screen.
3. Select **General** > **File system**, then click **Next**. The **File System** screen opens.
4. Check both the **hello_world** and **Release** directories in the left-hand pane. Ensure all project sources are checked in the right-hand pane.

   NOTE Some files in the right-hand pane may be deselected, such as the `hello_world.o` object file and the `hello_world` executable. **Create directory structure for files** in the **Options** section must be selected to successfully create the export directory. This process applies to project files in the `hello_world` directory.

5. Use the **Browse** button to target the export to an existing directory. Eclipse can create a directory for full paths entered in the **To directory** text box. For example, if the `/code/makefile` is specified as the export directory, Eclipse creates two new subdirectories:
   - `/code/makefile/hello_world`
   - `/code/makefile/hello_world/Release`
6. Click **Finish** to complete the export.

Run Make

From the command line, change to your project directory and run `make` with:

```
make clean all
```

You should see the following output:

```
rm -rf  ./new_source_file.o  ./new_source_file.d  hello_world

Building file: ../new_source_file.c
Invoking: Intel C++ Compiler
icx -O2 -MMD -MP -MF"new_source_file.d" -MT"new_source_file.d" -c -o "new_source_file.o" "../
new_source_file.c"
Finished building: ../new_source_file.c

Building target: hello_world
Invoking: Intel C++ compiler
icx  -o "hello_world"  ./new_source_file.o
Finished building target: hello_world
```

This process generates the `hello_world` executable in the project directory.

Use Intel Libraries with Eclipse

You can use the compiler with the following Intel Libraries, which that may be included as a part of the product:

- Intel® oneAPI Data Analytics Library (oneDAL)
- Intel® Integrated Performance Primitives (Intel® IPP)
- Intel® oneAPI Math Kernel Library (oneMKL)
- Intel® oneAPI Threading Building Blocks (oneTBB)

To access these libraries in Eclipse, use the property pages:

1. Select your project.
2. Open the property pages from **Project** > **Properties** and select **C/C++ Build** > **Settings**.
3. Select **Intel C/C++ Compiler** > **Performance Library Build Components**

   for C++ projects, or **Intel® oneAPI DPC++ Compiler** > **Performance Library Build Components** for DPC++ projects.

The **Use Intel® oneAPI Data Analytics Library** (oneDAL) property allows enabling the library and bringing in the associated headers.

- **None**: Disable Use of oneDAL.
- **Use threaded Intel® oneDAL**: Link using the threaded version of the library.
- **Use non-threaded Intel® oneDAL**: Link using the non-threaded version of the library.

The **Use Intel® Integrated Performance Primitives Libraries** property provides the following options in a drop-down menu:

- **None**: Disable use of Intel® IPP.
- **Use main libraries set**: Link using the main libraries set.
- **Use non-pic version of libraries**: Link using the version of the libraries that do not have position-independent code.
- **Use main libraries and cryptography library**: Link using main or cryptography libraries.

The **Use Intel® oneAPI Math Kernel Library** property provides the following options in a drop-down menu:

- **None**: Disables the use of the oneMKL.
- **Use threaded Intel® oneMKL library**: Link using the threaded version of the library.
- **Use non-threaded Intel® oneMKL library**: Link using the non-threaded version of the library.
- **Use Intel® oneMKL Cluster and sequential Intel® oneMKL libraries**: Link using the oneMKL Cluster Edition libraries and the sequential oneMKL libraries.

> **NOTE** The option value **Use Intel® oneMKL Cluster and sequential Intel® oneMKL libraries** is only available for Intel C Compiler or Intel C++ Compiler.

The **Use Intel® oneAPI Threading Building Blocks** on the **Property** page allows enabling the library and bringing in the associated headers.

For more information, see the oneDAL, Intel® IPP, oneMKL, and oneTBB documentation.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Use Microsoft Visual Studio

You can use the Intel® oneAPI DPC++/C++ Compiler within the Microsoft Visual Studio integrated development environment (IDE) to develop C++ or DPC++ applications, including static library (`.LIB`), dynamic link library (`.DLL`), and main executable (`.EXE`) applications. This environment makes it easy to create, debug, and execute programs. You can build your source code into several types of programs and libraries, using the IDE or from the command line.

The IDE offers these major advantages:

- Makes application development quicker and easier by providing a visual development environment.
- Provides integration with the native Microsoft Visual Studio debugger.
- Makes other IDE tools available.

Find Product Information

To access the product information for the Intel® oneAPI DPC++ Compiler:

1. Open **Help** > **About Microsoft Visual Studio**
2. In the left pane, select **Intel® oneAPI DPC++ Compiler - Package ID: [_package ID_]**.
3. In the bottom pane, product details will show: **Intel® oneAPI DPC++ Compiler - toolkit version [_toolkit version_], extension version [_extension version_], Package ID: [_package ID_], Copyright © [_copyright year_] Intel Corporation. All rights reserved.** Other names and brands may be claimed as the property of others.

To access the product information for the Intel® C++ Compiler:

1. Open **Help** > **About Microsoft Visual Studio**
2. In the left pane, select **Intel® C++ Compiler - Package ID: [_package ID_]**.
3. In the bottom pane, product details will show: **Intel® C++ Compiler - toolkit version [_toolkit version_], extension version [_extension version_], Package ID: [_package ID_], Copyright © [_copyright year_] Intel Corporation. All rights reserved.** Other names and brands may be claimed as the property of others.

To access the product information for the Intel Libraries for oneAPI:

1. Open **Help** > **About Microsoft Visual Studio**
2. In the left pane, select **Intel Libraries for oneAPI - Package ID: [_package ID_]**.
3. In the bottom pane, product details will show: **Intel Libraries for oneAPI - toolkit version [_toolkit version_], extension version [_extension version_], Package ID: [_package ID_], Copyright © [_copyright year_] Intel Corporation. All rights reserved.** Other names and brands may be claimed as the property of others.

Create a New Project

Create a New Project

When you create a project, Microsoft Visual Studio automatically creates a corresponding solution to contain it. To create a new Intel® oneAPI DPC++/C++ project using Microsoft Visual Studio:

NOTE Exact steps may vary depending on the version of Microsoft Visual Studio in use.

For a C/C++ project:

1. Select **File** > **New** > **Project**.
2. In the left pane, expand **Visual C++** and select **Windows Desktop**.
3. In the right pane, select **Windows Console Application**.
4. Accept or specify a project name in the **Name** field. For this example, use hello32 as the project name.
5. Accept or specify the Location for the project directory. Click **OK**.

For a DPC++ project:

1. Select **File** > **New** > **Project**.
2. In the left pane, expand **DPC++** and select **Console Application**.
3. In the right pane, select **DPC++ Console Application**.
4. Accept or specify a project name in the **Name** field. For this example, use hello_dpcpp as the project name.
5. Accept or specify the Location for the project directory. Click **OK**.

The `hello32` (for C++) or `hello_dpcpp` (for DPC++) project assumes focus in the **Solution Explorer** view. The default Microsoft Visual Studio solution is also named `hello32` (for C++) or `hello_dpcpp` (for DPC++).

Use the Intel® oneAPI DPC++/C++ Compiler

To use the compiler with Microsoft Visual C++ (MSVC):

1. Create a MSVC project, or open an existing project.
2. In **Solution Explorer**, select the project(s) to build with Intel® oneAPI DPC++/C++ Compiler.
3. Open **Project** > **Properties**.
4. In the left pane, expand the **Configuration Properties** category and select the **General** property page.
5. In the right pane, change the Platform Toolset to **<compiler selection>**. Alternatively, you can change the toolset by selecting **Project** > **Intel Compiler** > **Use Intel oneAPI DPC++/C++ Compiler**. This sets whichever version of the compiler that you specify as the toolset for all supported platforms and configurations.

   NOTESelect **Intel(R) oneAPI DPC++ Compiler** to invoke `dpcpp-cl`. Select **Intel C++ Compiler <major version>** (example 2021) to invoke `icx` or **Intel C++ Compiler <major.minor>** (example 19.2) to invoke `icl`.

6. To add options, go to **Project** > **Properties** > **C/C++** > **Command Line** and add new options to the **Additional Options** field. Alternatively, you can select options from Intel specific properties. Refer to complete list of options in the Compiler Options section in this documentation.
7. Rebuild, using either **Build** > **Project only** > **Rebuild** for a single project, or **Build** > **Rebuild Solution** for a solution.

Switch Back to the MSVC Compiler

If your project is using the Intel® oneAPI DPC++/C++ Compiler, you can switch back to MSVC:

1. Select your project.
2. Right-click and select **Intel Compiler** > **Use Visual C++** from the context menu.

Enable an Intel® oneAPI DPC++ Compiler Runtime Environment when using the MSVC Compiler

There are two ways to enable the Intel® oneAPI DPC++ Compiler runtime environment for an MSVC project.

Enable for a Current Configuration

1. Select your project, then select **Project** > **Properties**.
2. In the left pane, select **Configuration Properties** > **Debugging**.
3. In the right pane, set **Enable Intel® oneAPI DPC++ Compiler Runtime Environment** to **Yes**.

Enable for All Configurations

1. Select your project.
2. There are two ways to enable the runtime environment:

   - From the main menu, select **Project** > **Enable Intel® oneAPI DPC++ Compiler Runtime Environment**.
   - Right-click and select **Enable DPC++ Runtime Environment** from the context menu.

Verify Use of the Intel® oneAPI DPC++/C++ Compiler

To verify the use of the Intel® oneAPI DPC++/C++ Compiler:

1. Go to **Project** > **Properties** > **C/C++** > **General**.
2. Set **Suppress Startup Banner** to **No**. Click **OK**.
3. Rebuild your application.
4. Look at the **Output** window.

You should see a message similar to the following when using the Intel® oneAPI DPC++/C++ Compiler:

```
Intel(R) oneAPI DPC++/C++ Compiler for applications running on XXXX, Version XX.X.X
```

Unsupported MSVC Project Types

The following project types are not supported:

- Class Library
- CLR Console Application
- CLR Empty Project
- Windows Forms Application
- Windows Forms Control Library

Tips for Use

- Create a separate configuration for building with Intel® oneAPI DPC++/C++ Compiler:
  - After you have created your project and specified it as an Intel project, create a new configuration (for example, `rel_intelc` based on **Release** configuration or `debug_intelc` based on **Debug** configuration).
  - Add any special optimization options offered by Intel® oneAPI DPC++/C++ Compiler only to this new configuration (for example, `rel_intelc` or `debug_intelc`) through the project property page.
- Build with Intel® oneAPI DPC++/C++ Compiler.

Select the Compiler Version

If you have multiple versions of the Intel® oneAPI DPC++/C++ Compiler installed, you can select which version you want from the **Compiler Selection** dialog box:

1. Select a project, then go to **Tools** > **Options** > **Intel Compilers and Libraries** > **<compiler>** > **Compilers**.

   NOTE The <compiler> values are C++ or DPC++.

2. Use the **Selected Compiler** drop-down menu to select the appropriate version of the compiler.
3. Click **OK**.

See Also

Use Intel® C++ dialog box

Specify a Base Platform Toolset

By default, when your project uses the Intel® oneAPI DPC++/C++ Compiler, the Base Platform Toolset property is set to use that compiler with the build environment. This environment includes paths, libraries, included files, etc., of the toolset specific to the version of Microsoft Visual Studio* you are using.

You can set the general project level property **Base Platform Toolset** to use one of the non-Intel installed platform toolsets as the base.

For example, if you are using Microsoft Visual Studio 2019, and you selected the Intel® oneAPI DPC++/C++ Compiler in the Platform Toolset property, then the Base Platform Toolset uses the Microsoft Visual Studio 2019 environment (**v142**). If you want to use other environments from Microsoft Visual Studio along with the Intel® oneAPI DPC++/C++ Compiler, set the **Base Platform Toolset** property to:

- **v141** for Microsoft Visual Studio 2017
- **v142** for Microsoft Visual Studio 2019

- **v143** for Microsoft Visual Studio 2022

> **NOTE**
> Support for Microsoft Visual Studio 2017 is deprecated as of the Intel® oneAPI 2022.1 release, and will be removed in a future release.

This property displays all installed toolsets, not including Intel toolsets.

To set the Base Platform Toolset:

- Using property pages:
  1. Select the project and open **Project** > **Properties**.
  2. In the left pane, select **Configuration Properties** > **General**.
  3. In the right pane, find **Intel Specific** > **Base Platform Toolset**.
  4. Select a value from the pop-up menu.
- Using the `msbuild.exe` command line tool, use the `/p:PlatformToolset` and `/p:BasePlatformToolset` options.
  Example: When the Platform Toolset property is already set to use the Intel® oneAPI DPC++/C++ Compiler, to build a project using the Microsoft Visual Studio 2019 environment use the following command:

```
Msbuild.exe myproject.vcxproj /p:BasePlatformToolset=v142
```

  Example: To set the Platform Toolset property to use the Intel® oneAPI DPC++/C++ Compiler and build a project using the Microsoft Visual Studio 2019 environment use the following command:

```
Msbuild.exe myproject.vcxproj /p:PlatformToolset="Intel C++ Compiler 2021" /
p:BasePlatformToolset=v142
```

  For possible values for the `/p:BasePlatformToolset` property, see the properties described above.

The next time you build your project with the Intel® oneAPI DPC++/C++ Compiler, the option you selected is used to specify the build environment.

Use Property Pages

The Intel® oneAPI DPC++/C++ Compiler includes support for Property Pages to manage both Intel-specific and general compiler options.

To set compiler options in Microsoft Visual Studio*:

1. Right-click on a project or source file in the **Solution Explorer** view.
2. Select **Properties** from the pop-up menu.
3. In the **Property Pages** dialog box, expand the **C/C++** (for C++), or **DPC++** (for DPC++) section to view the categories of compiler options.
4. Click **OK** to complete your selection.

The option you selected is used in the compilation the next time you build your project.

Use Intel® Libraries with Microsoft Visual Studio*

You can use the compiler with the following Intel® Libraries, which may be included as a part of the product:

- Intel® oneAPI Data Analytics Library (oneDAL)
- Intel® Integrated Performance Primitives (Intel® IPP)
- Intel® oneAPI Threading Building Blocks (oneTBB)
- Intel® oneAPI Math Kernel Library (oneMKL)

Use the property pages to specify Intel Libraries to use with the selected project configuration. The functionality supports Intel® C++, Intel® oneAPI DPC++, and Microsoft Visual C++* project types.

To specify Intel Libraries, select **Project** > **Properties**. In **Configuration Properties**, select **Intel Libraries for oneAPI**, then do the following:

1. To use **oneDAL** change the **Use oneDAL** settings as follows:
   - **No**: Disable Use of oneDAL.
   - **Default Linking Method**: Use parallel dynamic oneDAL libraries.
   - **Multi-threaded Static Library**: Use parallel static oneDAL libraries.
   - **Single-threaded Static Library**: Use sequential static oneDAL libraries.
   - **Multi-threaded DLL**: Use parallel dynamic oneDAL libraries.
   - **Single-threaded DLL**: Use sequential dynamic oneDAL libraries.
2. To use **Intel® Integrated Performance Primitives**, change the **Use Intel® IPP** settings as follows:
   - **No**: Disable use of Intel® IPP libraries.
   - **Default Linking Method**: Use dynamic Intel® IPP libraries.
   - **Static Library**: Use static Intel® IPP libraries.
   - **Dynamic Library**: Use dynamic Intel® IPP libraries.
3. To use **oneTBB** in your project, change the **Use oneTBB** settings as follows:
   - **No**: Disable use of oneTBB libraries.
   - **Use oneTBB**: Set to **Yes** to use oneTBB in the application.
   - **Instrument for use with Analysis Tools**: Set to **Yes** to analyze your release mode application (not required for debug mode).
4. To use **oneMKL** in your project, change the **Use oneMKL** property settings as follows:
   - **No**: Disable use of oneMKL libraries.
   - **Parallel**: Use parallel oneMKL libraries.
   - **Sequential**: Use sequential oneMKL libraries.
   - **Cluster**: Use cluster libraries.

   The target platform of an Intel® oneAPI DCP++ project is set to **x64**, so a final selection appears: **Use interface**. If selected, the corresponding ilp oneMKL libraries are added to the linker command line. Additionally, the MKL_ILP64 preprocessor definition is added to the compiler command line. If you do not make this selection, the lp oneMKL libraries are used.

Additional settings for use with the Microsoft Visual C++* Platform Toolset are available on the **Intel Libraries for oneAPI** category, found at **Tools** > **Options**.

Change the Selected Intel Libraries for oneAPI

If you have installed multiple versions of the Intel Libraries for oneAPI, you can specify which version to use with the Microsoft Visual C++* compiler. To do this:

1. Select **Tools** > **Options**.
2. In the left pane, select **Intel Compilers and Libraries** > **Intel Libraries for oneAPI**.
3. Select the desired library version from the drop-down boxes in the right pane.

For more information, see the Intel® oneAPI Data Analytics Library (oneDAL), Intel® Integrated Performance Primitives (Intel® IPP), Intel® oneAPI Threading Building Blocks (oneTBB), and Intel® oneAPI Math Kernel Library (oneMKL) documentation.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.   Notice revision #20201201

Include MPI Support

To specify the type of message-passing interface (MPI) support you want:

1. Open the project's property pages and select **Configuration Properties** > **Intel Libraries for oneAPI**.
2. Set the property **Use oneMKL** to **Cluster**.
3. Set the property **Use MPI Library** to one of the following values:
   - **Intel® MPI Library**
   - **MS-MPI**
4. Build the project.

The next time you build your project with the Intel® oneAPI DPC++/C++ Compiler or Microsoft Visual C++ compiler, it will include support for the version of MPI that you specified.

Dialog Box Help

This section provides information about access to dialog boxes and information about compilers, libraries, and converter dialog boxes.

Options: Compilers dialog box

To access the **Compilers** page:

1. Open **Tools** > **Options**.
2. In the left pane, select **Intel Compilers and Libraries** > **C++** > **Compilers** for `icx` or **Intel Compilers and Libraries** > **DPC++** > **Compilers** for `dpcpp-cl`.

Compiler Selection for C++

Target Platform: Select your target platform.

Platform Toolset/Selected Compiler: Select your compiler for your platform toolset. The left column lists the platform toolset names. The right column lists combo boxes, which are used to select a compiler. The default value for all combo boxes in current table is **<Latest>**.

> **NOTE** The left column contains Intel® C++ Compiler Classic and Intel® oneAPI DPC++/C++ Compiler options. The **Intel C++ Compiler <major.minor>** (example 19.2) selects the Intel® C++ Compiler Classic (`icc`). The **Intel C++ Compiler <major>** (example 2021) selects the Intel® oneAPI DPC++/C++ Compiler (`icx`).

Default Options: Sets the default options for a selected compiler. You may specify this setting for each selected compiler.

Environment: Sets custom environment variables. You may specify this setting for each selected compiler.

> **NOTE** The Environment selection is only available for `icx`.

Compiler Information: Shows the detail description of the selected compiler.

Reset All: Sets all contents back to the default value on the dialog.

Compiler Selection for Intel® oneAPI DPC++ Compiler

Platform Toolset/Selected Compiler: Select your compiler for your platform toolset. The left column lists the platform toolset names. The right column lists combo boxes, which are used to select a compiler. The default value for all combo boxes in current table is **<Latest>**.

Default Options: Sets the default options for a selected compiler. You may specify this setting for each selected compiler.

Environment: Sets custom environment variables. You may specify this setting for each selected compiler.

NOTE The Environment selection is only available for `icx`.

Compiler Information: Shows the detail description of the selected compiler.

Reset All: Sets all contents back to the default value on the dialog.

See Also

Use Intel® oneAPI DPC++/C++ Compiler dialog box

To access the **Use Intel oneAPI DPC++/C++ Compiler** dialog box:

1. Select one or more files in the **Solution Explorer**.
2. Right-click and select **Intel Compiler** > **Use Intel oneAPI DPC++/C++ Compiler for selected file(s)...**

Use this dialog box to change the compiler for one or more selected files to the Intel® oneAPI DPC++/C++ Compiler.

To use the **Select the configuration(s) to update**:

1. Select your desired configuration.
2. Choose from **Active configuration** or **All configurations**.

   If you select **All configurations**, the compiler is changed in all configurations for the currently selected file(s).

To use the **Select the Platform Toolset:**

1. Select your desired toolset.

   This only applies if you have multiple platform toolsets installed.

See Also

Use the Intel® oneAPI DPC++/C++ Compiler

Options: Intel Libraries for oneAPI dialog box

Use the **Intel Libraries for oneAPI** dialog box to specify standalone library versions to use with the Microsoft Visual C++* compiler.

To access the **Intel Libraries for oneAPI** dialog box:

1. Open **Tools** > **Options**.
2. Select **Intel Compilers and Libraries** > **Intel Libraries for oneAPI**.

In the dialog box, you can select your desired library version from the drop-down box with one of the following values:

- **oneDAL**
- **Intel IPP**
- **oneTBB**

- **oneMKL**
- **Reset All**: Use the latest libraries (default)

> **NOTE** To enable or disable the Intel Libraries for oneAPI, use the property pages located in the **Configuration Properties** category.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

See Also

Use Intel® Libraries for oneAPI

Options: Converter dialog box

To access the **Converter** page:

1. Click **Tools** > **Options**.
2. select **Intel Compilers and Libraries** > **C++** > **Converter**.

Use the **Converter** page to specify which platform toolset to use when upgrading an Intel® C++ solution (`.icproj`) from an older version of Microsoft Visual Studio to a C++ project supported by Microsoft Visual Studio 2017 or later (`.vcxproj`). Once a solution is upgraded, the `.icproj` file is not used and can be deleted.

> **NOTE** Support for Microsoft Visual Studio 2017 is deprecated as of the Intel® oneAPI 2022.1 release, and will be removed in a future release.

Win32: Select the desired compiler version to be used when converting projects based on IA-32 architecture.

X64: Select the desired compiler version to be used when converting projects based on x64 architecture.

Reset All: Click this button to use the default platform toolsets.

Part III

Compiler Reference

This section contains compiler reference information. For example, it contains information about compiler options, compiler limits, and libraries.

C/C++/SYCL Calling Conventions

There are a number of calling conventions that set the rules on how arguments are passed to a function and how the values are returned from the function.

NOTE IA-32 applications do not apply for SYCL.

Calling Conventions on Linux

The following table summarizes the supported calling conventions on Linux:

Calling Convention	Compiler Option	Description
`__attribute((cdecl))`	None	Default calling convention for C/C++/SYCL programs. Can be specified on a function with variable arguments.
`__attribute((stdcall))`	None	Calling convention that specifies the arguments are passed on the stack. Cannot be specified on a function with variable arguments.
`__attribute((regparm (number))`	None	On systems based on IA-32 architecture, the `regparm` attribute causes the compiler to pass up to *number* arguments in registers `EAX`, `EDX`, and `ECX` instead of on the stack. Functions that take a variable number of arguments will continue to pass all of their arguments on the stack.

Calling Convention	Compiler Option	Description
`__attribute__((regcall))`	`-regcall` specifies that `__regcall` is the default calling convention for functions in the compilation, unless another calling convention is specified on a declaration.	Intel oneAPI DPC++/C++ Compiler calling convention that specifies that as many arguments as possible are passed in registers; likewise, `__regcall` uses registers whenever possible to return values. This calling convention is ignored if specified on a function with variable arguments.
`__attribute__((vectorcall))`	None	Calling convention that specifies that a function passing vector type arguments should use vector registers.

Calling Conventions on Windows

The following table summarizes the supported calling conventions on Windows:

Calling Convention	Compiler Option	Description
`__cdecl`	`/Gd`	This is the default calling convention for C/C++/SYCL programs. It can be specified on a function with variable arguments.
`__stdcall`	`/Gz`	Standard calling convention used for Win32 API functions.
`__fastcall`	`/Gr`	Fast calling convention that specifies that arguments are passed in registers rather than on the stack.
`__regcall`	`/Qregcall` specifies that `__regcall` is the default calling convention for functions in the compilation, unless another calling convention is specified on a declaration.	Intel® oneAPI DPC++/C++ Compiler calling convention that specifies that as many arguments as possible are passed in registers; likewise, `__regcall` uses registers whenever possible to return values. This calling convention is ignored if specified on a function with variable arguments. For more information about the Intel-compatible vector functions ABI, download the Vector Function Application Binary Interface PDF.
`__thiscall`	None	Default calling convention used by C++ member functions that do not use variable arguments.
`__vectorcall`	`/Gv`	Calling convention that specifies that a function passing vector type arguments should use vector registers.

The __regcall Calling Convention

The `__regcall` calling convention is unique to the Intel oneAPI DPC++/C++ Compiler and requires some additional explanation.

To use `__regcall`, place the keyword before a function declaration. For example:

Linux

```
__attribute__((regcall)) foo (int I, int j);
```

Windows

```
__regcall int foo (int i, int j);
```

Available __regcall Registers

All registers in a `__regcall` function can be used for parameter passing/returning a value, except those that are reserved by the compiler. The following table lists the registers that are available in each register class depending on the default ABI for the compilation. The registers are used in the order shown below.

Register Class/ Architecture	IA-32 for Linux	IA-32 for Windows	Intel® 64 for Linux	Intel® 64 for Windows
`GPR`	`EAX, ECX, EDX, EDI, ESI`	`ECX, EDX, EDI, ESI`	`RAX, RCX, RDX, RDI, RSI, R8, R9, R10, R11, R12, R14, R15`	`RAX, RCX, RDX, RDI, RSI, R8, R9, R11, R12, R14, R15`
`FP`	`ST0`	`ST0`	`ST0`	`ST0`
`MMX`	None	None	None	None
`XMM`	`XMM0 - XMM7`	`XMM0 - XMM7`	`XMM0 - XMM15`	`XMM0 - XMM15`
`YMM`	`YMM0 - YMM7`	`YMM0 - YMM7`	`YMM0 - YMM15`	`YMM0 - YMM15`
`ZMM`	`ZMM0 - ZMM7`	`ZMM0 - ZMM7`	`ZMM0 - YMM15`	`ZMM0 - YMM15`

__regcall Data Type Classification

Parameters and return values for *__regcall* are classified by data type and are passed in the registers of the classes shown in the following table.

NOTE
All types assigned to `XMM`, `YMM`, or `ZMM` in a non-SSE target are passed in the stack.

Type (Signed and Unsigned)	IA-32	Intel® 64
`bool, char, int, enum, _Decimal32, long, pointer`	`GPR`	`GPR`
`short, __mmask{8,16,32,64}`	`GPR`	`GPR`
`long long, __int64`	See Structured Data Type Classification Rules	`GPR`
`_Decimal64`	`XMM`	`GPR`

Type (Signed and Unsigned)	IA-32	Intel® 64
long double	FP	FP
float, double, float128, _Decimal128	XMM	XMM
__m128, __m128i, __m128d	XMM	XMM
__m256, __m256i, __m256d	YMM	YMM
__m512, __m512i, __m512d	ZMM	ZMM
complex type, struct, union	See Structured Data Type Classification Rules	See Structured Data Type Classification Rules

NOTE
For the purpose of structured types, the classification of GPR class is used.

On systems based on IA-32 architecture, these 64-bit integer types (long long, __int64) get classified to the GPR class and are passed in two registers, as if they were implemented as a structure of two 32-bit integer fields.

Types that are smaller in size than registers of their associated class are passed in the lower part of those registers; for example, float is passed in the lower four bytes of an XMM register.

__regcall Structured Data Type Classification Rules

Structures/unions and complex types are classified similarly to what is described in the x86_64 ABI, with the following exceptions:

- There is no limitation on the overall size of a structure.
- The register classes for basic types are given in Data Type Classifications.
- For systems based on the IA-32 architecture, classification is performed on four-bytes. For systems based on other architectures, classification is performed on eight-bytes.

__regcall Placement in Registers or on the Stack

After the classification described in Data Type Classifications and Structured Data Type Classification Rules, *__regcall* parameters and return values are either put into registers specified in Available Registers or placed in memory, according to the following:

- Each chunk (eight bytes on systems based on Intel® 64 architecture or four-bytes on systems based on IA-32 architecture of a value of Data Type is assigned a register class. If enough registers from Available Registers are available, the whole value is passed in registers, otherwise the value is passed using the stack.
- If the classification were to use one or more register classes, then the registers of these classes from the table in Available Registers are used, in the order given there.
- If no more registers are available in one of the required register classes, then the whole value is put on the stack.

__regcall Registers That Preserve Their Values

The following registers preserve their values across a __regcall call, as long as they were not used for passing a parameter or returning a value:

Register Class/ABI	IA-32	Intel® 64 for Linux	Intel® 64 for Windows
GPR	ESI, EDI, EBX, EBP, ESP	R12 - R15, RBX, RBP, RSP	R12 - R15, RBX, RBP, RSP
FP	None	None	None
MMX	None	None	None
XMM	XMM4 - XMM7	XMM8 - XMM15	XMM8 - XMM15
YMM	XMM4 - XMM7	XMM8 - XMM15	XMM8 - XMM15
ZMM	XMM4 - XMM7	XMM8 - XMM15	XMM8 - XMM15

All other registers do not preserve their values across this call.

See Also

Structured Data Type Classification Rules
Data Type Classifications
Available Registers

Compiler Options

This compiler supports many compiler options you can use in your applications.

In this section, we provide the following:

- An alphabetical list of compiler options that includes their short descriptions
- A list of deprecated options for SYCL and lists of deprecated and removed options for C++
- General rules for compiler options and the conventions we use when referring to options
- Details about what appears in the compiler option descriptions
- A description of each compiler option. The descriptions appear under the option's functional category. Within each category, the options are listed in alphabetical order.

Clang compiler options are supported for this compiler. We do not document these options, but you can check -help on the command line to see if a particular option is supported. For more information about Clang options, see the Clang documentation.

> **NOTE**
> On Windows, two compilers are available: `dpcpp` and `dpcpp-cl`.
>
> If you want to use Linux-style option syntax, where options start with dash (-), you should continue to use the `dpcpp` compiler.
>
> If you want to use Microsoft Visual C++ (MSVC)-compatible option syntax, where options start with /, you should use the `dpcpp-cl` compiler.

> **NOTE**macOS* is not supported for the `icx/icpx`, `dpcpp`, or `dpcpp-cl` compilers. For macOS or Xcode* support, visit the `icc` compiler: Intel® C++ Compiler Classic Developer Guide and Reference.

For details about new functionality, such as new compiler options, see the Release Notes for the product.

Alphabetical Option List

The following table lists all the current compiler options in alphabetical order.

align	Determines whether variables and arrays are naturally aligned.
ansi	Enables language compatibility with the gcc option ansi.
arch	Tells the compiler which features it may target, including which instruction sets it may generate.
ax, Qax	Tells the compiler to generate multiple, feature-specific auto-dispatch code paths for Intel® processors if there is a performance benefit.
B	Specifies a directory that can be used to find include files, libraries, and executables.
Bdynamic	Enables dynamic linking of libraries at run time.
Bstatic	Enables static linking of a user's library.
Bsymbolic	Binds references to all global symbols in a program to the definitions within a user's shared library.
Bsymbolic-functions	Binds references to all global function symbols in a program to the definitions within a user's shared library.
C	Places comments in preprocessed source output.
c	Prevents linking.
D	Defines a macro name that can be associated with an optional value.
dD, QdD	Same as option -dM, but outputs #define directives in preprocessed source.
debug (Linux*)	Enables or disables generation of debugging information.
debug (Windows*)	Enables or disables generation of debugging information.
device-math-lib	Enables or disables certain device libraries. This is a deprecated option that may be removed in a future release.
dM, QdM	Tells the compiler to output macro definitions in effect after preprocessing.
dryrun	Specifies that driver tool commands should be shown but not executed.
dumpmachine	Displays the target machine and operating system configuration.
dumpversion	Displays the version number of the compiler.
dynamic-linker	Specifies a dynamic linker other than the default.
E	Causes the preprocessor to send output to stdout.
EH	Specifies the model of exception handling to be performed.
EP	Causes the preprocessor to send output to stdout, omitting #line directives.
F (Windows*)	Specifies the stack reserve amount for the program.
Fa	Specifies that an assembly listing file should be generated.

fasm-blocks	Enables the use of blocks and entire functions of assembly code within a C or C++ file.
fast	Maximizes speed across the entire program.
fasynchronous-unwind-tables	Determines whether unwind information is precise at an instruction boundary or at a call boundary.
fbuiltin, Oi	Enables or disables inline expansion of intrinsic functions.
fcommon	Determines whether the compiler treats common symbols as global definitions.
fdata-sections, Gw	Places each data item in its own COMDAT section.
FD	Generates file dependencies related to the Microsoft* C/C++ compiler.
Fe	Specifies the name for a built program or dynamic-link library.
fexceptions	Enables exception handling table generation.
ffp-contract	Controls when the compiler is permitted to form fused floating-point operations, such as fused multiply-add (FMA). Fused operations are allowed to produce more precise results than performing the individual operations separately.
ffreestanding , Qfreestanding	Ensures that compilation takes place in a freestanding environment.
ffunction-sections, Gy	Places each function in its own COMDAT section.
fgnu89-inline	Tells the compiler to use C89 semantics for inline functions when in C99 mode.
fimf-absolute-error, Qimf-absolute-error	Defines the maximum allowable absolute error for math library function results.
fimf-accuracy-bits, Qimf-accuracy-bits	Defines the relative error for math library function results, including division and square root.
fimf-arch-consistency, Qimf-arch-consistency	Ensures that the math library functions produce consistent results across different microarchitectural implementations of the same architecture.
fimf-domain-exclusion, Qimf-domain-exclusion	Indicates the input arguments domain on which math functions must provide correct results.
fimf-force-dynamic-target, Qimf-force-dynamic-target	Instructs the compiler to use run-time dispatch in calls to math functions.
fimf-max-error, Qimf-max-error	Defines the maximum allowable relative error for math library function results, including division and square root.
fimf-precision, Qimf-precision	Lets you specify a level of accuracy (precision) that the compiler should use when determining which math library functions to use.
fimf-use-svml, Qimf-use-svml	Instructs the compiler to use the Short Vector Math Library (SVML) rather than the Intel® oneAPI DPC++/C++ Compiler Math Library (LIBM) to implement math library functions.
finline	Tells the compiler to inline functions declared with __inline and perform C ++ inlining .
finline-functions	Enables function inlining for single file compilation.
fintelfpga	Lets you perform ahead-of-time (AOT) compilation for the Field Programmable Gate Array (FPGA).

fiopenmp, Qiopenmp	Enables recognition of OpenMP* features, such as parallel, simd, and offloading directives. This is an alternate option for compiler option [Q or q]openmp.
FI	Tells the preprocessor to include a specified file name as the header file.
fixed	Causes the linker to create a program that can be loaded only at its preferred base address.
fjump-tables	Determines whether jump tables are generated for switch statements.
fkeep-static-consts , Qkeep-static-consts	Tells the compiler to preserve allocation of variables that are not referenced in the source.
Fm	Tells the linker to generate a link map file. This is a deprecated option that may be removed in a future release.
fma, Qfma	Determines whether the compiler generates fused multiply-add (FMA) instructions if such instructions exist on the target processor.
fmath-errno	Tells the compiler that errno can be reliably tested after calls to standard math library functions.
fno-gnu-keywords	Tells the compiler to not recognize typeof as a keyword.
fno-operator-names	Disables support for the operator names specified in the standard.
fno-rtti	Disables support for run-time type information (RTTI).
fno-sycl-libspirv	Disables the check for libspirv (the SPIR-V* tools library).
Fo	Specifies the name for an object file.
foffload-static-lib	Tells the compiler to link with a fat (multi-architecture) static library. This is a deprecated option that may be removed in a future release.
fomit-frame-pointer , Oy	Determines whether EBP is used as a general-purpose register in optimizations.
fopenmp	Option -fopenmp is a deprecated option that will be removed in a future release.
fopenmp-declare-target-scalar-defaultmap, Qopenmp-declare-target-scalar-defaultmap	Determines which implicit data-mapping/sharing rules are applied for a scalar variable referenced in a target pragma.
fopenmp-device-lib	Enables or disables certain device libraries for an OpenMP* target.
fopenmp-target-buffers, Qopenmp-target-buffers	Enables a way to overcome the problem where some OpenMP* offload SPIR-V* devices produce incorrect code when a target object is larger than 4GB.
fopenmp-targets, Qopenmp-targets	Enables offloading to a specified GPU target if OpenMP* features have been enabled.
foptimize-sibling-calls	Determines whether the compiler optimizes tail recursive calls.
Fp	Lets you specify an alternate path or file name for precompiled header files.
fpack-struct	Specifies that structure members should be packed together.
fpascal-strings	Tells the compiler to allow for Pascal-style string literals.

fpermissive	Tells the compiler to allow for non-conformant code.
fpic	Determines whether the compiler generates position-independent code.
fpie	Tells the compiler to generate position-independent code. The generated code can only be linked into executables.
fp-model, fp	Controls the semantics of floating-point calculations.
fp-speculation, Qfp-speculation	Tells the compiler the mode in which to speculate on floating-point operations.
freg-struct-return	Tells the compiler to return struct and union values in registers when possible.
fshort-enums	Tells the compiler to allocate as many bytes as needed for enumerated types.
fstack-protector	Enables or disables stack overflow security checks for certain (or all) routines.
fstack-security-check	Determines whether the compiler generates code that detects some buffer overruns.
fsycl	Enables a program to be compiled as a SYCL* program rather than as plain C++11 program.
fsycl-add-targets	Lets you add arbitrary device binary images to the fat SYCL* binary when linking. This is a deprecated option that may be removed in a future release.
fsycl-dead-args-optimization	Enables elimination of SYCL dead kernel arguments.
fsycl-device-code-split	Specifies a SYCL* device code module assembly.
fsycl-device-lib	Enables or disables certain device libraries for a SYCL* target.
fsycl-device-only	Tells the compiler to generate a device-only binary.
fsycl-early-optimizations	Enables LLVM-related optimizations before SPIR-V* generation.
fsycl-enable-function-pointers	Enables function pointers and support for virtual functions for SYCL kernels and device functions. This is an experimental feature.
fsycl-esimd-force-stateless-mem	Determines whether the compiler enforces stateless memory accesses within ESIMD kernels on the target device. This is an experimental feature.
fsycl-explicit-simd	Enables or disables the experimental "Explicit SIMD" SYCL* extension. This is a deprecated option that may be removed in a future release.
fsycl-help	Causes help information to be emitted from the device compiler backend.
fsycl-host-compiler	Tells the compiler to use the specified compiler for the host compilation of the overall offloading compilation that is performed.
fsycl-host-compiler-options	Passes options to the compiler specified by option fsycl-host-compiler.
fsycl-id-queries-fit-in-int	Tells the compiler to assume that SYCL ID queries fit within MAX_INT.
fsycl-instrument-device-code	Enables or disables linking of the Instrumentation and Tracing Technology (ITT) device libraries for VTune™.
fsycl-link	Tells the compiler to perform a partial link of device binaries to be used with Field Programmable Gate Array (FPGA).

fsycl-link-targets	Tells the compiler to link only device code. This is a deprecated option that may be removed in a future release.
fsycl-max-parallel-link-jobs	Tells the compiler that it can simultaneously spawn up to the specified number of processes to perform actions required to link SYCL applications. This is an experimental feature.
fsycl-targets	Tells the compiler to generate code for specified device targets.
fsycl-unnamed-lambda	Enables unnamed SYCL* lambda kernels.
fsycl-use-bitcode	Tells the compiler to produce device code in LLVM Intermediate Representation (IR) bitcode format into fat objects.
fsyntax-only	Tells the compiler to check only for correct syntax.
ftrapuv , Qtrapuv	Initializes stack local variables to an unusual value to aid error detection.
funsigned-char	Change default char type to unsigned.
fuse-ld	Tells the compiler to use a different linker instead of the default linker (ld).
fvec-peel-loops, Qvec-peel-loops	Enables peel loop vectorization.
fvec-remainder-loops, Qvec-remainder-loops	Enables remainder loop vectorization.
fvec-with-mask, Qvec-with-mask	Enables vectorization for short trip-count loops with masking.
fverbose-asm	Produces an assembly listing with compiler comments, including options and version information.
fvisibility	Specifies the default visibility for global symbols or the visibility for symbols in declarations, functions, or variables.
fzero-initialized-in-bss, Qzero-initialized-in-bss	Determines whether the compiler places in the DATA section any variables explicitly initialized with zeros.
g	Tells the compiler to generate a level of debugging information in the object file.
GA	Enables faster access to certain thread-local storage (TLS) variables.
gcc-toolchain	Lets you specify the location of the base toolchain.
Gd	Makes __cdecl the default calling convention.
gdwarf	Lets you specify a DWARF Version format when generating debug information.
GF	Enables read-only string-pooling optimization.
GR	Enables or disables C++ Run Time Type Information (RTTI).
Gr	Makes __fastcall the default calling convention.
grecord-gcc-switches	Causes the command line options that were used to invoke the compiler to be appended to the DW_AT_producer attribute in DWARF debugging information.
GS	Determines whether the compiler generates code that detects some buffer overruns.

Gs	Lets you control the threshold at which the stack checking routine is called or not called.
gsplit-dwarf	Creates a separate object file containing DWARF debug information.
guard	Enables the control flow protection mechanism.
Gv	Tells the compiler to use the vector calling convention (__vectorcall) when passing vector type arguments.
Gz	Makes __stdcall the default calling convention.
H, QH	Tells the compiler to display the include file order and continue compilation.
help	Displays a list of supported compiler options in alphabetical order.
I	Specifies an additional directory to search for include files.
I-	Splits the include path.
idirafter	Adds a directory to the second include file search path.
imacros	Allows a header to be specified that is included in front of the other headers in the translation unit.
ipo, Qipo	Enables interprocedural optimization between files.
ipp-link, Qipp-link	Controls whether the compiler links to static or dynamic threaded Intel® Integrated Performance Primitives (Intel® IPP) run-time libraries.
iprefix	Lets you indicate the prefix for referencing directories that contain header files.
iquote	Adds a directory to the front of the include file search path for files included with quotes but not brackets.
isystem	Specifies a directory to add to the start of the system include path.
iwithprefix	Appends a directory to the prefix passed in by -iprefix and puts it on the include search path at the end of the include directories.
iwithprefixbefore	Similar to -iwithprefix except the include directory is placed in the same place as -I command-line include directories.
J	Sets the default character type to unsigned.
Kc++, TP	Tells the compiler to process all source or unrecognized file types as C++ source files. This is a deprecated option that may be removed in a future release.
l	Tells the linker to search for a specified library when linking.
L	Tells the linker to search for libraries in a specified directory before searching the standard directories.
LD	Specifies that a program should be linked as a dynamic-link (DLL) library.
link	Passes user-specified options directly to the linker at compile time.
m	Tells the compiler which features it may target, including which instruction set architecture (ISA) it may generate.
M, QM	Tells the compiler to generate makefile dependency lines for each source file.

m32, m64 , Qm32, Qm64	Tells the compiler to generate code for a specific architecture. Option m32 (and Qm32) is deprecated and will be removed in a future release.
m80387	Specifies whether the compiler can use x87 instructions.
malign-double	Determines whether double, long double, and long long types are naturally aligned. This option is equivalent to specifying option align.
march	Tells the compiler to generate code for processors that support certain features.
masm	Tells the compiler to generate the assembler output file using a selected dialect.
mbranches-within-32B-boundaries, Qbranches-within-32B-boundaries	Tells the compiler to align branches and fused branches on 32-byte boundaries for better performance.
mcmodel	Tells the compiler to use a specific memory model to generate code and store data.
MD	Tells the linker to search for unresolved references in a multithreaded, dynamic-link run-time library.
MD, QMD	Preprocess and compile, generating output file containing dependency information ending with extension .d.
MF, QMF	Tells the compiler to generate makefile dependency information in a file.
MG, QMG	Tells the compiler to generate makefile dependency lines for each source file.
mintrinsic-promote, Qintrinsic-promote	Enables functions containing calls to intrinsics that require a specific CPU feature to have their target architecture automatically promoted to allow the required feature.
MM, QMM	Tells the compiler to generate makefile dependency lines for each source file.
MMD, QMMD	Tells the compiler to generate an output file containing dependency information.
momit-leaf-frame-pointer	Determines whether the frame pointer is omitted or kept in leaf functions.
MQ	Changes the default target rule for dependency generation.
mregparm	Lets you control the number registers used to pass integer arguments.
MT	Tells the linker to search for unresolved references in a multithreaded, static run-time library.
MT, QMT	Changes the default target rule for dependency generation.
mtune, tune	Performs optimizations for specific processors but does not cause extended instruction sets to be used (unlike -march).
nodefaultlibs	Prevents the compiler from using standard libraries when linking.
no-intel-lib, Qno-intel-lib	Disables linking to specified Intel® libraries, or to all Intel® libraries.
no-libgcc	Prevents the linking of certain gcc-specific libraries.
nolib-inline	Disables inline expansion of standard library or intrinsic functions.
nolibsycl	Disables linking of the SYCL* runtime library.

nologo	Tells the compiler to not display compiler version information.
nostartfiles	Prevents the compiler from using standard startup files when linking.
nostdinc++	Do not search for header files in the standard directories for C++, but search the other standard directories.
nostdlib	Prevents the compiler from using standard libraries and startup files when linking.
O	Specifies the code optimization for applications.
o	Specifies the name for an output file.
Od	Disables all optimizations.
Ofast	Sets certain aggressive options to improve the speed of your application.
Os	Enables optimizations that do not increase code size; it produces smaller code size than O2.
Ot	Enables all speed optimizations.
Ox	Enables maximum optimizations.
P	Tells the compiler to stop the compilation process and write the results to a file.
pc, Qpc	Enables control of floating-point significand precision.
pie	Determines whether the compiler generates position-independent code that will be linked into an executable.
pragma-optimization-level	Specifies which interpretation of the optimization_level pragma should be used if no prefix is specified.
pthread	Tells the compiler to use pthreads library for multithreading support.
qactypes, Qactypes	Tells the compiler to include the Algorithmic C (AC) data type folder for header searches and link to the AC data types libraries for Field Programmable Gate Array (FPGA) and CPU compilations.
qdaal, Qdaal	Tells the compiler to link to certain libraries in the Intel® oneAPI Data Analytics Library (oneDAL).
qipp, Qipp	Tells the compiler to link to some or all of the Intel® Integrated Performance Primitives (Intel® IPP) libraries.
Qlong-double	Changes the default size of the long double data type.
qmkl, Qmkl	Tells the compiler to link to certain libraries in the Intel® oneAPI Math Kernel Library (oneMKL) . On Windows systems, you must specify this option at compile time.
qopenmp, Qopenmp	Enables recognition of OpenMP* features and tells the parallelizer to generate multi-threaded code based on OpenMP* directives.
qopenmp-lib, Qopenmp-lib	Lets you specify an OpenMP* run-time library to use for linking.
qopenmp-link	Controls whether the compiler links to static or dynamic OpenMP* run-time libraries.
qopenmp-simd, Qopenmp-simd	Enables or disables OpenMP* SIMD compilation.

qopenmp-stubs, Qopenmp-stubs	Enables compilation of OpenMP* programs in sequential mode.
Qoption	Passes options to a specified tool.
qopt-assume-no-loop-carried-dep, Qopt-assume-no-loop-carried-dep	Lets you set a level of performance tuning for loops.
qopt-dynamic-align, Qopt-dynamic-align	Enables or disables dynamic data alignment optimizations.
qopt-for-throughput, Qopt-for-throughput	Determines how the compiler optimizes for throughput depending on whether the program is to run in single-job or multi-job mode.
qopt-multiple-gather-scatter-by-shuffles, Qopt-multiple-gather-scatter-by-shuffles	Enables or disables the optimization for multiple adjacent gather/scatter type vector memory references.
qopt-report, Qopt-report	Enables the generation of a YAML file that includes optimization transformation information.
qopt-report-file, Qopt-report-file	Specifies whether the output for the generated optimization report goes to a file, stderr, or stdout.
qopt-streaming-stores, Qopt-streaming-stores	Enables generation of streaming stores for optimization.
qtbb, Qtbb	Tells the compiler to link to the Intel® oneAPI Threading Building Blocks (oneTBB) libraries.
regcall, Qregcall	Tells the compiler that the __regcall calling convention should be used for functions that do not directly specify a calling convention.
reuse-exe	Tells the compiler to speed up Field Programmable Gate Array (FPGA) target compile time by reusing a previously compiled FPGA hardware image.
RTC	Enables checking for certain run-time conditions.
S	Causes the compiler to compile to an assembly file only and not link.
save-temps , Qsave-temps	Tells the compiler to save intermediate files created during compilation.
shared	Tells the compiler to produce a dynamic shared object instead of an executable.
shared-intel	Causes Intel-provided libraries to be linked in dynamically.
shared-libgcc	Links the GNU libgcc library dynamically.
showIncludes	Tells the compiler to display a list of the include files.
sox, Qsox	Tells the compiler to save the compilation options in the executable file.
static	Prevents linking with shared libraries.
static-intel	Causes Intel-provided libraries to be linked in statically.
static-libgcc	Links the GNU libgcc library statically.
static-libstdc++	Links the GNU libstdc++ library statically.
std, Qstd	Tells the compiler to conform to a specific language standard.
strict-ansi	Tells the compiler to implement strict ANSI conformance dialect.

sysroot	Specifies the root directory where headers and libraries are located.
T	Tells the linker to read link commands from a file.
Tc	Tells the compiler to process a file as a C source file.
TC	Tells the compiler to process all source or unrecognized file types as C source files.
Tp	Tells the compiler to process a file as a C++ source file.
U	Undefines any definition currently in effect for the specified macro .
u (Linux*)	Tells the compiler the specified symbol is undefined.
undef	Disables all predefined macros .
unroll , Qunroll	Tells the compiler the maximum number of times to unroll loops.
use-intel-optimized-headers, Quse-intel-optimized-headers	Determines whether the performance headers directory is added to the include path search list.
use-msasm	Enables the use of blocks and entire functions of assembly code within a C or C++ file.
v	Specifies that driver tool commands should be displayed and executed.
vd	Enables or suppresses hidden vtordisp members in C++ objects.
vec, Qvec	Enables or disables vectorization.
vec-threshold, Qvec-threshold	Sets a threshold for the vectorization of loops.
version	Tells the compiler to display GCC-style version information.
vmg	Selects the general representation that the compiler uses for pointers to members.
vmv	Enables pointers to members of any inheritance type.
w	Disables all warning messages.
w, W	Specifies the level of diagnostic messages to be generated by the compiler.
Wa	Passes options to the assembler for processing.
Wabi	Determines whether a warning is issued if generated code is not C++ ABI compliant.
Wall	Enables warning and error diagnostics.
watch	Tells the compiler to display certain information to the console output window.
Wcheck-unicode-security	Determines whether the compiler performs source code checking for Unicode vulnerabilities.
Wcomment	Determines whether a warning is issued when /* appears in the middle of a /* */ comment.
Wdeprecated	Determines whether warnings are issued for deprecated C++ headers.
Weffc++, Qeffc++	Enables warnings based on certain C++ programming guidelines.
Werror, WX	Changes all warnings to errors.

Werror-all	Causes all warnings and currently enabled remarks to be reported as errors.
Wextra-tokens	Determines whether warnings are issued about extra tokens at the end of preprocessor directives.
Wformat	Determines whether argument checking is enabled for calls to printf, scanf, and so forth.
Wformat-security	Determines whether the compiler issues a warning when the use of format functions may cause security problems.
Wl	Passes options to the linker for processing.
Wmain	Determines whether a warning is issued if the return type of main is not expected.
Wmissing-declarations	Determines whether warnings are issued for global functions and variables without prior declaration.
Wmissing-prototypes	Determines whether warnings are issued for missing prototypes.
Wno-sycl-strict	Disables warnings that enforce strict SYCL* language compatibility.
Wp	Passes options to the preprocessor.
Wpointer-arith	Determines whether warnings are issued for questionable pointer arithmetic.
Wreorder	Tells the compiler to issue a warning when the order of member initializers does not match the order in which they must be executed.
Wreturn-type	Determines whether warnings are issued when a function is declared without a return type, when the definition of a function returning void contains a return statement with an expression, or when the closing brace of a function returning non-void is reached.
Wshadow	Determines whether a warning is issued when a variable declaration hides a previous declaration.
Wsign-compare	Determines whether warnings are issued when a comparison between signed and unsigned values could produce an incorrect result when the signed value is converted to unsigned.
Wstrict-aliasing	Determines whether warnings are issued for code that might violate the optimizer's strict aliasing rules.
Wstrict-prototypes	Determines whether warnings are issued for functions declared or defined without specified argument types.
Wtrigraphs	Determines whether warnings are issued if any trigraphs are encountered that might change the meaning of the program.
Wuninitialized	Determines whether a warning is issued if a variable is used before being initialized.
Wunknown-pragmas	Determines whether a warning is issued if an unknown #pragma directive is used.
Wunused-function	Determines whether a warning is issued if a declared function is not used.
Wunused-variable	Determines whether a warning is issued if a local or non-constant static variable is unused after being declared.

Wwrite-strings	Issues a diagnostic message if const char * is converted to (non-const) char *.
X	Removes standard directories from the include file search path.
x (type option)	All source files found subsequent to -x type will be recognized as a particular type.
x, Qx	Tells the compiler which processor features it may target, including which instruction sets and optimizations it may generate.
xHost, QxHost	Tells the compiler to generate instructions for the highest instruction set available on the compilation host processor.
Xlinker	Passes a linker option directly to the linker.
Xopenmp-target	Enables options to be passed to the specified tool in the device compilation tool chain for the target.
Xs	Passes options to the backend tool.
Xsycl-target	Enables options to be passed to the specified tool in the device compilation tool chain for the target.
Y-	Tells the compiler to ignore all other precompiled header files.
Yc	Tells the compiler to create a precompiled header file.
Yu	Tells the compiler to use a precompiled header file.
Zc	Lets you specify ANSI C standard conformance for certain language features.
Zg	Tells the compiler to generate function prototypes. This is a deprecated option that may be removed in a future release.
Zi, Z7 , ZI	Tells the compiler to generate full debugging information in either an object (.obj) file or a project database (PDB) file.
Zl	Causes library names to be omitted from the object file.
Zp	Specifies alignment for structures on byte boundaries.
Zs	Tells the compiler to check only for correct syntax.

General Rules for Compiler Options

This section describes general rules for compiler options and it contains information about how we refer to compiler option names in descriptions.

General Rules for Compiler Options

Compiler options may be case sensitive, and may have different meanings depending on their case. For example, option `c` prevents linking, but option `C` places comments in preprocessed source output.

Options specified on the command line apply to all files named on the command line.

Options can take arguments in the form of file names, strings, letters, or numbers. If a string includes spaces, the string must be enclosed in quotation marks.

Compiler options can appear in any order.

Unless you specify certain options, the command line will both compile and link the files you specify.

You can abbreviate some option names, entering as many characters as are needed to uniquely identify the option.

Certain options accept one or more keyword arguments following the option name. For example, architecture option `x` option accepts several keywords.

To specify multiple keywords, you typically specify the option multiple times.

To disable an option, specify the negative form of the option if one exists.

If there are enabling and disabling versions of an option on the command line, the last one on the command line takes precedence.

Compiler options remain in effect for the whole compilation unless overridden by a compiler #pragma.

Linux

You cannot combine options with a single dash. For example, this form is incorrect: `-Ec`; this form is correct: `-E -c`

Windows

You cannot combine options with a single slash. For example: This form is incorrect: `/Ec`; this form is correct: `/E /c`

All compiler options must precede `/link` options, if any, on the command line.

Compiler options remain in effect for the whole compilation unless overridden by a compiler #pragma.

You can disable one or more optimization options by specifying option `/Od` last on the command line.

NOTE
The `/Od` option is part of a mutually-exclusive group of options that includes `/Od`, `/O1`, `/O2`, `/O3`, and `/Ox`. The last of any of these options specified on the command line will override the previous options from this group.

How We Refer to Compiler Option Names in Descriptions

Within documentation, compiler option names that are very different are spelled out in full.

However, many compiler option names are very similar except for initial characters. For these options, we use the following shortcuts when referencing their names in descriptions:

- No initial – or /

  This shortcut is used for option names that are the same for Linux and Windows except for the initial character.

  For example, `Fa` denotes:

  - Linux: `-Fa`
  - Windows: `/Fa`
- [Q]option-name

  This shortcut is used for option names that only differ because the Windows form starts with a Q.

  For example, [Q]ipo denotes:

  - Linux: `-ipo`
  - Windows: `/Qipo`
- [q or Q]option-name

This shortcut is used for option names that only differ because the Linux form starts with a q and the Windows form starts with a Q.

For example, [q or Q]opt-report denotes:

- Linux: `-qopt-report`
- Windows: `/Qopt-report`

What Appears in the Compiler Option Descriptions

This section contains details about what appears in the option descriptions.

Following sections include individual descriptions of all the current compiler options. The option descriptions are arranged by functional category. Within each category, the option names are listed in alphabetical order.

Each option description contains the following information:

- The primary name for the option and a short description of the option.
- Architecture Restrictions: This section only appears if there is a known architecture restriction for the option. Restrictions can appear for any of the following architectures:
  - IA-32 architecture (does not apply to SYCL)
  - Intel® 64 architecture

  Certain operating systems are not available on all the above architectures. For the latest information, check your Release Notes.
- Syntax: This section shows the syntax on Linux systems and the syntax on Windows systems. If the option is not valid on a particular operating system, it will specify **None**.
- Arguments: This section shows any arguments (parameters) that are related to the option. If the option has no arguments, it will specify **None**.
- Default: This section shows the default setting for the option.
- Description: This section shows the full description of the option. It may also include further information on any applicable arguments.
- IDE Equivalent: This section shows information related to the Intel® Integrated Development Environment (Intel® IDE) Property Pages on Linux and Windows systems. It shows on which Property Page the option appears, and under what category it's listed. The Windows IDE is Microsoft Visual Studio .NET. If the option has no IDE equivalent, it will specify **None**.
- Alternate Options (does not apply to SYCL): This section lists any options that are synonyms for the described option. If there are no alternate option names, it will show **None**. Some alternate option names are deprecated and may be removed in future releases. Many options have an older spelling where underscores ("_") instead of hyphens ("-") connect the main option names. The older spelling is a valid alternate option name.

Some option descriptions may also have the following:

- Example (or Examples): This section shows one or more examples that demonstrate the option.
- See Also: This section shows where you can get further information on the option or it shows related options.

Optimization Options

This section contains descriptions for compiler options that pertain to optimization. They are listed in alphabetical order.

fast

Maximizes speed across the entire program.

Syntax

Linux OS:

`-fast`

Windows OS:

`/fast`

Arguments

None

Default

OFF	The optimizations that maximize speed are not enabled.

Description

This option maximizes speed across the entire program.

Linux

It sets the following options:

`-ipo`, `-O3`, `-static`, `-fp-model fast=2`

Windows

It sets the following options:

`/O3`, `/Qipo`, `/Qprec-div-`, `/fp:fast=2`

> **NOTE**
> Option `fast` sets some aggressive optimizations that may not be appropriate for all applications. The resulting executable may not run on processor types different from the one on which you compile. You should make sure that you understand the individual optimization options that are enabled by option `fast`.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.
Notice revision #20201201

IDE Equivalent

None

Alternate Options

None

See Also

fp-model, fp compiler option

xHost, QxHost compiler option

x, Qx

compiler option

fbuiltin, Oi

Enables or disables inline expansion of intrinsic functions.

Syntax

Linux OS:

`-fbuiltin[-name]`

`-fno-builtin[-name]`

Windows OS:

`/Oi[-]`

`/Qno-builtin-name`

Arguments

name	Is a list of one or more intrinsic functions. If there is more than one intrinsic function, they must be separated by commas.

Default

ON	Inline expansion of intrinsic functions is enabled.

Description

This option enables or disables inline expansion of one or more intrinsic functions.

If `-fno-builtin-`*name* or `/Qno-builtin-`*name* is specified, inline expansion is disabled for the named functions. If *name* is not specified, `-fno-builtin` or `/Oi-` disables inline expansion for all intrinsic functions.

For a list of built-in functions affected by `-fbuiltin`, search for "built-in functions" in the appropriate gcc* documentation.

For a list of built-in functions affected by `/Oi`, search for "/Oi" in the appropriate Microsoft* Visual C/C++* documentation.

IDE Equivalent

Windows

Visual Studio: **Optimization > Enable Intrinsic Functions** (/Oi)

Linux

Eclipse: None

Alternate Options

None

foptimize-sibling-calls

Determines whether the compiler optimizes tail recursive calls.

Syntax

Linux OS:

`-foptimize-sibling-calls`

`-fno-optimize-sibling-calls`

Windows OS:

`None`

Arguments

None

Default

`-foptimize-sibling-calls`	The compiler optimizes tail recursive calls.

Description

This option determines whether the compiler optimizes tail recursive calls. It enables conversion of tail recursion into loops.

If you do not want to optimize tail recursive calls, specify `-fno-optimize-sibling-calls`.

Tail recursion is a special form of recursion that doesn't use stack space. In tail recursion, a recursive call is converted to a GOTO statement that returns to the beginning of the function. In this case, the return value of the recursive call is only used to be returned. It is not used in another expression. The recursive function is converted into a loop, which prevents modification of the stack space used.

IDE Equivalent

None

Alternate Options

None

GF

Enables read-only string-pooling optimization.

Syntax

Linux OS:

`None`

Windows OS:

`/GF`

Arguments

None

Default

OFF	Read/write string-pooling optimization is enabled.

Description

This option enables read only string-pooling optimization.

IDE Equivalent

Windows

Visual Studio: **Code Generation > Enable String Pooling**

Linux

Eclipse: None

Alternate Options

None

nolib-inline

Disables inline expansion of standard library or intrinsic functions.

Syntax

Linux OS:

`-nolib-inline`

Windows OS:

`None`

Arguments

None

Default

OFF	The compiler inlines many standard library and intrinsic functions.

Description

This option disables inline expansion of standard library or intrinsic functions. It prevents the unexpected results that can arise from inline expansion of these functions.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

Windows

Visual Studio: None

Linux

Eclipse: **Optimization > Disable Intrinsic Inline Expansion**

Alternate Options

None

O

Specifies the code optimization for applications.

Syntax

Linux OS:

`-O[n]`

Windows OS:

`/O[n]`

Arguments

n	Is the optimization level. Possible values are 1, 2, or 3. On Linux* systems, you can also specify 0.

Default

O2	Optimizes for code speed. This default may change depending on which other compiler options are specified. For details, see below.

Description

This option specifies the code optimization for applications.

Option	Description
O (Linux*)	This is the same as specifying O2.
O0 (Linux)	Disables all optimizations. This option may set other options. This is determined by the compiler, depending on which operating system and architecture you are using. The options that are set may change from release to release.
O1	Enables optimizations for speed and disables some optimizations that increase code size and affect speed. To limit code size, this option: • Enables global optimization; this includes data-flow analysis, code motion, strength reduction and test replacement, split-lifetime analysis, and instruction scheduling. • Disables inlining of some intrinsics. This option may set other options. This is determined by the compiler, depending on which operating system and architecture you are using. The options that are set may change from release to release. The O1 option may improve performance for applications with very large code size, many branches, and execution time not dominated by code within loops.
O2	Enables optimizations for speed. This is the generally recommended optimization level.

Option	Description
	Vectorization is enabled at O2 and higher levels. This content does not apply to SYCL. On systems using IA-32 architecture: Some basic loop optimizations such as Distribution, Predicate Opt, Interchange, multi-versioning, and scalar replacements are performed. This option also enables: • Inlining of intrinsics • Intra-file interprocedural optimization, which includes: • inlining • constant propagation • forward substitution • routine attribute propagation • variable address-taken analysis • dead static function elimination • removal of unreferenced variables • The following capabilities for performance gain: • constant propagation • copy propagation • dead-code elimination • global register allocation • global instruction scheduling and control speculation • loop unrolling • optimized code selection • partial redundancy elimination • strength reduction/induction variable simplification • variable renaming • exception handling optimizations • tail recursions • peephole optimizations • structure assignment lowering and optimizations • dead store elimination This option may set other options, especially options that optimize for code speed. This is determined by the compiler, depending on which operating system and architecture you are using. The options that are set may change from release to release. This content does not apply to SYCL. On Linux systems, the `-debug inline-debug-info` option will be enabled by default if you compile with optimizations (option `-O2` or higher) and debugging is enabled (option `-g`). Many routines in the shared libraries are more highly optimized for Intel® microprocessors than for non-Intel microprocessors.
O3	Performs O2 optimizations and enables more aggressive loop transformations such as Fusion, Block-Unroll-and-Jam, and collapsing IF statements. This option may set other options. This is determined by the compiler, depending on which operating system and architecture you are using. The options that are set may change from release to release.

Option	Description
	The `O3` optimizations may not cause higher performance unless loop and memory access transformations take place. The optimizations may slow down code in some cases compared to `O2` optimizations. The `O3` option is recommended for applications that have loops that heavily use floating-point calculations and process large data sets. Many routines in the shared libraries are more highly optimized for Intel® microprocessors than for non-Intel microprocessors.

The last `O` option specified on the command line takes precedence over any others.

IDE Equivalent

Windows

Visual Studio: **Optimization > Optimization**

Linux

Eclipse: **General > Optimization Level**

Alternate Options

`O0`	Linux: None Windows: `/Od`

See Also

`Od` compiler option

Od

Disables all optimizations.

Syntax

Linux OS:

`None`

Windows OS:

`/Od`

Arguments

None

Default

OFF	The compiler performs default optimizations.

Description

This option disables all optimizations. It can be used for selective optimizations, such as a combination of `/Od` and `/Ob1` (disables all optimizations, but enables inlining).

This content does not apply to SYCL. On IA-32 architecture, this option sets the `/Oy-` option.

IDE Equivalent

Visual Studio

Visual Studio: **Optimization > Optimization**

Eclipse

Eclipse: None

Alternate Options

Linux: `-O0`

Windows: None

See Also

- compiler option (see O0)

Ofast

Sets certain aggressive options to improve the speed of your application.

Syntax

Linux OS:

`-Ofast`

Windows OS:

`None`

Arguments

None

Default

OFF	The aggressive optimizations that improve speed are not enabled.

Description

This option improves the speed of your application.

This option is provided for compatibility with gcc.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

None

See Also

`O` compiler option

`fast` compiler option

`fp-model, fp` compiler option

Os

Enables optimizations that do not increase code size; it produces smaller code size than O2.

Syntax

Linux OS:

`-Os`

Windows OS:

`/Os`

Arguments

None

Default

OFF	Optimizations are made for code speed. However, if `O1` is specified, `Os` is the default.

Description

This option enables optimizations that do not increase code size; it produces smaller code size than `O2`. It disables some optimizations that increase code size for a small speed benefit.

This option tells the compiler to favor transformations that reduce code size over transformations that produce maximum performance.

IDE Equivalent

Visual Studio

Visual Studio: **Optimization > Favor Size or Speed**

Eclipse

Eclipse: None

Alternate Options

None

See Also

`O` compiler option

`Ot` compiler option

Ot

Enables all speed optimizations.

Syntax

Linux OS:

`None`

Windows OS:

`/Ot`

Arguments

None

Default

`/Ot`	Optimizations are made for code speed. If `Od` is specified, all optimizations are disabled. If `O1` is specified, `Os` is the default.

Description

This option enables all speed optimizations.

IDE Equivalent

Windows

Visual Studio: **Optimization > Favor Size or Speed**

Linux

Eclipse: None

Alternate Options

None

See Also

`O` compiler option

`Os` compiler option

Ox

Enables maximum optimizations.

Syntax

Linux OS:

`None`

Windows OS:

`/Ox`

Arguments

None

Default

OFF	The compiler does not enable optimizations.

Description

The compiler enables maximum optimizations by combining the following options:

- /Oi
- /Ot
- C++ only: /Oy

IDE Equivalent

Windows

Visual Studio: **Optimization > Optimization**

Linux

Eclipse: None

Alternate Options

None

Code Generation Options

This section contains descriptions for compiler options that pertain to code generation. They are listed in alphabetical order.

arch

Tells the compiler which features it may target, including which instruction sets it may generate.

Syntax

Linux OS:

`None`

Windows OS:

`/arch:`*`code`*

Arguments

code — Indicates to the compiler a feature set that it may target, including which instruction sets it may generate. Many of the following descriptions refer to Intel® Streaming SIMD Extensions (Intel® SSE) and Supplemental Streaming SIMD Extensions (SSSE). Possible values are:

`ALDERLAKE` `AMBERLAKE` `BROADWELL` `CANNONLAKE` `CASCADELAKE` `COFFEELAKE` `COOPERLAKE` `GOLDMONT` `GOLDMONT-PLUS` `HASWELL` `ICELAKE-CLIENT` (or `ICELAKE`) `ICELAKE-SERVER`	May generate instructions for processors that support the specified Intel® processor or microarchitecture code name. Keyword `ICELAKE` is deprecated and may be removed in a future release.

IVYBRIDGE KABYLAKE KNL KNM ROCKETLAKE SANDYBRIDGE SAPPHIRERAPIDS SILVERMONT SKYLAKE SKYLAKE-AVX512 TIGERLAKE TREMONT WHISKEYLAKE	
CORE-AVX2	May generate Intel® Advanced Vector Extensions 2 (Intel® AVX2), Intel® AVX, SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions.
CORE-AVX-I	May generate Float-16 conversion instructions and the RDRND instruction, Intel® Advanced Vector Extensions (Intel® AVX), Intel® SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions.
AVX2	May generate Intel® Advanced Vector Extensions 2 (Intel® AVX2), Intel® AVX, Intel® SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions.
AVX	May generate Intel® Advanced Vector Extensions (Intel® AVX), Intel® SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions.
SSE4.2	May generate Intel® SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions.
SSE4.1	May generate Intel® SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions.
SSSE3	May generate SSSE3 instructions and Intel® SSE3, SSE2, and SSE instructions.
SSE3	May generate Intel® SSE3, SSE2, and SSE instructions.
SSE2	May generate Intel® SSE2 and SSE instructions. This setting is only supported for C++ (icx).
SSE	This option has been deprecated; it is now the same as specifying `IA32`.
IA32	Generates x86/x87 generic code that is compatible with IA-32 architecture. Disables any default extended instruction settings, and any previously set extended instruction settings. It also disables all feature-specific optimizations and instructions. This value is only available on IA-32 architecture.

Default

varies	If option `arch` is not specified, the default target architecture supports Intel® SSE2 instructions.

Description

This option tells the compiler which features it may target, including which instruction sets it may generate.

Code generated with these options should execute on any compatible, non-Intel processor with support for the corresponding instruction set.

Options `/arch` and `/Qx` are mutually exclusive. If both are specified, the compiler uses the last one specified and generates a warning.

If you specify both the `/Qax` and `/arch` options, the compiler will not generate Intel-specific instructions.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

Visual Studio

Visual Studio: **Code Generation > Enable Enhanced Instruction Set**

Eclipse

Eclipse: None

Xcode

Xcode: None

Alternate Options

None

See Also

`x, Qx` compiler option
`xHost, QxHost` compiler option
`ax, Qax` compiler option
`arch` compiler option
`march` compiler option
`m` compiler option
`m32, m64` compiler option

ax, Qax

Tells the compiler to generate multiple, feature-specific auto-dispatch code paths for Intel® processors if there is a performance benefit.

Syntax

Linux OS:

`-ax`*`code`*

Windows OS:

`/Qax`*`code`*

Arguments

code	Indicates to the compiler a feature set that it may target, including which instruction sets it may generate. The following descriptions refer to Intel® Streaming SIMD Extensions (Intel® SSE) and Supplemental Streaming SIMD Extensions (SSSE). Possible values are:

`ALDERLAKE` `AMBERLAKE` `BROADWELL` `CANNONLAKE` `CASCADELAKE` `COFFEELAKE` `COOPERLAKE` `GOLDMONT` `GOLDMONT-PLUS` `HASWELL` `ICELAKE-CLIENT` (or `ICELAKE`) `ICELAKE-SERVER` `IVYBRIDGE` `KABYLAKE` `KNL` `KNM` `ROCKETLAKE` `SANDYBRIDGE` `SAPPHIRERAPIDS` `SILVERMONT` `SKYLAKE` `SKYLAKE-AVX512` `TIGERLAKE` `TREMONT` `WHISKEYLAKE`	May generate instructions for processors that support the specified Intel® processor or microarchitecture code name. Keywords `KNL` and `SILVERMONT` are only available on Windows* and Linux* systems. Keyword `ICELAKE` is deprecated and may be removed in a future release.
`COMMON-AVX512`	May generate Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Foundation instructions, Intel® AVX-512 Conflict Detection Instructions (CDI), as well as the instructions enabled with CORE-AVX2.
`CORE-AVX512`	May generate Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Foundation instructions, Intel® AVX-512 Conflict Detection Instructions (CDI), Intel® AVX-512 Doubleword and Quadword Instructions (DQI), Intel® AVX-512 Byte and Word Instructions (BWI) and Intel® AVX-512 Vector Length extensions, as well as the instructions enabled with CORE-AVX2.

`CORE-AVX2`	May generate Intel® Advanced Vector Extensions 2 (Intel® AVX2), Intel® AVX, SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions for Intel® processors.
`CORE-AVX-I`	May generate Float-16 conversion instructions and the RDRND instruction, Intel® Advanced Vector Extensions (Intel® AVX), Intel® SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions for Intel® processors.
`AVX`	May generate Intel® Advanced Vector Extensions (Intel® AVX), Intel® SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions for Intel® processors.
`SSE4.2`	May generate Intel® SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions for Intel processors.
`SSE4.1`	May generate Intel® SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions for Intel® processors.
`SSSE3`	May generate SSSE3 instructions and Intel® SSE3, SSE2, and SSE instructions for Intel® processors. This replaces value T, which is deprecated.
`SSE3`	May generate Intel® SSE3, SSE2, and SSE instructions for Intel® processors.
`SSE2`	May generate Intel® SSE2 and SSE instructions for Intel® processors.

You can specify more than one *code* value. When specifying more than one *code* value, each value must be separated with a comma. See the Examples section below.

Default

OFF — No auto-dispatch code is generated. Feature-specific code is generated and is controlled by the setting of the following compiler options:

- Linux*: `-m` and `-x`
- Windows*: `/arch` and `/Qx`

Description

This option tells the compiler to generate multiple, feature-specific auto-dispatch code paths for Intel® processors if there is a performance benefit. It also generates a baseline code path. The Intel feature-specific auto-dispatch path is usually more optimized than the baseline path. Other options, such as `O3`, control how much optimization is performed on the baseline path.

The baseline code path is determined by the architecture specified by options `-m` or `-x` (Linux*) or options `/arch` or `/Qx` (Windows*). While there are defaults for the `[Q]x` option that depend on the operating system being used, you can specify an architecture and optimization level for the baseline code that is higher or lower than the default. The specified architecture becomes the effective minimum architecture for the baseline code path.

If you specify both the `[Q]ax` and `[Q]x` options, the baseline code will only execute on Intel® processors compatible with the setting specified for the `[Q]x`.

If you specify both the `-ax` and `-m` options (Linux) or the `/Qax` and `/arch` options (Windows), the baseline code will execute on non-Intel® processors compatible with the setting specified for the `-m` or `/arch` option.

A Non-Intel® baseline and an Intel® baseline have the same set of optimizations enabled, and the default for both is SSE4.2 for x86-based architectures.

If you specify both the `-ax` and `-march` options (Linux), or the `/Qax` and `/arch` options (Windows), the compiler will not generate Intel-specific instructions. This is because specifying `-march` disables `-ax` and specifying `/arch` disables `/Qax`.

The `[Q]ax` option tells the compiler to find opportunities to generate separate versions of functions that take advantage of features of the specified instruction features.

If the compiler finds such an opportunity, it first checks whether generating a feature-specific version of a function is likely to result in a performance gain. If this is the case, the compiler generates both a feature-specific version of a function and a baseline version of the function. At run time, one of the versions is chosen to execute, depending on the Intel® processor in use. In this way, the program can benefit from performance gains on more advanced Intel processors, while still working properly on older processors and non-Intel processors. A non-Intel processor always executes the baseline code path.

You can use more than one of the feature values by combining them. For example, you can specify `-axSSE4.1,SSSE3` (Linux) or `/QaxSSE4.1,SSSE3` (Windows). You cannot combine the old style, deprecated options and the new options. For example, you cannot specify `-axSSE4.1,T` (Linux) or `/QaxSSE4.1,T` (Windows).

NOTE
This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

IDE Equivalent

Visual Studio

Visual Studio: **Code Generation > Add Processor-Optimized Code Path**

Eclipse

Eclipse: **Code Generation > Add Processor-Optimized Code Path**

Xcode

Xcode: **Code Generation > Add Processor-Optimized Code Path**

Alternate Options

None

Examples

The following shows an example of how to specify this option:

The following shows an example of how to specify more than one *code* value:

Note that the comma-separated list must have no spaces between the names.

See Also

`x, Qx` compiler option
`xHost, QxHost` compiler option
`march` compiler option
`arch` compiler option
`m` compiler option

EH

Specifies the model of exception handling to be performed.

Syntax

Linux OS:

`None`

Windows OS:

`/EH`*`type`*

`/EH`*`type`*`-`

Arguments

type	Specifies the exception handling model. Possible values are:	
	`a`	Specifies the asynchronous C++ exception handling model.
	`s`	Specifies the synchronous C++ exception handling model.
	`c`	Tells the compiler to assume that extern "C" functions do not throw exceptions.
	`r`	Tells the compiler to always generate runtime termination checks for all noexcept functions. IT forces runtime termination checks in all functions that have a noexcept attribute.

If you specify `c`, you must also specify `a` or `s`.

Default

OFF	Some exception handling is performed by default.

Description

This option specifies the model of exception handling to be performed.

If you specify the negative form of the option, it disables the exception handling performed by *type* or the last *type* if there are two. For example, if you specify `/EHsc-`, it is interpreted as `/EHs`.

For more details about option `/EH`, see the Microsoft documentation.

IDE Equivalent

Windows

Visual Studio: **Code Generation > Enable C++ Exceptions**

Linux

Eclipse: None

Alternate Options

`/EHsc`	Linux: None Windows: `/GX`

fasynchronous-unwind-tables

Determines whether unwind information is precise at an instruction boundary or at a call boundary.

Syntax

Linux OS:

`-fasynchronous-unwind-tables`

`-fno-asynchronous-unwind-tables`

Windows OS:

None

Arguments

None

Default

Intel® 64 architecture: `-fasynchronous-unwind-tables`	The unwind table generated is precise at an instruction boundary, enabling accurate unwinding at any instruction.
C++: IA-32 architecture (Linux* only): `-fno-asynchronous-unwind-tables`	The unwind table generated is precise at call boundaries only.

Description

This option determines whether unwind information is precise at an instruction boundary or at a call boundary. The compiler generates an unwind table in DWARF2 or DWARF3 format, depending on which format is supported on your system.

If `-fno-asynchronous-unwind-tables` is specified, the unwind table is precise at call boundaries only. In this case, the compiler will avoid creating unwind tables for routines such as the following:

- A C++ routine that does not declare objects with destructors and does not contain calls to routines that might throw an exception.
- A C/C++ or Fortran routine compiled without `-fexceptions`.
- A C/C++ or Fortran routine compiled with `-fexceptions` that does not contain calls to routines that might throw an exception.

IDE Equivalent

None

Alternate Options

None

See Also

fexceptions compiler option

fdata-sections, Gw

Places each data item in its own COMDAT section.

Syntax

Linux OS:

`-fdata-sections`

Windows OS:

`/Gw`

Arguments

None

Default

OFF	The compiler does not separate functions into COMDATs.

Description

This option places each data item in its own COMDAT section.

When using this compiler option, you can add the linker option `-Wl,--gc-sections` (LInux) or `/link /OPT:REF` (Windows), which will remove all unused code.

NOTE
When you put each data item in its own section, it enables the linker to reorder the sections for other possible optimization.

Alternate Options

None

See Also

`ffunction-sections, Gy` compiler option

fexceptions

Enables exception handling table generation.

Syntax

Linux OS:

`-fexceptions`

`-fno-exceptions`

Windows OS:

`None`

Arguments

None

Default

-fexceptions	Exception handling table generation is enabled. Default for C++.
-fno-exceptions	Exception handling table generation is disabled. Default for C.

Description

This option enables exception handling table generation. The `-fno-exceptions` option disables exception handling table generation, resulting in smaller code. When this option is used, any use of exception handling constructs (such as try blocks and throw statements) will produce an error. Exception specifications are parsed but ignored. It also undefines the preprocessor symbol __EXCEPTIONS.

IDE Equivalent

None

Alternate Options

None

ffunction-sections, Gy

Places each function in its own COMDAT section.

Syntax

Linux OS:

`-ffunction-sections`

Windows OS:

`/Gy`

Arguments

None

Default

OFF	The compiler does not separate functions into COMDATs.

Description

This option places each function in its own COMDAT section.

When using this compiler option, you can add the linker option `-Wl,--gc-sections` (LInux) or `/link /OPT:REF` (Windows), which will remove all unused code.

NOTE
When you put each function in its own section, it enables the linker to reorder the sections for other possible optimization.

IDE Equivalent

Windows

Visual Studio: **Code Generation > Enable Function-Level Linking**

Linux

Eclipse: None

Alternate Options

None

See Also

`fdata-sections, Gw` compiler option

fomit-frame-pointer, Oy

Determines whether EBP is used as a general-purpose register in optimizations.

Architecture Restrictions

Option `/Oy[-]` is only available on IA-32 architecture. IA-32 architecture is only supported for C++.

Syntax

Linux OS:

`-fomit-frame-pointer`

`-fno-omit-frame-pointer`

Windows OS:

`/Oy (C++ only)`

`/Oy- (C++ only)`

Arguments

None

Default

`-fomit-frame-pointer` C++: or `/Oy`	EBP is used as a general-purpose register in optimizations. However, the default can change depending on the following: **Linux** If option `-O0` or `-g` is specified, the default is `-fno-omit-frame-pointer`. **Windows** C++: If option `/Od` is specified, the default is `/Oy-` .

Description

These options determine whether EBP is used as a general-purpose register in optimizations. Option `-fomit-frame-pointer` and option `/Oy` allows this use. Option `-fno-omit-frame-pointer` and option `/Oy-` disallows it.

Some debuggers expect EBP to be used as a stack frame pointer, and cannot produce a stack backtrace unless this is so. The `-fno-omit-frame-pointer` and the `/Oy-` option directs the compiler to generate code that maintains and uses EBP as a stack frame pointer for all functions so that a debugger can still produce a stack backtrace without doing the following:

- For `-fno-omit-frame-pointer`: turning off optimizations with `-O0`
- This content does not apply to SYCL.

  C++: For `/Oy-`: turning off `/O1`, `/O2`, or `/O3` optimizations

The `-fno-omit-frame-pointer` option is set when you specify option `-O0` or the `-g` option. The `-fomit-frame-pointer` option is set when you specify option `-O1`, `-O2`, or `-O3`.

For C++, the `/Oy` option is set when you specify the `/O1`, `/O2`, or `/O3` option. Option `/Oy-` is set when you specify the `/Od` option.

Using the `-fno-omit-frame-pointer` or `/Oy-` option reduces the number of available general-purpose registers by 1, and can result in slightly less efficient code.

> **NOTE**
> **Linux**
>
> There is currently an issue with GCC 3.2 exception handling. Therefore, the compiler ignores this option when GCC 3.2 is installed for C++ and exception handling is turned on (the default).

IDE Equivalent

Windows

Visual Studio: **Optimization > Omit Frame Pointers**

Linux

Eclipse: **Optimization > Provide Frame Pointer**

Alternate Options

Linux: `-fp` (this is a deprecated option)

Windows: None

See Also

momit-leaf-frame-pointer compiler option

Gd

Makes __cdecl the default calling convention.

Architecture Restrictions

Not available on IA-32 architecture. IA-32 architecture is only supported for C++.

Syntax

Linux OS:

`None`

Windows OS:

`/Gd`

Arguments

None

Default

ON	The default calling convention is `__cdecl`.

Description

This option makes `__cdecl` the default calling convention.

IDE Equivalent

Windows

Visual Studio: **Advanced > Calling Convention**

Linux

Eclipse: None

Alternate Options

None

See Also

C C++ Calling Conventions

Gr

Makes __fastcall the default calling convention.

Architecture Restrictions

Only available on IA-32 architecture

Syntax

Linux OS:

`None`

Windows OS:

`/Gr`

Arguments

None

Default

OFF	The default calling convention is `__cdecl`

Description

This option makes `__fastcall` the default calling convention.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

Windows

Visual Studio: **Advanced > Calling Convention**

Linux

Eclipse: None

Alternate Options

None

See Also

C C++ Calling Conventions

GR

Enables or disables C++ Run Time Type Information (RTTI).

Syntax

Linux OS:

`None`

Windows OS:

`/GR`

`/GR-`

Arguments

None

Default

`/GR`	C++ Run Time Type Information (RTTI) is enabled.

Description

This option enables or disables C++ Run Time Type Information (RTTI).

To disable C++ Run Time Type Information (RTTI), specify option `/GR-`.

IDE Equivalent

Windows

Visual Studio: **Language > Enable Run-Time Type Information**

Linux

Eclipse: None

Alternate Options

None

guard

Enables the control flow protection mechanism.

Syntax

Linux OS:

`None`

Windows OS:

`/guard:`*`keyword`*

Arguments

keyword	Specifies the the control flow protection mechanism. Possible values are:	
	`cf[-]`	Tells the compiler to analyze control flow of valid targets for indirect calls and to insert code to verify the targets at runtime. To explicitly disable this option, specify `/guard:cf-`.

Default

`OFF`	The control flow protection mechanism is disabled.

Description

This option enables the control flow protection mechanism. It tells the compiler to analyze control flow of valid targets for indirect calls and inserts a call to a checking routine before each indirect call to verify the target of the given indirect call.

The `/guard:cf` option must be passed to both the compiler and linker.

Code compiled using `/guard:cf` can be linked to libraries and object files that are not compiled using the option.

This option has been added for Microsoft compatibility. It uses the Microsoft implementation.

NOTE
This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

Windows

Visual Studio: **Code Generation > Control Flow Guard**

Linux

Eclipse: None

Alternate Options

None

Gv

Tells the compiler to use the vector calling convention (__vectorcall) when passing vector type arguments.

Syntax

Linux OS:

`None`

Windows OS:

`/Gv`

Arguments

None

Default

OFF	The default calling convention is `__cdecl`.

Description

This option tells the compiler to use the vector calling convention (__vectorcall) when passing vector type arguments.

It causes each function in the module to compile as __vectorcall unless the function is declared with a conflicting attribute, or the name of the function is main.

This option has been added for Microsoft compatibility.

For more details about the __vectorcall calling convention, see the Microsoft documentation.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

Windows

Visual Studio: **Advanced > Calling Convention**

Linux

Eclipse: None

Alternate Options

None

See Also

C C++ Calling Conventions

Gz

Makes __stdcall the default calling convention.

Architecture Restrictions

Only available on IA-32 architecture

Syntax

Linux OS:

`None`

Windows OS:

`/Gz`

Arguments

None

Default

OFF	The default calling convention is `__cdecl`.

Description

This option makes `__stdcall` the default calling convention.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

Windows

Visual Studio: **Advanced > Calling Convention**

Linux

Eclipse: None

Alternate Options

None

See Also

C C++ Calling Conventions

m

Tells the compiler which features it may target, including which instruction set architecture (ISA) it may generate.

Syntax

Linux OS:

`-m`*`code`*

Windows OS:

None

Arguments

code	Indicates to the compiler a feature set that it may target, including which instruction sets it may generate. Many of the following descriptions refer to Intel® Streaming SIMD Extensions (Intel® SSE) and Supplemental Streaming SIMD Extensions (SSSE). Possible values are:

`avx`	May generate Intel® Advanced Vector Extensions (Intel® AVX), SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions.
`sse4.2`	May generate Intel® SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions.
`sse4.1`	May generate Intel® SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions.
`ssse3`	May generate SSSE3 instructions and Intel® SSE3, SSE2, and SSE instructions.
`sse3`	May generate Intel® SSE3, SSE2, and SSE instructions.
`sse2`	May generate Intel® SSE2 and SSE instructions.
`sse`	This setting has been deprecated; it is the same as specifying `ia32`.
`ia32`	Generates x86/x87 generic code that is compatible with IA-32 architecture. Disables any default extended instruction settings, and any previously set extended instruction settings. It also disables all feature-specific optimizations and instructions. This value is only available on Linux* systems using IA-32 architecture.

This compiler option also supports many of the `-m` option settings available with gcc. For more information on gcc settings for `-m`, see the gcc documentation.

Default

varies	If option `arch` is not specified, the default target architecture supports Intel® SSE2 instructions.

Description

This option tells the compiler which features it may target, including which instruction sets it may generate.

Code generated with these options should execute on any compatible, non-Intel processor with support for the corresponding instruction set.

For compatibility with gcc, the compiler allows the following options but they have no effect. You will get a warning error, but the instructions associated with the name will not be generated. You should use the suggested replacement options.

gcc Compatibility Option	Suggested Replacement Option
`-mfma`	`-march=core-avx2`
`-mbmi, -mavx2, -mlzcnt`	`-march=core-avx2`
`-mmovbe`	`-march=atom -minstruction=movbe`
`-mcrc32, -maes, -mpclmul, -mpopcnt`	`-march=corei7`
`-mvzeroupper`	`-march=corei7-avx`
`-mfsgsbase, -mrdrnd, -mf16c`	`-march=core-avx-i`

NOTE
This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

None

See Also

`x, Qx` compiler option
`xHost, QxHost` compiler option
`ax, Qax` compiler option
`arch` compiler option
`march` compiler option
`m32, m64` compiler option

m32, m64, Qm32, Qm64

Tells the compiler to generate code for a specific architecture. Option m32 (and Qm32) is deprecated and will be removed in a future release.

Syntax

Linux OS:

`-m32 (C++ only)`

`-m64`

Windows OS:

`/Qm32 (C++ only)`

`/Qm64 (C++ only)`

Windows OS:

`None (SYCL only)`

Arguments

None

Default

OFF	The compiler's behavior depends on the host system.

Description

These options tell the compiler to generate code for a specific architecture.

Option	Description
`C++: -m32` or `/Qm32`	Tells the compiler to generate code for IA-32 architecture.
`-m64` C++: or `/Qm64`	Tells the compiler to generate code for Intel® 64 architecture.

On Linux* systems, these options are provided for compatibility with gcc.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

None

m80387

Specifies whether the compiler can use x87 instructions.

Syntax

Linux OS:

`-m80387`

`-mno-80387`

Windows OS:

`None`

Arguments

None

Default

`-m80387`	The compiler may use x87 instructions.

Description

This option specifies whether the compiler can use x87 instructions.

If you specify option `-mno-80387`, it prevents the compiler from using x87 instructions. If the compiler is forced to generate x87 instructions, it issues an error message.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

-m[no-]x87

march

Tells the compiler to generate code for processors that support certain features.

Syntax

Linux OS:

`-march=`*`processor`*

Windows OS:

`None`

Arguments

processor Tells the compiler the code it can generate. Possible values are:

`alderlake` `broadwell` `cannonlake` `cascadelake` `cooperlake` `goldmont` `goldmont-plus` `haswell` `icelake-client` `icelake-server` `ivybridge` `knl`	May generate instructions for processors that support the specified Intel® processor or microarchitecture code name. Keywords `knl` and `silvermont` are only available on Linux* systems.

`knm` `rocketlake` `sandybridge` `sapphirerapids` `silvermont` `skylake` `skylake-avx512` `tigerlake` `tremont`	
`core-avx2`	Generates code for processors that support Intel® Advanced Vector Extensions 2 (Intel® AVX2), Intel® AVX, SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions.
`core-avx-i`	Generates code for processors that support Float-16 conversion instructions and the RDRND instruction, Intel® Advanced Vector Extensions (Intel® AVX), Intel® SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions.
`corei7-avx`	Generates code for processors that support Intel® Advanced Vector Extensions (Intel® AVX), Intel® SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions.
`corei7`	Generates code for processors that support Intel® SSE4 Efficient Accelerated String and Text Processing instructions. May also generate code for Intel® SSE4 Vectorizing Compiler and Media Accelerator, Intel® SSE3, SSE2, SSE, and SSSE3 instructions.
`atom`	Generates code for processors that support MOVBE instructions. May also generate code for SSSE3 instructions and Intel® SSE3, SSE2, and SSE instructions.
`core2`	Generates code for the Intel® Core™2 processor family.
`pentium4m`	Generates for Intel® Pentium® 4 processors with MMX technology.
`pentium-m` `pentium4` `pentium3` `pentium`	Generates code for Intel® Pentium® processors. Value `pentium3` is only available on Linux* systems.

Default

`pentium4`	If no architecture option is specified, value `pentium4` is used by the compiler to generate code.

Description

This option tells the compiler to generate code for processors that support certain features.

Options `-x` and `-march` are mutually exclusive. If both are specified, the compiler uses the last one specified and generates a warning.

For compatibility, a number of historical *processor* values are also supported, but the generated code will not differ from the default.

NOTE
This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

IDE Equivalent

None

Alternate Options

`-march=pentium3`	Linux: `-xSSE` Windows: None
`-march=pentium4` `-march=pentium-m`	Linux: `-xSSE2` Windows: None
`-march=core2`	Linux: `-xSSSE3` Windows: None

See Also

`xHost, QxHost` compiler option
`x, Qx` compiler option
`ax, Qax` compiler option
`arch` compiler option
`m` compiler option

masm

Tells the compiler to generate the assembler output file using a selected dialect.

Syntax

Linux OS:

`-masm=`*`dialect`*

Windows OS:

`None`

Arguments

dialect	Is the dialect to use for the assembler output file. Possible values are:	
	`att`	Tells the compiler to generate the assembler output file using AT&T* syntax.
	`intel`	Tells the compiler to generate the assembler output file using Intel syntax.

Default

`-masm=att`	The compiler generates the assembler output file using AT&T* syntax.

Description

This option tells the compiler to generate the assembler output file using a selected dialect.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

None

mbranches-within-32B-boundaries, Qbranches-within-32B-boundaries

Tells the compiler to align branches and fused branches on 32-byte boundaries for better performance.

Syntax

Linux OS:

`-mbranches-within-32B-boundaries`

`-mno-branches-within-32B-boundaries`

Windows OS:

`/Qbranches-within-32B-boundaries`

`/Qbranches-within-32B-boundaries-`

Arguments

None

Default

`-mno-branches-within-32B-boundaries` or `/Qbranches-within-32B-boundaries-`	Branches and fused branches are not aligned on 32-byte boundaries.

Description

This option tells the compiler to align branches and fused branches on 32-byte boundaries for better performance.

> **NOTE**
> When you use this option, it may affect binary utilities usage experience, such as debugability.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

None

mintrinsic-promote, Qintrinsic-promote

Enables functions containing calls to intrinsics that require a specific CPU feature to have their target architecture automatically promoted to allow the required feature.

Syntax

Linux OS:

`-mintrinsic-promote`

Windows OS:

`/Qintrinsic-promote`

Arguments

None

Default

OFF	If this option is not specified and you call an intrinsic that requires a CPU feature not provided by the specified (or default) target processor, an error will be reported.

Description

This option enables functions containing calls to intrinsics that require a specific CPU feature to have their target architecture automatically promoted to allow the required feature.

All code within the function will be compiled with that target architecture, and the resulting code for such functions will not execute correctly on processors that do not support the required feature.

You are responsible for guarding the execution path at run time so that such functions are not dynamically reachable when the program is run on processors that do not support the required feature.

NOTE
We recommend that you use __attribute__((target(<required target>))) to mark functions that are intended to be executed on specific target architectures instead of using this option. This attribute will provide significantly better compile time error checking.

NOTE
This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

None

momit-leaf-frame-pointer

Determines whether the frame pointer is omitted or kept in leaf functions.

Syntax

Linux OS:

`-momit-leaf-frame-pointer`

`-mno-omit-leaf-frame-pointer`

Windows OS:

`None`

Arguments

None

Default

Varies	If you specify option `-fomit-frame-pointer` (or it is set by default), the default is `-momit-leaf-frame-pointer`. If you specify option `-fno-omit-frame-pointer`, the default is `-mno-omit-leaf-frame-pointer`.

Description

This option determines whether the frame pointer is omitted or kept in leaf functions. It is related to option `-f[no-]omit-frame-pointer` and the setting for that option has an effect on this option.

Consider the following option combinations:

Option Combination	Result
`-fomit-frame-pointer -momit-leaf-frame-pointer` or `-fomit-frame-pointer -mno-omit-leaf-frame-pointer`	Both combinations are the same as specifying `-fomit-frame-pointer`. Frame pointers are omitted for all routines.

Option Combination	Result
`-fno-omit-frame-pointer -momit-leaf-frame-pointer`	In this case, the frame pointer is omitted for leaf routines, but other routines will keep the frame pointer. This is the intended effect of option `-momit-leaf-frame-pointer`.
`-fno-omit-frame-pointer -mno-omit-leaf-frame-pointer`	In this case, `-mno-omit-leaf-frame-pointer` is ignored since `-fno-omit-frame-pointer` retains frame pointers in all routines . This combination is the same as specifying `-fno-omit-frame-pointer`.

This option is provided for compatibility with gcc.

NOTE
This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

Windows

Visual Studio: None

Linux

Eclipse: **Optimization > Omit frame pointer for leaf routines**

Alternate Options

None

See Also

fomit-frame-pointer, Oy compiler option

mregparm

Lets you control the number registers used to pass integer arguments.

Architecture Restrictions

Only available on IA-32 architecture

Syntax

Linux OS:

`-mregparm=`*n*

Windows OS:

`None`

Arguments

n	Specifies the number of registers to use when passing integer arguments. You can specify at most 3 registers. If you specify a nonzero value for *n*, you must build all modules, including startup modules, and all libraries, including system libraries, with the same value.

Default

OFF	The compiler does not use registers to pass arguments.

Description

Control the number registers used to pass integer arguments. This option is provided for compatibility with gcc.

NOTE
This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

None

mtune, tune

Performs optimizations for specific processors but does not cause extended instruction sets to be used (unlike -march).

Syntax

Linux OS:

`-mtune=`*`processor`*

Windows OS:

`/tune:`*`processor`*

Arguments

processor — Is the processor for which the compiler should perform optimizations. Possible values are:

`generic`	Optimizes code for the compiler's default behavior.
`alderlake` `amberlake` `broadwell` `cannonlake` `cascadelake`	Optimizes code for processors that support the specified Intel® processor or microarchitecture code name. Keywords `knl` and `silvermont` are only available on Windows and Linux systems.

`coffeelake` `cooperlake` `goldmont` `goldmont-plus` `haswell` `icelake-client` (or `icelake`) `icelake-server` `ivybridge` `kabylake` `knl` `knm` `rocketlake` `sandybridge` `sapphirerapids` `silvermont` `skylake` `skylake-avx512` `tigerlake` `tremont` `whiskeylake`	Keyword `icelake` is deprecated and may be removed in a future release.
`core-avx2`	Optimizes code for processors that support Intel® Advanced Vector Extensions 2 (Intel® AVX2), Intel® AVX, SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions.
`core-avx-i`	Optimizes code for processors that support Float-16 conversion instructions and the RDRND instruction, Intel® Advanced Vector Extensions (Intel® AVX), Intel® SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions.
`corei7-avx`	Optimizes code for processors that support Intel® Advanced Vector Extensions (Intel® AVX), Intel® SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions.
`corei7`	Optimizes code for processors that support Intel® SSE4 Efficient Accelerated String and Text Processing instructions. May also generate code for Intel® SSE4 Vectorizing Compiler and Media Accelerator, Intel® SSE3, SSE2, SSE, and SSSE3 instructions.
`atom`	Optimizes code for processors that support MOVBE instructions. May also generate code for SSSE3 instructions and Intel® SSE3, SSE2, and SSE instructions.
`core2`	Optimizes for the Intel® Core™2 processor family, including support for MMX™, Intel® SSE, SSE2, SSE3, and SSSE3 instruction sets.

`pentium-mmx`	Optimizes for Intel® Pentium® Processor with MMX technology.
`pentiumpro`	Optimizes for Intel® Pentium® Pro, Intel Pentium II, and Intel Pentium III processors.
`pentium4m`	Optimizes for Intel® Pentium® 4 processors with MMX technology.
`pentium-m` `pentium4` `pentium3` `pentium`	Optimizes code for Intel® Pentium® processors. Value `pentium3` is only available on Linux systems.

Default

generic	Code is generated for the compiler's default behavior.

Description

This option performs optimizations for specific processors but does not cause extended instruction sets to be used (unlike `-march`).

The resulting executable is backwards compatible and generated code is optimized for specific processors. For example, code generated with `-mtune=core2` or `/tune:core2` runs correctly on 4th Generation Intel® Core™ processors, but it might not run as fast as if it had been generated using `-mtune=haswell` or `/tune:haswell`. Code generated with `-mtune=haswell` (`/tune:haswell`) or `-mtune=core-avx2` (`/tune:core-avx2`) will also run correctly on Intel® Core™2 processors, but it might not run as fast as if it had been generated using `-mtune=core2` or `/tune:core2`. This is in contrast to code generated with `-march=core-avx2`, which will not run correctly on older processors such as Intel® Core™2 processors.

NOTE
This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

IDE Equivalent

Windows

Visual Studio: **Code Generation [Intel C++] >Intel Processor Microarchitecture-Specific Optimization**

Linux

Eclipse: **Code Generation > Intel Processor Microarchitecture-Specific Optimization**

OS X

Xcode: **Code Generation > Intel Processor Microarchitecture-Specific Optimization**

Alternate Options

`-mtune`	Linux: `-mcpu` (this is a deprecated option) Windows: None

See Also

`march` compiler option

regcall, Qregcall

Tells the compiler that the __regcall calling convention should be used for functions that do not directly specify a calling convention.

Syntax

Linux OS:

`-regcall`

Windows OS:

`/Qregcall`

Arguments

None

Default

OFF	The __regcall calling convention will only be used if a function explicitly specifies it.

Description

This option tells the compiler that the __regcall calling convention should be used for functions that do not directly specify a calling convention. This calling convention ensures that as many values as possible are passed or returned in registers.

It ensures that __regcall is the default calling convention for functions in the compilation, unless another calling convention is specified in a declaration.

This calling convention is ignored if it is specified for a function with variable arguments.

Note that all __regcall functions must have prototypes.

IDE Equivalent

None

Alternate Options

None

See Also

C/C++ Calling Conventions

x, Qx

Tells the compiler which processor features it may target, including which instruction sets and optimizations it may generate.

Syntax

Linux OS:

`-x`*code*

Windows OS:

`/Qx`*code*

Arguments

code	Specifies a feature set that the compiler can target, including which instruction sets and optimizations it may generate. Many of the following descriptions refer to Intel® Streaming SIMD Extensions (Intel® SSE) and Supplemental Streaming SIMD Extensions (Intel® SSSE). Possible values are:

`ALDERLAKE` `AMBERLAKE` `BROADWELL` `CANNONLAKE` `CASCADELAKE` `COFFEELAKE` `COOPERLAKE` `GOLDMONT` `GOLDMONT-PLUS` `HASWELL` `ICELAKE-CLIENT` (or `ICELAKE`) `ICELAKE-SERVER` `IVYBRIDGE` `KABYLAKE` `KNL` `KNM` `ROCKETLAKE` `SANDYBRIDGE` `SAPPHIRERAPIDS` `SILVERMONT` `SKYLAKE` `SKYLAKE-AVX512` `TIGERLAKE` `TREMONT` `WHISKEYLAKE`	May generate instructions for processors that support the specified Intel® processor or microarchitecture code name. Optimizes for the specified Intel® processor or microarchitecture code name. Keywords `KNL` and `SILVERMONT` are only available on Windows and Linux systems. Keyword `ICELAKE` is deprecated and may be removed in a future release.
`COMMON-AVX512`	May generate Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Foundation instructions, Intel® AVX-512 Conflict Detection Instructions (CDI), as well as the instructions enabled with CORE-AVX2. Optimizes for Intel® processors that support Intel® AVX-512 instructions.

`CORE-AVX512`	May generate Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Foundation instructions, Intel® AVX-512 Conflict Detection Instructions (CDI), Intel® AVX-512 Doubleword and Quadword Instructions (DQI), Intel® AVX-512 Byte and Word Instructions (BWI) and Intel® AVX-512 Vector Length Extensions (VLE), as well as the instructions enabled with CORE-AVX2. Optimizes for Intel® processors that support Intel® AVX-512 instructions.
`CORE-AVX2`	May generate Intel® Advanced Vector Extensions 2 (Intel® AVX2), Intel® AVX, SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions for Intel® processors. Optimizes for Intel® processors that support Intel® AVX2 instructions.
`CORE-AVX-I`	May generate Float-16 conversion instructions and the RDRND instruction, Intel® Advanced Vector Extensions (Intel® AVX), Intel® SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions for Intel® processors. Optimizes for Intel® processors that support Float-16 conversion instructions and the RDRND instruction.
`AVX`	May generate Intel® Advanced Vector Extensions (Intel® AVX), Intel® SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions for Intel® processors. Optimizes for Intel processors that support Intel® AVX instructions.
`SSE4.2`	May generate Intel® SSE4 Efficient Accelerated String and Text Processing instructions, Intel® SSE4 Vectorizing Compiler and Media Accelerator, and Intel® SSE3, SSE2, SSE, and SSSE3 instructions for Intel® processors. Optimizes for Intel processors that support Intel® SSE4.2 instructions.
`SSE4.1`	May generate Intel® SSE4 Vectorizing Compiler and Media Accelerator instructions for Intel® processors. May generate Intel® SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions for Intel processors that support Intel® SSE4.1 instructions.
`ATOM_SSE4.2`	May generate MOVBE instructions for Intel® processors, depending on the setting of option `-minstruction` (Linux) or `/Qinstruction` (Windows). May also generate Intel® SSE4.2, SSE3, SSE2, and SSE instructions for Intel processors. Optimizes for Intel Atom® processors that support Intel® SSE4.2 and MOVBE instructions. This keyword is only available on Windows and Linux systems.
`ATOM_SSSE3`	May generate MOVBE instructions for Intel® processors, depending on the setting of option `-minstruction` (Linux) or `/Qinstruction` (Windows). May also generate SSSE3, Intel® SSE3, SSE2, and SSE instructions for Intel processors. Optimizes for Intel Atom® processors that support Intel® SSE3 and MOVBE instructions.

	This keyword is only available on Windows and Linux systems.
`SSSE3`	May generate SSSE3 and Intel® SSE3, SSE2, and SSE instructions for Intel® processors. Optimizes for Intel processors that support SSSE3 instructions.
`SSE3`	May generate Intel® SSE3, SSE2, and SSE instructions for Intel® processors. Optimizes for Intel processors that support Intel® SSE3 instructions.
`SSE2`	May generate Intel® SSE2 and SSE instructions for Intel® processors. Optimizes for Intel processors that support Intel® SSE2 instructions. This setting is only supported for C++ (icx).

Default

varies	On Windows systems, if neither `/Qx` nor `/arch` is specified, the default target architecture supports Intel® SSE2 instructions. On Linux systems, if neither `-x` nor `-m` is specified, the default target architecture supports Intel® SSE2 instructions.

Description

This option tells the compiler which processor features it may target, including which instruction sets and optimizations it may generate. It also enables optimizations in addition to Intel feature-specific optimizations.

The specialized code generated by this option may only run on a subset of Intel® processors.

The resulting executables created from these option *code* values can only be run on Intel® processors that support the indicated instruction set.

The binaries produced by these *code* values will run on Intel® processors that support the specified features.

Do not use *code* values to create binaries that will execute on a processor that is not compatible with the targeted processor. The resulting program may fail with an illegal instruction exception or display other unexpected behavior.

Compiling the function main() with any of the *code* values produces binaries that display a fatal runtime error if they are executed on unsupported processors, including all non-Intel processors.

Compiler options `m` and `arch` produce binaries that should run on processors not made by Intel that implement the same capabilities as the corresponding Intel® processors.

The `-x` and `/Qx` options enable additional optimizations not enabled with options `-m` or `/arch`.

Linux

Options `-x` and `-m` are mutually exclusive. If both are specified, the compiler uses the last one specified and generates a warning.

Windows

Options `/Qx` and /arch are mutually exclusive. If both are specified, the compiler uses the last one specified and generates a warning.

NOTE
All settings except SSE2 do a CPU check. However, if you specify option -O0 (Linux) or option /Od (Windows), no CPU check is performed.

NOTE
This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

IDE Equivalent

Visual Studio

Visual Studio: **Code Generation > Intel Processor-Specific Optimization**

Eclipse

Eclipse: **Code Generation > Intel Processor-Specific Optimization**

Xcode

Xcode: **Code Generation > Intel Processor-Specific Optimization**

Alternate Options

None

See Also

`xHost, QxHost` compiler option
`ax, Qax` compiler option
`arch` compiler option
`march` compiler option
`m` compiler option

xHost, QxHost

Tells the compiler to generate instructions for the highest instruction set available on the compilation host processor.

Syntax

Linux OS:

`-xHost`

Windows OS:

`/QxHost`

Arguments

None

Default

varies	On Windows systems, if neither `/Qx` nor `/arch` is specified, the default target architecture supports Intel® SSE2 instructions. On Linux systems, if neither `-x` nor `-m` is specified, the default target architecture supports Intel® SSE2 instructions.

Description

This option tells the compiler to generate instructions for the highest instruction set available on the compilation host processor.

The instructions generated by this compiler option differ depending on the compilation host processor.

The following table describes the effects of specifying the `[Q]xHost` option and it tells whether the resulting executable will run on processors different from the host processor.

Descriptions in the table refer to Intel® Advanced Vector Extensions 2 (Intel® AVX2), Intel® Advanced Vector Extensions (Intel® AVX), Intel® Streaming SIMD Extensions (Intel® SSE), and Supplemental Streaming SIMD Extensions (SSSE).

Instruction Set of Host Processor	Effects When the -xHost or /QxHost Compiler Option is Specified
Intel® AVX2	When compiling on Intel® processors: Corresponds to option `[Q]xCORE-AVX2`. The generated executable will not run on non-Intel processors and it will not run on Intel® processors that do not support Intel® AVX2 instructions. When compiling on non-Intel processors: Corresponds to option `-march=core-avx2` (Linux*) or `/arch:CORE-AVX2` (Windows*). The generated executable will run on Intel® processors and non-Intel processors that support at least Intel® AVX2 instructions.. You may see a run-time error if the run-time processor does not support Intel® AVX2 instructions.
Intel® AVX	When compiling on Intel® processors: Corresponds to option `[Q]xAVX`. The generated executable will not run on non-Intel processors and it will not run on Intel® processors that do not support Intel® AVX instructions. When compiling on non-Intel processors: Corresponds to option `-mavx` (Linux) or `/arch:AVX` (Windows). The generated executable will run on Intel® processors and non-Intel processors that support at least Intel® AVX instructions. You may see a run-time error if the run-time processor does not support Intel® AVX instructions.
Intel® SSE4.2	When compiling on Intel® processors: Corresponds to option `[Q]xSSE4.2`. The generated executable will not run on non-Intel processors and it will not run on Intel® processors that do not support Intel® SSE4.2 instructions.

Instruction Set of Host Processor	Effects When the -xHost or /QxHost Compiler Option is Specified
	When compiling on non-Intel processors: Corresponds to option `–msse4.2` (Linux) or `/arch:SSE4.2` (Windows). The generated executable will run on Intel® processors and non-Intel processors that support at least Intel® SSE4.2 instructions. You may see a run-time error if the run-time processor does not support Intel® SSE4.2 instructions.
Intel® SSE4.1	When compiling on Intel® processors: Corresponds to option `[Q]xSSE4.1`. The generated executable will not run on non-Intel processors and it will not run on Intel® processors that do not support Intel® SSE4.1 instructions. When compiling on non-Intel processors: Corresponds to option `–msse4.1` (Linux) or `/arch:SSE4.1` (Windows). The generated executable will run on Intel® processors and non-Intel processors that support at least Intel® SSE4.1 instructions. You may see a run-time error if the run-time processor does not support Intel® SSE4.1 instructions.
SSSE3	When compiling on Intel® processors: Corresponds to option `[Q]xSSSE3`. The generated executable will not run on non-Intel processors and it will not run on Intel® processors that do not support SSSE3 instructions. When compiling on non-Intel processors: Corresponds to option `–mssse3` (Linux) or `/arch:SSSE3` (Windows). The generated executable will run on Intel® processors and non-Intel processors that support at least SSSE3 instructions. You may see a run-time error if the run-time processor does not support SSSE3 instructions.
Intel® SSE3	When compiling on Intel® processors: Corresponds to option `[Q]xSSE3`. The generated executable will not run on non-Intel processors and it will not run on Intel® processors that do not support Intel® SSE3 instructions. When compiling on non-Intel processors: Corresponds to option `–msse3` (Linux) or `/arch:SSE3` (Windows). The generated executable will run on Intel® processors and non-Intel processors that support at least Intel® SSE3 instructions. You may see a warning run-time error if the run-time processor does not support Intel® SSE3 instructions.
Intel® SSE2	When compiling on Intel® processors or non-Intel processors: Corresponds to option `–msse2` (Linux) or `/arch:SSE2` (Windows). The generated executable will run on Intel® processors and non-Intel processors that support at least Intel® SSE2 instructions. You may see a run-time error if the run-time processor does not support Intel® SSE2 instructions.

For more information on other settings for option `[Q]x`, see that option description.

NOTE
This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

IDE Equivalent

Visual Studio

Visual Studio: **Code Generation > Intel Processor-Specific Optimization**

Eclipse

Eclipse: **Code Generation > Intel Processor-Specific Optimization**

Alternate Options

None

See Also

`x, Qx` compiler option
`ax, Qax` compiler option
`m` compiler option
`arch` compiler option

Interprocedural Optimization Options

This section contains descriptions for compiler options that pertain to interprocedural optimization.

ipo, Qipo

Enables interprocedural optimization between files.

Syntax

Linux OS:

`-ipo[n]`

`-no-ipo`

Windows OS:

`/Qipo[n]`

`/Qipo-`

Arguments

n	Is an optional integer that specifies the number of object files the compiler should create. The integer must be greater than or equal to 0.

Default

`-no-ipo` or `/Qipo-`	Multifile interprocedural optimization is not enabled.

Description

This option enables interprocedural optimization between files. This is also called multifile interprocedural optimization (multifile IPO) or Whole Program Optimization (WPO).

When you specify this option, the compiler performs inline function expansion for calls to functions defined in separate files.

You cannot specify the names for the files that are created.

If *n* is 0, the compiler decides whether to create one or more object files based on an estimate of the size of the application. It generates one object file for small applications, and two or more object files for large applications.

If *n* is greater than 0, the compiler generates *n* object files, unless *n* exceeds the number of source files (*m*), in which case the compiler generates only *m* object files.

If you do not specify *n*, the default is 0.

NOTE
When you specify option `[Q]ipo` with option `[q or Q]opt-report`, IPO information is generated in the compiler optimization report at link time. After linking, you will see a report named ipo_out.optrpt in the folder where you linked all of the object files.

NOTE
This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

Windows

Visual Studio: **Optimization > Interprocedural Optimization**

Linux

Eclipse: **Optimization > Enable Whole Program Optimization**

Alternate Options

None

Advanced Optimization Options

This section contains descriptions for compiler options that pertain to advanced optimization. They are listed in alphabetical order.

ffreestanding, Qfreestanding

Ensures that compilation takes place in a freestanding environment.

Syntax

Linux OS:

`-ffreestanding`

Windows OS:

`/Qfreestanding`

Arguments

None

Default

OFF	Standard libraries are used during compilation.

Description

This option ensures that compilation takes place in a freestanding environment. The compiler assumes that the standard library may not exist and program startup may not necessarily be at main. This environment meets the definition of a freestanding environment as described in the C and C++ standard.

An example of an application requiring such an environment is an OS kernel.

> **NOTE**
> When you specify this option, the compiler will not assume the presence of compiler-specific libraries. It will only generate calls that appear in the source code.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

None

fjump-tables

Determines whether jump tables are generated for switch statements.

Syntax

Linux OS:

`-fjump-tables`

`-fno-jump-tables`

Windows OS:

None

Arguments

None

Default

`-fjump-tables`	The compiler may use jump tables for switch statements.

Description

This option determines whether jump tables are generated for switch statements.

Option `-fno-jump-tables` prevents the compiler from generating jump tables for switch statements. This action is performed unconditionally and independent of any generated code performance consideration.

Option `-fno-jump-tables` also prevents the compiler from creating switch statements internally as a result of optimizations.

Use `-fno-jump-tables` with `-fpic` when compiling objects that will be loaded in a way where the jump table relocation cannot be resolved.

IDE Equivalent

None

Alternate Options

None

See Also

`fpic` compiler option

fvec-peel-loops, Qvec-peel-loops

Enables peel loop vectorization.

Syntax

Linux OS:

`-fvec-peel-loops`

`-fno-vec-peel-loops`

Windows OS:

`/Qvec-peel-loops`

`/Qvec-peel-loops-`

Arguments

None

Default

`-fno-vec-peel-loops` or `/Qvec-peel-loops-`	No peel loop vectorization occurs.

Description

This option enables vectorization of peeling loops created during loop vectorization. It causes the compiler to perform additional steps to vectorize a peel loop that was created to improve alignment of memory references in the main vectorized loop.

The peel loop can be vectorized only when the masked mode of vectorization is enabled by specifying option `-fvec-with-mask` or `/Qvec-with-mask`.

The vectorization of a peel loop cannot be enforced because the compiler uses the cost model to determine whether it should be done.

IDE Equivalent

None

Alternate Options

None

See Also

`fvec-with-mask, Qvec-with-mask` compiler option

`fvec-remainder-loops, Qvec-remainder-loops` compiler option

fvec-remainder-loops, Qvec-remainder-loops

Enables remainder loop vectorization.

Syntax

Linux OS:

`-fvec-remainder-loops`

`-fno-vec-remainder-loops`

Windows OS:

`/Qvec-remainder-loops`

`/Qvec-remainder-loops-`

Arguments

None

Default

`-fno-vec-remainder-loops` or `/Qvec-remainder-loops-`	No remainder loop vectorization occurs.

Description

This option enables vectorization of remainder loops created during loop vectorization. It causes the compiler to perform additional steps to vectorize the remainder loop that was created for the vectorized main loop.

The compiler uses the cost model to determine vector factor and mode of vectorization for remainder loops.

The vectorization of remainder can be enforced using #pragma vector vecremainder on the loop.

IDE Equivalent

None

Alternate Options

None

See Also

`fvec-vec-peel-loops, Qvec-peel-loops` compiler option

`fvec-with-mask, Qvec-with-mask` compiler option
pragma vector

fvec-with-mask, Qvec-with-mask

Enables vectorization for short trip-count loops with masking.

Syntax

Linux OS:

`-fvec-with-mask`

`-fno-vec-with-mask`

Windows OS:

`/Qvec-with-mask`

`/Qvec-with-mask-`

Arguments

None

Default

`-fno-vec-with-mask` or `/Qvec-with-mask-`	No vectorization for short trip-count loops with masking occurs.

Description

This option enables a special mode of vectorization, which is applicable for loops with small number of iterations known at compile time. The peeling and remainder loops created during vectorization also fit into this category.

In this mode, the compiler uses a vector factor that is the lowest power-of-two integer greater than the known (maximum) number of loop iterations. Usually, such vectorized loops have one iteration with most of operations masked.

IDE Equivalent

None

Alternate Options

None

See Also

`fvec-vec-peel-loops, Qvec-peel-loops` compiler option
`fvec-remainder-loops, Qvec-remainder-loops` compiler option

ipp-link, Qipp-link

Controls whether the compiler links to static or dynamic threaded Intel® Integrated Performance Primitives (Intel® IPP) run-time libraries.

Syntax

Linux OS:

`-ipp-link[=lib]`

Windows OS:

`/Qipp-link[:lib]`

Arguments

lib	Specifies the Intel® IPP library to use. Possible values are:	
	`static`	Tells the compiler to link to the set of static run-time libraries.
	`dynamic`	Tells the compiler to link to the set of dynamic threaded run-time libraries.

Default

`dynamic`	The compiler links to the Intel® IPP set of dynamic run-time libraries. However, if Linux* option `-static` is specified, the compiler links to the set of static run-time libraries.

Description

This option controls whether the compiler links to static or dynamic threaded Intel® Integrated Performance Primitives (Intel® IPP) run-time libraries.

To use this option, you must also specify the `[Q]ipp` option.

NOTE
This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

IDE Equivalent

None

Alternate Options

None

See Also

`ipp, Qipp` compiler option

qactypes, Qactypes

Tells the compiler to include the Algorithmic C (AC) data type folder for header searches and link to the AC data types libraries for Field Programmable Gate Array (FPGA) and CPU compilations.

Syntax

Linux OS:

`-qactypes`

Windows OS:

`/Qactypes`

Arguments

None

Default

OFF	The compiler does not search the Algorithmic C (AC) data type folders for headers and doesn't link to AC data type libraries for FPGA and CPU compilations. As a result, AC data types cannot be used in the source program.

Description

This option tells the compiler to include the Algorithmic C (AC) data type folder when searching for headers, and to link to the AC data types libraries for Field Programmable Gate Array (FPGA) and CPU compilations.

AC data types provide support for arbitrary precision integers, fixed precision integers and arbitrary precision floating-point data types. They are built on top of the _ExtInt extended-integer type class.

When you specify option `[q or Q]actypes`, dynamic linking is the default. You cannot link to the AC data type libraries statically.

Linux

The driver must add the library names explicitly to the link command. You must use option `-qactypes` to perform the link to pull in the dependent libraries.

Windows

This option adds directives to the compiled code, which the linker then reads without further input from the driver. You do not need to specify a separate link command.

NOTE
This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

None

qdaal, Qdaal

Tells the compiler to link to certain libraries in the Intel® oneAPI Data Analytics Library (oneDAL).

Syntax

Linux OS:

`-qdaal[=lib]`

Windows OS:

`/Qdaal[:lib]`

Arguments

lib	Indicates which oneDAL library files should be linked. Possible values are:	
	`parallel`	Tells the compiler to link using the threaded oneDAL libraries. This is the default if the option is specified with no *lib*.
	`sequential`	Tells the compiler to link using the non-threaded oneDAL libraries.

Default

OFF	The compiler does not link to the oneDAL.

Description

This option tells the compiler to link to certain libraries in the Intel® oneAPI Data Analytics Library (oneDAL).

On Linux* systems, the associated oneDAL headers are included when you specify this option.

NOTE
On Windows* systems, this option adds directives to the compiled code, which the linker then reads without further input from the driver. You do not need to specify a separate link command.

On Linux* systems, the driver must add the library names explicitly to the link command. You must use option `-qdaal` to perform the link to pull in the dependent libraries.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

IDE Equivalent

Visual Studio

Visual Studio: None

Eclipse

Eclipse: **Performance Library Build Components -> Use Intel® oneAPI Data Analytics Library**

Alternate Options

Linux: `-daal` (this is a deprecated option)

See Also

Using Intel® Performance Libraries

qipp, Qipp

Tells the compiler to link to some or all of the Intel® Integrated Performance Primitives (Intel® IPP) libraries.

Syntax

Linux OS:

`-qipp[=`*lib*`]`

Windows OS:

`/Qipp[:`*lib*`]`

Arguments

lib	Indicates the Intel® IPP libraries that the compiler should link to. Possible values are:

`common`	Tells the compiler to link using the main libraries set. This is the default if the option is specified with no *lib*.
`crypto`	Tells the compiler to link using the Intel® Integrated Performance Primitives Cryptography (Intel® IPP Cryptography) libraries.
`nonpic` (Linux* only)	Tells the compiler to link using the version of the libraries that do not have position-independent code.
`nonpic_crypto` (Linux only)	Tells the compiler to link using the Intel® IPP Cryptography libraries. It uses the version of the libraries that do not have position-independent code.

Default

OFF	The compiler does not link to the Intel® IPP libraries.

Description

The option tells the compiler to link to some or all of the Intel® IPP libraries and include the Intel® IPP headers.

The `[Q]ipp-link` option controls whether the compiler links to static, dynamic threaded, or static threaded Intel® IPP runtime libraries.

NOTE
On Windows* systems, this option adds directives to the compiled code, which the linker then reads without further input from the driver. You do not need to specify a separate link command.

On Linux* systems, the driver must add the library names explicitly to the link command. You must use option `qipp` to perform the link to pull in the dependent libraries.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

IDE Equivalent

Visual Studio

Visual Studio: None

Eclipse

Eclipse: **Performance Library Build Components > Use Intel(R) Integrated Performance Primitives Libraries**

Alternate Options

Linux: `-qipp` (this is a deprecated option)

See Also

`ipp-link, Qipp-link` compiler option

qmkl, Qmkl

Tells the compiler to link to certain libraries in the Intel® oneAPI Math Kernel Library (oneMKL). On Windows systems, you must specify this option at compile time.

Syntax

Linux OS:

`-qmkl[=`*`lib`*`]`

Windows OS:

`/Qmkl[:`*`lib`*`]`

Arguments

lib	Indicates which oneMKL library files should be linked. Possible values are:	
	`parallel`	Tells the compiler to link using the threaded libraries in oneMKL. This is the default if the option is specified with no *lib*.

`sequential`	Tells the compiler to link using the sequential libraries in oneMKL.
`cluster`	Tells the compiler to link using the cluster-specific libraries and the sequential libraries in oneMKL.

Default

OFF	The compiler does not link to the oneMKL library.

Description

This option tells the compiler to link to certain libraries in the Intel® oneAPI Math Kernel Library (oneMKL).

On Linux* systems, dynamic linking is the default when you specify `-qmkl`.

On C++ systems, to link with oneMKL statically, you must specify:

```
-qmkl -static-intel
```

On Windows* systems, static linking is the default when you specify `/Qmkl`. To link with oneMKL dynamically, you must specify:

```
/Qmkl /MD
```

For more information about using oneMKL libraries, see the article titled: Intel® oneAPI Math Kernel Library Link Line Advisor.

NOTE
On Windows* systems, this option adds directives to the compiled code, which the linker then reads without further input from the driver. You do not need to specify a separate link command.

On Linux* systems, the driver must add the library names explicitly to the link command. You must use option `-qmkl` to perform the link to pull in the dependent libraries.

NOTE
If you specify option `[q or Q]mkl`, or `-qmkl=parallel` or `/Qmkl:parallel`, and you also specify option `[Q]tbb`, the compiler links to the standard threaded version of oneMKL.

However, if you specify `[q or Q]mkl`, or `-qmkl=parallel` or `/Qmkl:parallel`, and you also specify option `[Q]tbb` and option `[q or Q]openmp`, the compiler links to the OpenMP* threaded version of oneMKL.

IDE Equivalent

Visual Studio

Visual Studio: None

Eclipse

Eclipse: **Performance Library Build Components > Use Intel® oneAPI Math Kernel Library**

Alternate Options

None

See Also

`static-intel` compiler option
`MD` compiler option

qopt-assume-no-loop-carried-dep, Qopt-assume-no-loop-carried-dep

Lets you set a level of performance tuning for loops.

Syntax

Linux OS:

`-qopt-assume-no-loop-carried-dep[=n]`

Windows OS:

`/Qopt-assume-no-loop-carried-dep[=n]`

Arguments

n	Is the action for loop-carried dependencies. Possible values are:	
	`0`	The compiler does not assume there are no loop carried dependencies. This is the default if this option is not specified.
	`1`	Tells the compiler to assume there are no loop-carried dependencies for innermost loops. This is the default if the option is used but *n* is not specified.
	`2`	Tells the compiler to assume there are no loop-carried dependencies for all loop levels.

Default

`[q or Q]qopt-assume-no-loop-carried-dep=0`	The compiler does not assume there are no loop carried dependencies.

Description

This option lets you set a level of performance tuning for loops.

It is useful for C/C++ applications and benchmarks where pointers and arguments could be aliased. This is because when you specify level 1 or level 2, more loops will be vectorized or benefit from loop transformations.

This option is applied to all loops in the file. It does not apply to code outside loops.

IDE Equivalent

None

Alternate Options

None

Examples

The following loop will not be vectorized because of data dependency. Specifying `[q or Q]opt-assume-no-loop-carried-dep=1` tells the compiler to assume no data dependence will occur in this loop and it allows this loop to be vectorized:

```
void   sub   (float *A,  float *B,  int* M ) {
 for (int i =0; i< 10000 ; i++) {
    A[i]  += B[M[i]] + 1;
   }
 }
```

In the following example, this matrix multiply kernel will not be optimized because of dependency in all loop nests. Specifying `[q or Q]opt-assume-no-loop-carried-dep=2` will result in loop transformations such as blocking, unroll and jam, and vectorization:

```
void matmul(double *a, double *b,    double *c) {
 int i, j, k;
 int n = 1024;
 for (i = 0; i < 1024; i++) {
    for (j = 0; j < 1024; j++) {
       for (k = 0; k < 1024; k++) {
         c[i * n + j] += a[i * n + k] * b[k * n + j];
       }
     }
    }
 }
```

qopt-dynamic-align, Qopt-dynamic-align

Enables or disables dynamic data alignment optimizations.

Syntax

Linux OS:

`-qopt-dynamic-align`

`-qno-opt-dynamic-align`

Windows OS:

`/Qopt-dynamic-align`

`/Qopt-dynamic-align-`

Arguments

None

Default

`-qno-opt-dynamic-align` or `/Qopt-dynamic-align-`	The compiler does not generate code dynamically dependent on alignment.

Description

This option enables or disables dynamic data alignment optimizations.

If you specify `-qno-opt-dynamic-align` or `/Qopt-dynamic-align-`, the compiler generates no code dynamically dependent on alignment. It will not do any optimizations based on data location and results will depend on the data values themselves.

When you specify `[q or Q]qopt-dynamic-align`, the compiler may implement conditional optimizations based on dynamic alignment of the input data. These dynamic alignment optimizations may result in different bitwise results for aligned and unaligned data with the same values.

Dynamic alignment optimizations can improve the performance of some vectorized code, especially for long trip count loops, but there is an associated cost of increased code size and compile time. Disabling such optimizations can improve the performance of some other vectorized code. It may also improve bitwise reproducibility of results, factoring out data location from possible sources of discrepancy.

IDE Equivalent

None

Alternate Options

None

qopt-for-throughput, Qopt-for-throughput

Determines how the compiler optimizes for throughput depending on whether the program is to run in single-job or multi-job mode.

Syntax

Linux OS:

`-qopt-for-throughput[=`*`value`*`]`

Windows OS:

`/Qopt-for-throughput[:`*`value`*`]`

Arguments

value	Is one of the values "multi-job" or "single-job". If no value is specified, the default is "multi-job".

Default

OFF	If this option is not specified, the compiler will not optimize for throughput performance.

Description

This option determines whether throughput performance optimization occurs for a program that is run as a single job or one that is run in a multiple job environment.

The memory optimizations for a single job versus multiple jobs can be tuned in different ways by the compiler. For example, the cost model for loop tiling and prefetching are different for a single job versus multiple jobs. When a single job is running, more memory is available and the tunings will be different.

IDE Equivalent

None

Alternate Options

None

qopt-multiple-gather-scatter-by-shuffles, Qopt-multiple-gather-scatter-by-shuffles

Enables or disables the optimization for multiple adjacent gather/scatter type vector memory references.

Syntax

Linux OS:

`-qopt-multiple-gather-scatter-by-shuffles`

`-qno-opt-multiple-gather-scatter-by-shuffles`

Windows OS:

`/Qopt-multiple-gather-scatter-by-shuffles`

`/Qopt-multiple-gather-scatter-by-shuffles-`

Arguments

None

Default

varies	When this option is not specified, the compiler uses default heuristics for optimization.

Description

This option controls the optimization for multiple adjacent gather/scatter type vector memory references. This optimization hint is useful for performance tuning. It tries to generate more optimal software sequences using shuffles.

If you specify this option, the compiler will apply the optimization heuristics. If you specify `-qno-opt-multiple-gather-scatter-by-shuffles` or `/Qopt-multiple-gather-scatter-by-shuffles-`, the compiler will not apply the optimization.

> **NOTE**
> Optimization is affected by optimization compiler options, such as `[Q]x`, `-march` (Linux*), and `/arch` (Windows*).

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

IDE Equivalent

None

Alternate Options

None

See Also

`x, Qx` compiler option
`march` compiler option
`arch` compiler option

qopt-streaming-stores, Qopt-streaming-stores

Enables generation of streaming stores for optimization.

Syntax

Linux OS:

`-qopt-streaming-stores=`*`keyword`*

`-qno-opt-streaming-stores`

Windows OS:

`/Qopt-streaming-stores:`*`keyword`*

`/Qopt-streaming-stores-`

Arguments

keyword	Specifies whether streaming stores are generated. Possible values are:	
	`always`	Enables generation of streaming stores for optimization. The compiler optimizes under the assumption that the application is memory bound. When this option setting is specified, it is your responsibility to also insert any memory barriers (fences) as required to ensure correct memory ordering within a thread or across threads. See the Examples section for one way to do this.
	`never`	Disables generation of streaming stores for optimization. Normal stores are performed. This setting has the same effect as specifying `-qno-opt-streaming-stores` or `/Qopt-streaming-stores-`.
	`auto`	Lets the compiler decide which instructions to use.

Default

`-qopt-streaming-stores=auto` or `/Qopt-streaming-stores:auto`	The compiler decides whether to use streaming stores or normal stores.

Description

This option enables generation of streaming stores for optimization. This method stores data with instructions that use a non-temporal buffer, which minimizes memory hierarchy pollution.

This option may be useful for applications that can benefit from streaming stores.

IDE Equivalent

None

Alternate Options

None

Example

The following example shows one way to insert fences when specifying `-qopt-streaming-stores=always` or `/Qopt-streaming-stores:always`. It inserts a _mm_sfence() intrinsic call just after the loops (such as the initialization loop) where the compiler may insert streaming store instructions.

```
void simple1(double * restrict a, double * restrict b, double * restrict c, double *d, int n)
{
    int i, j;

#pragma omp parallel for
      for (j=0; j<n; j++) {
        a[j] = 1.0;
        b[j] = 2.0;
        c[j] = 0.0;
        }

      _mm_sfence(); // OR _mm_mfence();

#pragma omp parallel for
    for (i=0; i<n; i++)
        a[i] = a[i] + c[i]*b[i];
}
```

See Also

`x, Qx` compiler option

qtbb, Qtbb

Tells the compiler to link to the Intel® oneAPI Threading Building Blocks (oneTBB) libraries.

Syntax

Linux OS:

`-qtbb`

Windows OS:

`/Qtbb`

Arguments

None

Default

OFF	The compiler does not link to the oneTBB libraries.

Description

This option tells the compiler to link to the Intel® oneAPI Threading Building Blocks (oneTBB) libraries and include the oneTBB headers.

> **NOTE**
> On Windows* systems, this option adds directives to the compiled code, which the linker then reads without further input from the driver. You do not need to specify a separate link command.
>
> On Linux* systems, the driver must add the library names explicitly to the link command. You must use option `-qtbb` to perform the link to pull in the dependent libraries.

IDE Equivalent

Visual Studio

Visual Studio: None

Eclipse

Eclipse: **Performance Library Build Components > Use Intel® oneAPI Threading Building Blocks**

Alternate Options

Linux: `-tbb` (this is a deprecated option)

unroll, Qunroll

Tells the compiler the maximum number of times to unroll loops.

Syntax

Linux OS:

`-unroll[=n]`

Windows OS:

`/Qunroll[:n] (C++ only)`

Arguments

n	Is the maximum number of times a loop can be unrolled. To disable loop enrolling, specify 0.

Default

`-unroll` or `/Qunroll` (C++ only)	The compiler uses default heuristics when unrolling loops.

Description

This option tells the compiler the maximum number of times to unroll loops.

If you do not specify *n*, the optimizer determines how many times loops can be unrolled.

IDE Equivalent

Windows

Visual Studio: **Optimization > Loop Unrolling**

Linux

Eclipse: **Optimization > Loop Unroll Count**

Alternate Options

Linux: `-funroll-loops`

Windows: None

use-intel-optimized-headers, Quse-intel-optimized-headers

Determines whether the performance headers directory is added to the include path search list.

Syntax

Linux OS:

`-use-intel-optimized-headers`

Windows OS:

`/Quse-intel-optimized-headers`

Arguments

None

Default

`-no-use-intel-optimized-headers` or `/Quse-intel-optimized-headers-`	The performance headers directory is not added to the include path search list.

Description

This option determines whether the performance headers directory is added to the include path search list.

The performance headers directory is added if you specify `[Q]use-intel-optimized-headers`. Appropriate libraries are also linked in, as needed, for proper functionality.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

IDE Equivalent

Windows

Visual Studio: **Optimization > Use Intel Optimized Headers**

Linux

Eclipse: **Preprocessor > Use Intel Optimized Headers**

Alternate Options

None

See Also

Intel's valarray Implementation

vec, Qvec

Enables or disables vectorization.

Syntax

Linux OS:

`-vec`

`-no-vec`

Windows OS:

`/Qvec`

`/Qvec-`

Arguments

None

Default

`-vec` or `/Qvec`	Vectorization is enabled if option `O2` or higher is in effect.

Description

This option enables or disables vectorization.

To disable vectorization, specify `-no-vec` (Linux*) or `/Qvec-` (Windows*).

> **NOTE**
> Using this option enables vectorization at default optimization levels for both Intel® microprocessors and non-Intel microprocessors. Vectorization may call library routines that can result in additional performance gain on Intel microprocessors than on non-Intel microprocessors.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

None

vec-threshold, Qvec-threshold

Sets a threshold for the vectorization of loops.

Syntax

Linux OS:

`-vec-threshold[n]`

Windows OS:

`/Qvec-threshold[[:]n]`

Arguments

n	Is an integer whose value is the threshold for the vectorization of loops. Possible values are 0 through 100. If *n* is 0, loops get vectorized always, regardless of computation work volume. If *n* is 100, loops get vectorized when performance gains are predicted based on the compiler analysis data. Loops get vectorized only if profitable vector-level parallel execution is almost certain. The intermediate 1 to 99 values represent the percentage probability for profitable speed-up. For example, *n*=50 directs the compiler to vectorize only if there is a 50% probability of the code speeding up if executed in vector form.

Default

`-vec-threshold100` or `/Qvec-threshold100`	Loops get vectorized only if profitable vector-level parallel execution is almost certain. This is also the default if you do not specify *n*.

Description

This option sets a threshold for the vectorization of loops based on the probability of profitable execution of the vectorized loop in parallel.

This option is useful for loops whose computation work volume cannot be determined at compile-time. The threshold is usually relevant when the loop trip count is unknown at compile-time.

The compiler applies a heuristic that tries to balance the overhead of creating multiple threads versus the amount of work available to be shared amongst the threads.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

Windows

Visual Studio: **Optimization > Threshold For Vectorization**

Linux

Eclipse: **Optimization > Enable Maximum Vector-level Parallelism**

OS X

Xcode: **Optimization > Enable Maximum Vector-level Parallelism**

Alternate Options

None

Optimization Report Options

This section contains descriptions for compiler options that pertain to optimization reports.

qopt-report, Qopt-report

Enables the generation of a YAML file that includes optimization transformation information.

Syntax

Linux OS:

`-qopt-report[=arg]`

Windows OS:

`/Qopt-report[=arg]`

Arguments

arg Determines the level of detail in the report. Possible values are:

`0`	Disables generation of an optimization report. This is the default when the option is not specified.
`1 or min`	Tells the compiler to create a report with minimum details.
`2 or med`	Tells the compiler to create a report with medium details. This is the default if you do not specify *arg*.
`3 or max`	Tells the compiler to create a report with maximum details.

Levels 1, 2, and 3 (min, med, and max) include all the information of the previous level, as well as potentially some additional information.

Default

OFF	No optimization report is generated.

Description

This option enables the generation of a YAML file that includes optimization transformation information.

The YAML-formatted file provides the optimization information for the source file being compiled. For example:

```
icx -fiopenmp -qopt-report foo.c
```

This command will generate a file called foo.opt.yaml containing the optimization report messages.

Use opt-viewer.py (from llvm/tools/opt-viewer) to create html files from the YAML file. For example:

```
opt-viewer.py foo.opt.yaml
```

You can use any web-browser to open the html file to see the opt-report messages displayed inline with the original. For example:

```
Firefox html/foo.c.html source code
```

For SYCL compilations, you can also use this option to detail the variables passed to the OpenCL kernel in the optimization report. For example:

```
icpx -fsycl -qopt-report foo.cpp
```

The above command will generate a YAML-formatted optimization report that contains optimization remarks for the SYCL pass. These remarks will list the OpenCL kernel arguments generated by the compiler for the user-defined SYCL kernels in foo.cpp. The remarks will also provide additional information like name, type, and size for the OpenCL kernel arguments.

You can then use opt-viewer.py to create html files from the YAML file, and use any web-browser to open the html file to see the opt-report remarks

Note that the YAML file is used to drive the community llvm-opt-report tool.

NOTE
This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

None

See Also

qopt-report-file, Qopt-report-file compiler option

qopt-report-file, Qopt-report-file

Specifies whether the output for the generated optimization report goes to a file, stderr, or stdout.

Syntax

Linux OS:

`-qopt-report-file=`*keyword*

Windows OS:

`/Qopt-report-file:`*keyword*

Arguments

keyword	Specifies where the output for the report goes. You can specify one of the following:	
	`filename`	Specifies the name of the file where the generated report should go.
	`stderr`	Indicates that the generated report should go to stderr.

`stdout`	Indicates that the generated report should go to stdout. This setting can also be specified as `-qopt-report-stdout` (Linux) or `/Qopt-report-stdout` (Windows).

Default

OFF	No optimization report is generated.

Description

This option specifies whether the output for the generated optimization report goes to a file, stderr, or stdout.

If you use this option, you do not have to specify option `[q or Q]opt-report`.

NOTE
This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

Visual Studio

Visual Studio: **Diagnostics > Optimization Diagnostic File**

Diagnostics > Emit Optimization Diagnostic to File

Eclipse

Eclipse: **Compilation Diagnostics > Emit Optimization Diagnostics to File**

Compilation Diagnostics > Optimization Diagnostics File

Alternate Options

None

See Also

`qopt-report, Qopt-report` compiler option

Offload Compilation, OpenMP*, and Parallel Processing Options

This section contains descriptions for compiler options that pertain to offload compilation, OpenMP*, or parallel processing. They are listed in alphabetical order.

device-math-lib

Enables or disables certain device libraries. This is a deprecated option that may be removed in a future release.

Syntax

Linux OS:

`-device-math-lib=`*`library`*

`-no-device-math-lib=`*`library`*

Windows OS:

`/device-math-lib:`*`library`*

`/no-device-math-lib:`*`library`*

Arguments

`library`	Possible values are:	
	`fp32`	Links the fp32 device math library.
	`fp64`	Links the fp64 device math library.
	To link more than one library, include a comma between the library names.	
	For example, if you want to link both the fp32 and fp64 device libraries, specify: fp32, fp64	

Default

fp32, fp64	Both the fp32 and fp64 device libraries are linked.

Description

This option enables or disables certain device libraries.

This is a deprecated option that may be removed in a future release. There is no replacement option.

IDE Equivalent

None

Alternate Options

None

See Also

fopenmp-device-lib compiler option

fintelfpga

Lets you perform ahead-of-time (AOT) compilation for the Field Programmable Gate Array (FPGA).

Syntax

Linux OS:

`-fintelfpga`

Windows OS:

`-fintelfpga`

Arguments

None

Default

OFF	The ahead-of-time (AOT) compilation is not performed.

Description

This option lets you perform ahead-of-time (AOT) compilation for the FPGA.

It is functionally equivalent to specifying the following, which is compiled with dependency and debug information enabled:

```
-fsycl-targets=spir64-unknown-unknown
```

IDE Equivalent

Visual Studio

Visual Studio: **DPC++ > General > Enable FPGA workflows**

Eclipse

Eclipse: **Intel(R) oneAPI DPC++ Compiler > General > Enable FPGA workflows**

Alternate Options

None

See Also

`fsycl-targets` compiler option
`fsycl-link` compiler option
`Xs` compiler option

fiopenmp, Qiopenmp

Enables recognition of OpenMP features, such as parallel, simd, and offloading directives. This is an alternate option for compiler option [Q or q]openmp.*

Syntax

Linux OS:

`-fiopenmp`

Windows OS:

`/Qiopenmp`

Arguments

None

Default

OFF	If this option is not specified, OpenMP* features are not transformed in LLVM*.

Description

This option enables recognition of OpenMP* features, such as parallel, simd, and offloading directives. This is an alternate option for compiler option `[Q or q]openmp`.

The `-fiopenmp` and `/Qiopenmp` options enable Intel's implementation of OpenMP* in the compiler back end. The compiler front end produces an intermediate representation that preserves the parallelism exposed by OpenMP* directives. The back end uses the exposed parallelism to do more advanced optimizations, such as SIMD vectorization.

NOTE
To enable offloading to a specified GPU target, you must also specify option `fopenmp-targets` (Linux*) or `/Qopenmp-targets` (Windows).

NOTE
Option `-fopenmp` is *not* the same as option `-fiopenmp`. Option `-fopenmp` will *not* do offloading.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

IDE Equivalent

Windows

Visual Studio: **DPC++ > Language > OpenMP Support**

C/C++ > Language [Intel C++] > OpenMP Support

Intel(R) oneAPI DPC++ Compiler > Language > OpenMP Support

Intel C++ Compiler > Language > OpenMP Support

Linux

Eclipse: **Intel(R) oneAPI DPC++ Compiler > Language > OpenMP Support**

Intel C++ Compiler > Language > OpenMP Support

Alternate Options

Linux: `-qopenmp`

Windows: `/Qopenmp`

See Also

`qopenmp, Qopenmp` compiler option

`fopenmp-targets, Qopenmp-targets` compiler option

fno-sycl-libspirv

Disables the check for libspirv (the SPIR-V tools library).*

Syntax

Linux OS:

`-fno-sycl-libspirv`

Windows OS:

`-fno-sycl-libspirv`

Arguments

None

Default

OFF	The check for libspirv is enabled.

Description

This option disables the check for libspirv (the SPIR-V* tools library).

> **NOTE**
> When SYCL offloading is enabled, this option only applies to device-specific compilation.

IDE Equivalent

None

Alternate Options

None

foffload-static-lib

Tells the compiler to link with a fat (multi-architecture) static library. This is a deprecated option that may be removed in a future release.

Syntax

Linux OS:

`-foffload-static-lib=`*`file`*

Windows OS:

`-foffload-static-lib=`*`file`*

Arguments

file	Is the name of the fat static library to use. It can include the path where the library is located.

Default

OFF	No linking occurs to a fat static library.

Description

This option tells the compiler to link with a fat (multi-architecture) static library.

The filename specified is treated as a "fat" static library of device code - an archive of fat objects. When linking, the compiler will extract the device code from the objects contained in the library and link it with other device objects coming from the individual fat objects passed on the command line.

NOTE
If you try to pass libraries by using compiler option l, there can be dynamic libraries and partial linking with dynamic libraries, which may lead to a crash.

IDE Equivalent

None

Alternate Options

None

fopenmp

Option -fopenmp is a deprecated option that will be removed in a future release.

Syntax

Linux OS:

`-fopenmp`

Windows OS:

`None`

Arguments

None

Default

OFF	No OpenMP* multi-threaded code is generated by the compiler.

Description

Enables recognition of OpenMP* features and tells the parallelizer to generate multi-threaded code based on OpenMP* directives.

This option is meant for advanced users who prefer to use OpenMP* as it is implemented by the LLVM community. You can get most of that functionality by using this option and option `-fopenmp-simd`.

Option `-fopenmp` is a deprecated option that will be removed in a future release. For most users, we recommend that you instead use option qopenmp, Qopenmp.

NOTE
Option `-fopenmp` is not the same as option `-fiopenmp`. If you want to get full advantage of SIMD vectorization or offloading, you must use option `-qopenmp`.

NOTE
This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

None

fopenmp-declare-target-scalar-defaultmap, Qopenmp-declare-target-scalar-defaultmap

Determines which implicit data-mapping/sharing rules are applied for a scalar variable referenced in a target pragma.

Syntax

Linux OS:

`-fopenmp-declare-target-scalar-defaultmap=`*`keyword`*

Windows OS:

`/Qopenmp-declare-target-scalar-defaultmap:`*`keyword`*

Arguments

keyword	Is the rule to be applied for a scalar variable referenced in a target pragma.TARGET directive.. Possible values are:	
	`default`	Specifies that the compiler should apply implicit data-mapping/sharing rules according to the OpenMP* specification. Thus, if a scalar variable referenced in a target construct appears in a `to` or `link` clause in a `declare target` pragma that does not have a `device_type` (nohost) clause, and the target construct's clauses do not define explicit data-mapping/sharing rules for this variable, then the compiler should treat it as if it had appeared in a `map` clause with a map-type of `tofrom`.
	`firstprivate`	Specifies that when a scalar variable referenced in a target construct appears in a `to` or `link` clause in a `declare target` pragma that does not have a `device_type` (nohost) clause, and the target construct's clauses do not define explicit data-mapping/sharing rules for this variable, then the scalar variable should not be mapped, but instead it has an implicit data-sharing attribute of `firstprivate`.

Default

`-fopenmp-declare-target-scalar-defaultmap=default` or `/Qopenmp-declare-target-scalar-defaultmap:default`	The compiler applies implicit data-mapping/sharing rules according to OpenMP specification.

Description

This option determines which implicit data-mapping/sharing rules are applied for a scalar variable referenced in a target pragma, when that scalar variable appears in a `declare target` pragma that has a `to` or `link` clause, but not clause `device_type` (nohost).

It tells the compiler to assume that a scalar `declare target` variable with implicit data-mapping/sharing referenced in a target construct has the same value before the target construct (in the host environment) and at the beginning the target region (in the device environment). This may enable some optimizations in the host code invoking the target region for execution.

The option only affects data-mapping/sharing rules for scalar variables referenced in a target construct that do not appear in one of the target clauses `map`, `is_device_ptr`, or `has_device_addr`.

For more information about implicit data-mapping/sharing rules, see the OpenMP 5.2 specification. For example, see section 5.8.1 in that specification.

IDE Equivalent

None

Alternate Options

None

Examples

Consider the following:

```
#pragma omp declare target
int N;
#pragma omp end declare target
...
void program() {
#pragma omp target teams distribute parallel for
  for (int i = 0; i < N; ++i) ...
}
```

Specifying `-fopenmp-declare-target-scalar-defaultmap=firstprivate` (or `/Qopenmp-declare-target-scalar-defaultmap:firstprivate`) or an explicit 'firstprivate(N)' lets the compiler generate efficient host code that issues the most appropriate number of teams and threads to execute the iterations of the distribute parallel for loop, assuming that N does not change its value between the beginning of the target region and the beginning of the distribute parallel for region.

If the compiler option (or 'firstprivate(N)') is not used, then the value of N in the host code (before the target construct) may be different from the value of N in the for statement. To compute the right number of teams/threads on the host the value of N must be transferred from the device to the host, which may result in a performance penalty.

The option may not behave correctly for all OpenMP programs. In particular, it may behave incorrectly for programs that allow different values of the same declare target scalar variables on entry to target regions.

For example, consider the following:

```
#include <stdio.h>

#pragma omp declare target
int x = 0; /* host 'x' is 0, target 'x' is 0 */
#pragma omp end declare target

int main() {
  x = -1;

  /* host 'x' is -1, target 'x' is 0 */
#pragma omp target
  x = 1;

  /* host 'x' is -1, target 'x' is 1 */
```

```
#pragma omp target
  printf("target: %d == 1\n", x);

#pragma omp target update from(x)

  /* host 'x' is 1, target 'x' is 1 */
  printf("host: %d == 1\n", x);

  return 0;
}
```

The following is the correct output for the above code:

```
target: 1 == 1
host: 1 == 1
```

However, this is the output when option `-fopenmp-declare-target-scalar-defaultmap=firstprivate` (or `/Qopenmp-declare-target-scalar-defaultmap:firstprivate`) is specified:

```
target: -1 == 1
host: 0 == 1
```

fopenmp-device-lib

Enables or disables certain device libraries for an OpenMP target.*

Syntax

Linux OS:

`-fopenmp-device-lib=`*`library`*`[,`*`library`*`,...]`

`-fno-openmp-device-lib=`*`library`*`[,`*`library`*`,...]`

Windows OS:

`-fopenmp-device-lib=`*`library`*`[,`*`library`*`,...]`

`-fno-openmp-device-lib=`*`library`*`[,`*`library`*`,...]`

Arguments

`library`

Possible values are:

`libm-fp32`	Enables linking to the fp32 device math library.
`libm-fp64`	Enables linking to the fp64 device math library.
`libc`	Enables linking to the C library.
`all`	Enables linking to libraries libm-fp32, libm-fp-64, and libc.

To link more than one library, include a comma between the library names. For example, if you want to link both the libm-fp32 device library and the C library, specify: libm-fp32,libc.

Do not add spaces between library names.

Note that if you specify "all", it supersedes any additional value you may specify.

Default

OFF	Disables linking to device libraries for this target.

Description

This option enables or disables certain device libraries for an OpenMP* target.

If you specify `fno-openmp-device-lib=library`, linking to the specified library is disabled for the OpenMP* target.

> **NOTE**
> When OpenMP* offloading is enabled, this option only applies to device-specific compilation.

IDE Equivalent

Windows

Visual Studio: **Linker > General > Enable linking of the device libraries for OpenMP offload**

Linker > General > Disable linking of the device libraries for OpenMP offload

Linux

Eclipse: **Linker > Libraries > Enable linking of the device libraries for OpenMP offload**

Linker > Libraries > Disable linking of the device libraries for OpenMP offload

Alternate Options

None

fopenmp-target-buffers, Qopenmp-target-buffers

Enables a way to overcome the problem where some OpenMP offload SPIR-V* devices produce incorrect code when a target object is larger than 4GB.*

Syntax

Linux OS:

`-fopenmp-target-buffers=`*`keyword`*

Windows OS:

`/Qopenmp-target-buffers:`*`keyword`*

Arguments

`keyword` Possible values are:

`default`	Tells the compiler to use default heuristics. This may produce incorrect code on some OpenMP* offload SPIR-V* devices when a target object is larger than 4GB.

`4GB`	Tells the compiler to generate code to prevent the issue described by `default`. OpenMP* offload programs that access target objects of size larger than 4GB in target code require this option. This setting applies to the following: • Target objects declared in OpenMP* target regions or inside OpenMP* declare target functions • Target objects that exist in the OpenMP* device data environment • Objects that are mapped and/or allocated by means of OpenMP* APIs (such as omp_target_alloc)

Default

`default`	If you do not specify this option, the compiler may produce incorrect code on some OpenMP* offload SPIR-V* devices when a target object is larger than 4GB.

Description

This option enables a way to overcome the problem where some OpenMP* offload SPIR-V* devices produce incorrect code when a target object is larger than 4GB (4294959104 bytes).

However, note that when `-fopenmp-target-buffers=4GB` (or `/Qopenmp-target-buffers:4GB`) is specified on Intel® GPUs, there may be a decrease in performance.

To use this option, you must also specify option `-fopenmp-targets` (Linux*) or `/Qopenmp-targets` (Windows*).

NOTE
This option may have no effect for some OpenMP* offload SPIR-V* devices, and for OpenMP* offload targets different from SPIR*.

NOTE
When OpenMP* offloading is enabled, this option only applies to device-specific compilation.

IDE Equivalent

Windows

Visual Studio: **DPC++ > Language > Specify buffer size for OpenMP offload kernel access limitations** (DPC++)

Windows

Visual Studio: **C/C++ > Language [Intel C++] > Specify buffer size for OpenMP offload kernel access limitations** (C++)

Linux

Eclipse: **Intel(R) oneAPI DPC++ Compiler > Language > Specify buffer size for OpenMP offload kernel access limitations** (DPC++)

Linux

Eclipse: **Intel C++ Compiler > Language > Specify buffer size for OpenMP offload kernel access limitations** (C++)

Alternate Options

None

See Also

`fopenmp-targets, Qopenmp-targets` compiler option

fopenmp-targets, Qopenmp-targets

Enables offloading to a specified GPU target if OpenMP features have been enabled.*

Syntax

Linux OS:

`-fopenmp-targets=`*`keyword`*

Windows OS:

`/Qopenmp-targets:`*`keyword`*

Arguments

keyword	The only supported value for this argument is `spir64`. When you specify `spir64`, the compiler generates an x86 + SPIR64 (64-bit Standard Portable Intermediate Representation) fat binary for Intel® GPU devices.

Default

OFF	If this option is not specified, no x86 + SPIR64 fat binary is created.

Description

This option enables offloading to a specified GPU target if OpenMP* features have been enabled.

To use this option, you must enable recognition of OpenMP* features by specifying one of the following options:

Linux

- `-qopenmp`
- `-fiopenmp`
- `-fopenmp` (deprecated; it is equivalent to `-qopenmp`)

Windows

- `/Qopenmp`
- `/Qiopenmp`

The following shows an example:

```
icx (or icpx) -fiopenmp -fopenmp-targets=spir64 matmul_offload.cpp -o matmul
```

When you specify `-fopenmp-targets` or `/Qopenmp-targets`, C++ exception handling is disabled for target compilations.

Linux

For host compilations, if you want to disable C++ exception handling, you must specify option `-fno-exceptions`.

NOTE
When OpenMP* offloading is enabled, this option only applies to device-specific compilation.

IDE Equivalent

Windows

Visual Studio: **DPC++ > Language > Enable OpenMP Offloading**

C/C++ > Language [Intel C++] > Enable OpenMP Offloading

Intel(R) oneAPI DPC++ Compiler > Language > Enable OpenMP Offloading

Intel C++ Compiler > Language > Enable OpenMP Offloading

Linux

Eclipse: **Intel(R) oneAPI DPC++ Compiler > Language > Enable OpenMP Offloading**

Intel C++ Compiler > Language > Enable OpenMP Offloading

Alternate Options

None

See Also

`fiopenmp, Qiopenmp` compiler option
`qopenmp, Qopenmp` compiler option

fsycl

Enables a program to be compiled as a SYCL program rather than as plain C++11 program.

Syntax

Linux OS:

`-fsycl`

Windows OS:

`-fsycl`

Arguments

None

Default

SYCL: ON	A C++ program is compiled as a SYCL program.
C++: OFF	A C++ program is compiled as a C++11 program.

Description

This option enables a program to be compiled as a SYCL program rather than as plain C++11 program.

IDE Equivalent

None

Alternate Options

None

See Also

`fsycl-targets` compiler option

fsycl-add-targets

Lets you add arbitrary device binary images to the fat SYCL binary when linking. This is a deprecated option that may be removed in a future release.*

Syntax

Linux OS:

`-fsycl-add-targets=`*`T1`*`:`*`file1`*`,...,`*`Tn`*`:`*`filen`*

Windows OS:

`-fsycl-add-targets=`*`T1`*`:`*`file1`*`,...,`*`Tn`*`:`*`filen`*

Arguments

T	Is a target triple for the device binary image.
file	Is the location of the device binary image.

You can specify one or more pair of *T*:*file*.

Default

OFF	Arbitrary device images are not added to any fat SYCL* binary being linked.

Description

This option lets you add arbitrary device binary images to the fat SYCL* binary when linking.

NOTE
When using the icx/icpx compiler driver for this option, you must also specify option `-fsycl`.

When using the dpcpp compiler driver, option `-fsycl` is implied by default.

NOTE
When SYCL offloading is enabled, this option only applies to device-specific compilation.

IDE Equivalent

None

Alternate Options

None

See Also

`fsycl-link-targets` compiler option

fsycl-dead-args-optimization

Enables elimination of SYCL dead kernel arguments.

Syntax

Linux OS:

`-fsycl-dead-args-optimization`

`-fno-sycl-dead-args-optimization`

Windows OS:

`-fsycl-dead-args-optimization`

`-fno-sycl-dead-args-optimization`

Arguments

None

Default

OFF	SYCL dead kernel arguments are not eliminated. This default may change in the future.

Description

This option enables elimination of SYCL dead kernel arguments. This optimization can improve performance.

If you specify `-fno-sycl-dead-args-optimization`, this optimization is disabled.

> **NOTE**
> When using the icx/icpx compiler driver for option `-fsycl-dead-args-optimization`, you must also specify option `-fsycl`.
>
> When using the dpcpp compiler driver, option `-fsycl` is implied by default.

> **NOTE**
> When SYCL offloading is enabled, this option only applies to device-specific compilation.

IDE Equivalent

None

Alternate Options

None

fsycl-device-code-split

Specifies a SYCL device code module assembly.*

Syntax

Linux OS:

`-fsycl-device-code-split[=`*`value`*`]`

Windows OS:

`-fsycl-device-code-split[=value]`

Arguments

value	Can be only one of the following:	
	`per_kernel`	Creates a separate device code module for each SYCL* kernel. Each device code module will contain a kernel and all its dependencies, such as called functions and used variables.
	`per_source`	Creates a separate device code module for each source (translation unit). Each device code module will contain a collection of kernels grouped on per-source basis and all their dependencies, such as all used variables and called functions, including the SYCL_EXTERNAL macro-marked functions from other translation units.
	`off`	Creates a single module for all kernels.
	`auto`	The compiler will use a heuristic to select the best way of splitting device code. This is the same as specifying `fsycl-device-code-split` with no value.

Default

auto	This is the default whether you do not specify the compiler option or you do specify the compiler option, but omit a value. The compiler will use a heuristic to select the best way of splitting device code.

Description

This option specifies a SYCL* device code module assembly.

NOTE
When using the icx/icpx compiler driver for this option, you must also specify option `-fsycl`.

When using the dpcpp compiler driver, option `-fsycl` is implied by default.

NOTE
When SYCL offloading is enabled, this option only applies to device-specific compilation.

IDE Equivalent

None

Alternate Options

None

fsycl-device-lib

Enables or disables certain device libraries for a SYCL target.*

Syntax

Linux OS:

`-fsycl-device-lib=`*`library`*`[,`*`library`*`,...]`

`-fno-sycl-device-lib=`*`library`*`[,`*`library`*`,...]`

Windows OS:

`-fsycl-device-lib=`*`library`*`[,`*`library`*`,...]`

`-fsycl-device-lib=`*`library`*`[,`*`library`*`,...]`

Arguments

library

Possible values are:

`libm-fp32`	Enables linking to the fp32 device math library.
`libm-fp64`	Enables linking to the fp64 device math library.
`libc`	Enables linking to the C library.
`all`	Enables linking to libraries libm-fp32, libm-fp-64, and libc.

To link more than one library, include a comma between the library names. For example, if you want to link both the libm-fp32 device library and the C library, specify: libm-fp32,libc.

Do not add spaces between library names.

Note that if you specify "all", it supersedes any additional value you may specify.

Default

OFF	Disables linking to device libraries for this target.

Description

This option enables or disables certain device libraries for a SYCL* target.

If you specify `fno-sycl-device-lib=library`, linking to the specified library is disabled for the SYCL* target.

NOTE
When using the icx/icpx compiler driver for option `-fsycl-device-lib`, you must also specify option `-fsycl`.

When using the dpcpp compiler driver, option `-fsycl` is implied by default.

NOTE
When SYCL offloading is enabled, this option only applies to device-specific compilation.

IDE Equivalent

Windows

Visual Studio: **Linker > General > Enable linking of the device libraries**

Linker > General > Disable linking of the device libraries

Linux

Eclipse: **Linker > Libraries > Enable linking of the device libraries**

Linker > Libraries > Disable linking of the device libraries

Alternate Options

None

fsycl-device-only

Tells the compiler to generate a device-only binary.

Syntax

Linux OS:

`-fsycl-device-only`

Windows OS:

`-fsycl-device-only`

Arguments

None

Default

OFF	No device-only binary is generated.

Description

This option tells the compiler to generate a device-only binary.

NOTE
When SYCL offloading is enabled, this option only applies to device-specific compilation.

IDE Equivalent

None

Alternate Options

None

fsycl-early-optimizations

Enables LLVM-related optimizations before SPIR-V generation.*

Syntax

Linux OS:

`-fsycl-early-optimizations`

`-fno-sycl-early-optimizations`

Windows OS:

`-fsycl-early-optimizations`

`-fno-sycl-early-optimizations`

Arguments

None

Default

ON	LLVM-related optimizations are enabled before SPIR-V* generation.

Description

This option enables LLVM-related optimizations before SPIR-V* generation. These optimizations can improve performance.

If you specify `-fno-sycl-early-optimizations`, these optimizations are disabled.

NOTE
When using the icx/icpx compiler driver for option `-fsycl-early-optimizations`, you must also specify option `-fsycl`.

When using the dpcpp compiler driver, option `-fsycl` is implied by default.

NOTE
When SYCL offloading is enabled, this option only applies to device-specific compilation.

IDE Equivalent

Visual Studio

Visual Studio: **DPC++ > Optimization > Enable/Disable DPC++ early optimization before generation of SPIR-V code**

Eclipse

Eclipse: **Intel(R) oneAPI DPC++ Compiler > Optimization > Enable/Disable DPC++ early optimization before generation of SPIR-V code**

Alternate Options

None

fsycl-enable-function-pointers

Enables function pointers and support for virtual functions for SYCL kernels and device functions. This is an experimental feature.

Syntax

Linux OS:

`-fsycl-enable-function-pointers`

Windows OS:

`-fsycl-enable-function-pointers`

Arguments

None

Default

OFF	Function pointers are not enabled and virtual functions for SYCL kernels and device functions are not supported.

Description

This option enables function pointers and support for virtual functions for SYCL kernels and device functions. This is an experimental feature.

This enhanced support is limited to CPU-device only and cannot currently be used for GPU devices.

NOTE
When using the icx/icpx compiler driver for this option, you must also specify option `-fsycl`.

When using the dpcpp compiler driver, option `-fsycl` is implied by default.

NOTE
When SYCL offloading is enabled, this option only applies to device-specific compilation.

IDE Equivalent

None

Alternate Options

None

fsycl-esimd-force-stateless-mem

Determines whether the compiler enforces stateless memory accesses within ESIMD kernels on the target device. This is an experimental feature.

Syntax

Linux OS:

`-fsycl-esimd-force-stateless-mem`

`-fno-sycl-esimd-force-stateless-mem`

Windows OS:

`-fsycl-esimd-force-stateless-mem`

`-fno-sycl-esimd-force-stateless-mem`

Arguments

None

Default

OFF	Memory accesses that are stateful are not converted to stateless.

Description

This option determines whether the compiler enforces stateless memory accesses within ESIMD kernels on the target device. This is an experimental feature.

Option `-fsycl-esimd-force-stateless-mem` uses SYCL* accessors to convert stateful memory to stateless memory. SIMD intrinsics that cannot be automatically converted are disabled and reported during the compilation phase.

In cases where a target does not support stateful accesses, option `-fsycl-esimd-force-stateless-mem` may be helpful to avoid issues caused by the 4Gb-per-surface limitation in programs written with SYCL accessors.

If you specify `-fno-sycl-esimd-force-stateless-mem`, the compiler does not enforce stateless memory accesses.

NOTE
When using the icx/icpx compiler driver for option `-fsycl-esimd-force-stateless-mem`, you must also specify option `-fsycl`.

When using the dpcpp compiler driver, option `-fsycl` is implied by default.

NOTE
When SYCL offloading is enabled, this option only applies to device-specific compilation.

IDE Equivalent

None

Alternate Options

None

fsycl-explicit-simd

Enables or disables the experimental "Explicit SIMD" SYCL extension. This is a deprecated option that may be removed in a future release.*

Syntax

Linux OS:

`-fsycl-explicit-simd`

`-fno-sycl-explicit-simd`

Windows OS:

`-fsycl-explicit-simd`

`-fno-sycl-explicit-simd`

Arguments

None

Default

`-fno-sycl-explicit-simd`	The explicit SIMD SYCL* extension is disabled.

Description

This option enables or disables the experimental "Explicit SIMD" SYCL* extension.

If you specify option `-fsycl-explicit-simd`, it enables the experimental "Explicit SIMD" SYCL* extension for lower-level Intel GPU programming. It allows you to write explicitly vectorized device code. Note that APIs for this feature may change in the future.

NOTE
When using the icx/icpx compiler driver for option `-fsycl-explicit-simd`, you must also specify option `-fsycl`.

When using the dpcpp compiler driver, option `-fsycl` is implied by default.

NOTE
When SYCL offloading is enabled, this option only applies to device-specific compilation.

IDE Equivalent

None

Alternate Options

None

See Also

Explicit SIMD SYCL* Extension

fsycl-help

Causes help information to be emitted from the device compiler backend.

Syntax

Linux OS:

`-fsycl-help[=arg]`

Windows OS:

`-fsycl-help[=arg]`

Arguments

arg	Can be one of "x86_64", "fpga", "gen", or "all". Option `-fsycl-help=all` outputs help for "x86_64", "fpga", and "gen". Specifying "all" is the same as specifying `fsycl-help` with no *arg*.

Default

OFF	No help information is emitted from the device compiler backend.

Description

This option causes help information to be emitted from the device compiler backend.

> **NOTE**
> When SYCL offloading is enabled, this option only applies to device-specific compilation.

IDE Equivalent

None

Alternate Options

None

fsycl-host-compiler

Tells the compiler to use the specified compiler for the host compilation of the overall offloading compilation that is performed.

Syntax

Linux OS:

`-fsycl-host-compiler=arg`

Windows OS:

`-fsycl-host-compiler=arg`

Arguments

arg	Is the compiler that will be the host for compilation. It can be the name of a compiler or the specific path to the compiler.

Default

OFF	The host compilation will be performed by the Intel® DPC++ Compiler.

Description

This option tells the compiler to use the specified compiler for the host compilation of the overall offloading compilation that is performed.

> **NOTE**
> When using the icx/icpx compiler driver for this option, you must also specify option `-fsycl`.
>
> When using the dpcpp compiler driver, option `-fsycl` is implied by default.

IDE Equivalent

None

Alternate Options

None

Example

Consider the following:

```
-fsycl-host-compiler=g++            // the compiler looks for g++ in the current path
-fsycl-host-compiler=/usr/bin/g++    // the compiler looks for g++ in the explicit path
```

See Also

`fsycl` compiler option

`fsycl-host-compiler-options` compiler option

fsycl-host-compiler-options

Passes options to the compiler specified by option *f*`sycl-host-compiler`*.*

Syntax

Linux OS:

`-fsycl-host-compiler-options="`*opts*`"`

Windows OS:

`-fsycl-host-compiler-options="`*opts*`"`

Arguments

opts	Is a string of compatible compiler options to be passed. The string must appear within quotes. If there is more than one compiler option, a space must appear between each option name.

Default

OFF	No options are passed to the compiler specified by `-fsycl-host-compiler`.

Description

This option tells the compiler to pass options to the compiler specified by option `fsycl-host-compiler`. The options must be compatible with the compiler specified by `fsycl-host-compiler`.

> **NOTE**
> Specifying any kind of phase limiting options (such as `-c`, `-E`, or `-S`) may interfere with the expected output set during the host compilation. This can cause undefined behavior.

> **NOTE**
> When using the icx/icpx compiler driver for this option, you must also specify option `-fsycl`.
>
> When using the dpcpp compiler driver, option `-fsycl` is implied by default.

IDE Equivalent

None

Alternate Options

None

See Also

`fsycl-host-compiler` compiler option

fsycl-id-queries-fit-in-int

Tells the compiler to assume that SYCL ID queries fit within MAX_INT.

Syntax

Linux OS:

`-fsycl-id-queries-fit-in-int`

`-fno-sycl-id-queries-fit-in-int`

Windows OS:

`-fsycl-id-queries-fit-in-int`

`-fno-sycl-id-queries-fit-in-int`

Arguments

None

Default

ON	The compiler assumes that SYCL ID queries fit within MAX_INT.

Description

This option tells the compiler to assume that SYCL ID queries fit within MAX_INT. It assumes that the following values fit within MAX_INT:

- id class get() member function and operator[]
- item class get_id() member function and operator[]
- nd_item class get_global_id()/get_global_linear_id() member functions

For more information about these values, see the Khronos* Group SYCL* 1.2.1 Specification.

If you need to use a larger number of work items, use the OFF setting for this option, which is `-fno-sycl-id-queries-fit-in-int`.

Caution
You should carefully evaluate whether you should use the OFF setting when you have a larger number of work items. Truncating to data type int is often incorrect in such circumstances. If the OFF setting is used when the values fit within MAX_INT, it can lead to unexpected performance issues.

NOTE
When using the icx/icpx compiler driver for option `-fsycl-id-queries-fit-in-int`, you must also specify option `-fsycl`.

When using the dpcpp compiler driver, option `-fsycl` is implied by default.

NOTE
When SYCL offloading is enabled, this option only applies to device-specific compilation.

IDE Equivalent

None

Alternate Options

None

fsycl-instrument-device-code

Enables or disables linking of the Instrumentation and Tracing Technology (ITT) device libraries for VTune™.

Syntax

Linux OS:

`-fsycl-instrument-device-code-split`

`-fno-sycl-instrument-device-code-split`

Windows OS:

`-fsycl-instrument-device-code-split`

`-fno-sycl-instrument-device-code-split`

Arguments

None

Default

ON	The device libraries needed for Instrumentation and Tracing Technology (ITT) are enabled.

Description

This option enables or disables linking of the Instrumentation and Tracing Technology (ITT) device libraries for VTune™. This provides annotations to intercept various events inside kernels generated by Just in Time (JIT) compilation.

If you specify `-fno-sycl-instrument-device-code-split`, no linking occurs to the Instrumentation and Tracing Technology (ITT) device libraries.

> **NOTE**
> When using the icx/icpx compiler driver for option `-fsycl-instrument-device-code-split`, you must also specify option `-fsycl`.
>
> When using the dpcpp compiler driver, option `-fsycl` is implied by default.

> **NOTE**
> When SYCL offloading is enabled, this option only applies to device-specific compilation.

IDE Equivalent

None

Alternate Options

None

fsycl-link

Tells the compiler to perform a partial link of device binaries to be used with Field Programmable Gate Array (FPGA).

Syntax

Linux OS:

`-fsycl-link[=`*value*`]`

Windows OS:

`-fsycl-link[=`*value*`]`

Arguments

value	Can be one of the following:	
	`early`	Tells the compiler to generate an HTML report when the partial link is created. This capability lets you check the program if need be. You can resume from this point and generate an FPGA image by specifying option `-fintelfpga` with the generated binary.
	`image`	Tells the compiler to generate an FPGA bitstream. It will then be ready to be linked and used on an FPGA board.
	image takes much longer to generate than does *early*.	

Default

OFF	No partial link of device binaries is performed.

Description

This option tells the compiler to perform a partial link of device binaries to be used with FPGA.

This partial link is then wrapped by the offload wrapper, allowing the device binaries to be linked by the host compiler or linker.

If you do not specify a *value*, the following occurs:

- When used with just `-fsycl` (`-fsycl -fsycl-link`), the driver will generate a host linkable device object.
- When also used with `-fintelfpga` (`-fsycl -fintelfpga -fsycl-link`), the behavior is the same as specifying `-fsycl-link=early`.

NOTE
When using the icx/icpx compiler driver for this option, you must also specify option `-fsycl`.

When using the dpcpp compiler driver, option `-fsycl` is implied by default.

NOTE
When SYCL offloading is enabled, this option only applies to device-specific compilation.

IDE Equivalent

Visual Studio

Visual Studio: **Linker > General > Generate partially linked device object to be used with the host link**

Eclipse

Eclipse: **Linker > General > Generate partially linked device object to be used with the host link**

Alternate Options

None

See Also

fintelfpga
  compiler option

fsycl-link-targets

Tells the compiler to link only device code. This is a deprecated option that may be removed in a future release.

Syntax

Linux OS:

`-fsycl-link-targets=T1,...,Tn`

Windows OS:

`-fsycl-link-targets=T1,...,Tn`

Arguments

T	Is a target triple for the device code. You can specify more than one *T*.

Default

OFF	No link is performed.

Description

This option tells the compiler to link only device code. It is used in a link step.

It tells the compiler to link device code for the given target triples, and output multiple linked device code images. It does not produce fat binary.

> **NOTE**
> You should be familiar with ahead-of-time (AOT) compilation when using this option.

> **NOTE**
> When using the icx/icpx compiler driver for this option, you must also specify option `-fsycl`.
>
> When using the dpcpp compiler driver, option `-fsycl` is implied by default.

> **NOTE**
> When SYCL offloading is enabled, this option only applies to device-specific compilation.

IDE Equivalent

None

Alternate Options

None

Example

The following command-line sequence demonstrates a way to use this option:

```
dpcpp -fsycl-targets=spir64 -c a.cpp -o a.o
dpcpp -fsycl-targets=spir64 -c b.cpp -o b.o
dpcpp -fsycl-link-targets=spir64 a.o b.o -o linked.spv
aoc linked.spv -o linked.aocx
dpcpp -fsycl-add-targets=fpga:linked.aocx a.o b.o -o final.out -lOpenCL -lsycl
```

See Also

`fsycl` compiler option
`fsycl-add-targets` compiler option
`fsycl-targets` compiler option
Ahead of Time Compilation

fsycl-max-parallel-link-jobs

Tells the compiler that it can simultaneously spawn up to the specified number of processes to perform actions required to link SYCL applications. This is an experimental feature.

Syntax

Linux OS:

`-fsycl-max-parallel-link-jobs=`*n*

Windows OS:

`-fsycl-max-parallel-link-jobs=`*n*

Arguments

n	Is the number of processes to spawn to.

Default

`-fsycl-max-parallel-link-jobs=1`	One process is simultaneously spawned to perform actions necessary to link SYCL applications.

Description

This option tells the compiler that it can simultaneously spawn up to the specified number of processes to perform actions required to link SYCL applications. This is an experimental feature.

This option has no effect if compiler options such as `c` or `E` are specified.

NOTE
If you specify a large number of processes, it can cause performance issues and compilation crashes due to excessive RAM consumption.

NOTE
When using the icx/icpx compiler driver for this option, you must also specify option `-fsycl`.

When using the dpcpp compiler driver, option `-fsycl` is implied by default.

NOTE
When SYCL offloading is enabled, this option only applies to device-specific compilation.

IDE Equivalent

None

Alternate Options

None

Example

The following shows examples of using this option on Linux*:

```
dpcpp -fsycl-max-parallel-link-jobs=4 a.cpp b.cpp c.cpp d.cpp -o a.out
dpcpp -fsycl-max-parallel-link-jobs=8 a.o b.o c.o d.so e.a -o b.out
```

See Also

`fsycl` compiler option

fsycl-targets

Tells the compiler to generate code for specified device targets.

Syntax

Linux OS:

`-fsycl-targets=`*`T1`*`,...,`*`Tn`*

Windows OS:

`-fsycl-targets=`*`T1`*`,...,`*`Tn`*

Arguments

T Is a target triple device name. If you specify more than one *T*, they must be separated by commas. The following triplets are supported:

`spir64`	Tells the compiler to use default heuristics for SPIR64-based devices. This is the default. You can also specify this value as spir64-unknown-unknown.
`spir64_x86_64`	Tells the compiler to generate code for Intel® CPUs. You can also specify this value as spir64_x86_64-unknown-unknown.

`x86_64`	Tells the compiler to generate code ahead of time for x86_64 CPUs; it provides better debuggability. This triplet can also be specified as x86_64-unknown-unknown.
`spir64_fpga`	Tells the compiler to generate code for Intel® FPGA. You can also specify this value as spir64_fpga-unknown-unknown.
`spir64_gen`	Tells the compiler to generate code for Intel® Processor Graphics. You can also specify this value as spir64_gen-unknown-unknown.

Default

`spir64`	The compiler will use default heuristics for SPIR64-based devices.

Description

This option tells the compiler to generate code for specified device targets.

> **NOTE**
> The long syntax values that include `-sycldevice`, such as `spir64-unknown-unknown-sycldevice`, are still supported, but they are deprecated.

> **NOTE**
> When using the icx/icpx compiler driver for this option, you must also specify option `-fsycl`.
>
> When using the dpcpp compiler driver, option `-fsycl` is implied by default.

> **NOTE**
> When SYCL offloading is enabled, this option only applies to device-specific compilation.

IDE Equivalent

Visual Studio

Visual Studio: **DPC++ > General > Specify SYCL offloading targets for AOT compilation**

Eclipse

Eclipse: **Intel(R) oneAPI DPC++ Compiler > General > Specify SYCL offloading targets for AOT compilation**

Alternate Options

None

fsycl-unnamed-lambda

Enables unnamed SYCL lambda kernels.*

Syntax

Linux OS:

`-fsycl-unnamed-lambda`

`-fno-sycl-unnamed-lambda`

Windows OS:

`-fsycl-unnamed-lambda`

`-fno-sycl-unnamed-lambda`

Arguments

None

Default

ON	Unnamed SYCL lambda kernels are enabled.

Description

This option enables unnamed SYCL kernels that are defined as lambdas.

If you specify `-fno-sycl-unnamed-lambda`, unnamed SYCL lambda kernels are disabled.

NOTE
When using the icx/icpx compiler driver for option `-fsycl-unnamed-lambda`, you must also specify option `-fsycl`.

When using the dpcpp compiler driver, option `-fsycl` is implied by default.

NOTE
When SYCL offloading is enabled, this option only applies to device-specific compilation.

IDE Equivalent

Visual Studio

Visual Studio: **DPC++ > General > Allow unnamed SYCL lambda kernels**

Eclipse

Eclipse: **Intel(R) oneAPI DPC++ Compiler > Language > Allow unnamed SYCL lambda kernels**

Alternate Options

None

fsycl-use-bitcode

Tells the compiler to produce device code in LLVM Intermediate Representation (IR) bitcode format into fat objects.

Syntax

Linux OS:

`-fsycl-use-bitcode`

Windows OS:

`-fsycl-use-bitcode`

Arguments

None

Default

ON	LLVM IR bitcode format is emitted.

Description

This option tells the compiler to produce device code in LLVM Intermediate Representation (IR) bitcode format into fat objects.

NOTE
When using the icx/icpx compiler driver for this option, you must also specify option `-fsycl`.

When using the dpcpp compiler driver, option `-fsycl` is implied by default.

NOTE
When SYCL offloading is enabled, this option only applies to device-specific compilation.

IDE Equivalent

None

Alternate Options

None

nolibsycl

Disables linking of the SYCL runtime library.*

Syntax

Linux OS:

`-nolibsycl`

Windows OS:

`-nolibsycl`

Arguments

None

Default

OFF	The SYCL* runtime library is linked.

Description

This option disables linking of the SYCL* runtime library.

When using the icx/icpx compiler driver, this option is only effective if you have specified option `-fsycl`.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

None

qopenmp, Qopenmp

Enables recognition of OpenMP features and tells the parallelizer to generate multi-threaded code based on OpenMP* directives.*

Syntax

Linux OS:

`-qopenmp`

`-qno-openmp`

Windows OS:

`/Qopenmp`

`/Qopenmp-`

Arguments

None

Default

`-qno-openmp` or `/Qopenmp-`	No OpenMP* multi-threaded code is generated by the compiler.

Description

This option enables recognition of OpenMP* features and tells the parallelizer to generate multi-threaded code based on OpenMP* directives. The code can be executed in parallel on both uniprocessor and multiprocessor systems.

This option works with any optimization level. Specifying no optimization (-O0 on Linux* or /Od on Windows*) helps to debug OpenMP applications.

This option can also be specified as -fopenmp on Linux* systems.

NOTE
To enable offloading to a specified GPU target, you must also specify option fopenmp-targets (Linux*) or /Qopenmp-targets (Windows).

NOTE
Option -fopenmp is not the same as option -qopenmp. Option -fopenmp will *not* do offloading.

NOTE
Options that use OpenMP* API are available for both Intel® microprocessors and non-Intel microprocessors, but these options may perform additional optimizations on Intel® microprocessors than they perform on non-Intel microprocessors. The list of major, user-visible OpenMP constructs and features that may perform differently on Intel® microprocessors versus non-Intel microprocessors include: locks (internal and user visible), the SINGLE construct, barriers (explicit and implicit), parallel loop scheduling, reductions, memory allocation, thread affinity, and binding.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

IDE Equivalent

Visual Studio

Visual Studio: **Language > OpenMP* Support**

Eclipse

Eclipse: **Language > Process OpenMP Directives**

Alternate Options

Linux: -fiopenmp

Windows: /Qiopenmp

See Also

fopenmp-targets, Qopenmp-targets compiler option

fiopenmp, Qiopenmp compiler option

qopenmp-lib, Qopenmp-lib

Lets you specify an OpenMP run-time library to use for linking.*

Syntax

Linux OS:

`-qopenmp-lib=`*`type`*

Windows OS:

`/Qopenmp-lib:`*`type`*

Arguments

type	Specifies the type of library to use; it implies compatibility levels. Currently, the only possible value is:	
	`compat`	Tells the compiler to use the compatibility OpenMP* run-time library (libiomp). This setting provides compatibility with object files created using Microsoft* and GNU* compilers.

Default

`-qopenmp-lib=compat` or `/Qopenmp-lib:compat`	The compiler uses the compatibility OpenMP* run-time library (libiomp).

Description

This option lets you specify an OpenMP* run-time library to use for linking.

The compatibility OpenMP run-time libraries are compatible with object files created using the Microsoft* OpenMP run-time library (vcomp) or the GNU OpenMP run-time library (libgomp).

To use the compatibility OpenMP run-time library, compile and link your application using the compat setting for option `[q or Q]openmp-lib`. To use this option, you must also specify one of the following compiler options:

- Linux* systems: `-qopenmp` or `-qopenmp-stubs`
- Windows* systems: `/Qopenmp` or `/Qopenmp-stubs`

Linux

The compatibility Intel OpenMP* run-time library lets you combine OpenMP* object files compiled with the GNU* gcc or gfortran compilers with similar OpenMP* object files compiled with the Intel® C, Intel® C++, or Intel® Fortran Compiler. The linking phase results in a single, coherent copy of the run-time library.

This option is processed by the compiler driver command that initiates linking, adding library names explicitly to the link command.

Windows

The compatibility OpenMP* run-time library lets you combine OpenMP* object files compiled with the Microsoft* C/C++ compiler with OpenMP* object files compiled with the Intel® C, Intel® C++, or Intel® Fortran compilers. The linking phase results in a single, coherent copy of the run-time library.

This option is processed by the compiler, which adds directives to the compiled object file that are processed by the linker.

NOTE The compatibility OpenMP run-time library is not compatible with object files created using versions of the Intel compilers earlier than 10.0.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

None

See Also

`qopenmp, Qopenmp` compiler option

`qopenmp-stubs, Qopenmp-stubs` compiler option

qopenmp-link

Controls whether the compiler links to static or dynamic OpenMP run-time libraries.*

Syntax

Linux OS:

`-qopenmp-link=`*`library`*

Windows OS:

`None`

Arguments

library	Specifies the OpenMP library to use. Possible values are:	
	`static`	Tells the compiler to link to static OpenMP run-time libraries. Note that static OpenMP libraries are deprecated.
	`dynamic`	Tells the compiler to link to dynamic OpenMP run-time libraries.

Default

`-qopenmp-link=dynamic`	The compiler links to dynamic OpenMP* run-time libraries. However, if Linux* option `-static` is specified, the compiler links to static OpenMP run-time libraries.

Description

This option controls whether the compiler links to static or dynamic OpenMP* run-time libraries.

To link to the static OpenMP run-time library (RTL) and create a purely static executable, you must specify `-qopenmp-link=static`. However, we strongly recommend you use the default setting, `-qopenmp-link=dynamic`.

Option `-qopenmp-link=dynamic` cannot be used in conjunction with option `-static`. If you try to specify both options together, an error will be displayed.

NOTE
Compiler options `-static-intel` and `-shared-intel` (Linux*) have no effect on which OpenMP run-time library is linked.

NOTE
On Linux systems, the OpenMP runtime library depends on using libpthread and libc (libgcc when compiled with gcc). Libpthread and libc (libgcc) must both be static or both be dynamic.

If both libpthread and libc (libgcc) are static, then the static version of the OpenMP runtime should be used. If both libpthread and libc (libgcc) are dynamic, then either the static or dynamic version of the OpenMP runtime may be used.

NOTE
This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

None

qopenmp-simd, Qopenmp-simd

Enables or disables OpenMP SIMD compilation.*

Syntax

Linux OS:

`-qopenmp-simd`

`-qno-openmp-simd`

Windows OS:

`/Qopenmp-simd`

`/Qopenmp-simd-`

Arguments

None

Default

`-qno-openmp-simd` or `/Qopenmp-simd-`	OpenMP* SIMD compilation is disabled.

Description

This option enables or disables OpenMP* SIMD compilation.

You can use this option if you want to enable or disable the SIMD support with no impact on other OpenMP features. In this case, no OpenMP runtime library is needed to link and the compiler does not need to generate OpenMP runtime initialization code.

When you specify `[q or Q]openmp`, it implies `[q or Q]openmp-simd`.

If you specify this option with the `[q or Q]openmp` option, it can impact other OpenMP features.

Option `-qopenmp-simd` is equivalent to option `-fiopenmp-simd`; option `/Qopenmp-simd` is equivalent to option `/Qiopenmp-simd`.

NOTE
Advanced users who prefer to use OpenMP* as it is implemented by the LLVM community, can get most of that functionality by using options `-fopenmp` and `-fopenmp-simd`.

IDE Equivalent

None

Alternate Options

Linux: -fiopenmp-simd

Windows /Qiopenmp-simd

Example

The lines in the following example are equivalent to specifying only `[q or Q]openmp-simd`. In this case, only SIMD support is provided, the OpenMP* library is not linked, and only the !$OMP directives related to SIMD are processed:

Linux

```
-qno-openmp -qopenmp-simd
```

Windows

```
/Qopenmp- /Qopenmp-simd
```

In the following example, SIMD support is provided, the OpenMP library is linked, and OpenMP runtime initialization code is generated:

Linux

```
-qopenmp -qopenmp-simd
```

Windows

```
/Qopenmp /Qopenmp-simd
```

See Also

`qopenmp, Qopenmp` compiler option
`O` compiler option

qopenmp-stubs, Qopenmp-stubs

Enables compilation of OpenMP programs in sequential mode.*

Syntax

Linux OS:

`-qopenmp-stubs`

Windows OS:

`/Qopenmp-stubs`

Arguments

None

Default

OFF	The library of OpenMP* function stubs is not linked.

Description

This option enables compilation of OpenMP* programs in sequential mode. The OpenMP directives are ignored and a stub OpenMP library is linked.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

Windows

Visual Studio: **Language > OpenMP Support**

Linux

Eclipse: **Language > Process OpenMP Directives**

Alternate Options

None

See Also

`qopenmp, Qopenmp` compiler option

reuse-exe

Tells the compiler to speed up Field Programmable Gate Array (FPGA) target compile time by reusing a previously compiled FPGA hardware image.

Syntax

Linux OS:

`-reuse-exe=`*`exe-filename`*

Windows OS:

`-reuse-exe=`*`exe-filename`*

Arguments

exe-filename	Is a previously compiled SYCL* binary.

Default

OFF	There is no potential compile-time speed up by reusing the executable for the FPGA target.

Description

This option tells the compiler to speed up FPGA target compile time by reusing a previously compiled FPGA hardware image. This option is useful only when compiling for hardware.

It lets you minimize or avoid long Intel® Quartus® Prime Software compile times for FPGA targets when the device code is unchanged.

> **NOTE**
> When offloading is enabled, this option only applies to device-specific compilation.

IDE Equivalent

None

Alternate Options

None

Wno-sycl-strict

Disables warnings that enforce strict SYCL language compatibility.*

Syntax

Linux OS:

`-Wno-sycl-strict`

Windows OS:

`-Wno-sycl-strict`

Arguments

None

Default

OFF	Warnings that enforce strict SYCL* language compatibility are enabled.

Description

This option disables warnings that enforce strict SYCL* language compatibility.

IDE Equivalent

None

Alternate Options

None

Xopenmp-target

Enables options to be passed to the specified tool in the device compilation tool chain for the target.

Syntax

Linux OS:

`-Xopenmp-target-tool=T "options"`

Windows OS:

`-Xopenmp-target-tool=T "options"`

Arguments

tool — Can be one of the following:

`frontend`	Indicates the frontend + middle end of the Standard Portable Intermediate Representation (SPIR-V*)-based device compiler for target triple *T*. The middle end is the part of a SPIR-V*-based device compiler that generates SPIR-V*. This SPIR-V* is then passed by the dpcpp driver to the backend of target *T*.
`backend`	Indicates Ahead of Time (AOT) compilation for target triple *T* and Just in Time (JIT) compilation for target *T* at runtime.
`linker`	Indicates the device code linker for target triple *T*.

Some targets may have *frontend* and *backend* in one component; in that case, options are merged.

T — Is the target triple device.

options — Are the options you want to pass to *tool*.

Default

OFF	No options are passed to a tool.

Description

This option enables options to be passed to the specified tool in the device compilation tool chain for the target.

NOTE
When OpenMP* offloading is enabled, this option only applies to device-specific compilation.

IDE Equivalent

Windows

Visual Studio: **Linker > General > Pass <arg> to the backend of target device compiler specified by <triple> for OpenMP offload**

DPC++ > Language > Pass <arg> to the frontend of target device compiler for OpenMP offload

C/C++ > Language [Intel C++] > Pass <arg> to the frontend of target device compiler for OpenMP offload

Linker > General > Pass <arg> to the device code linker for OpenMP offload

Linux

Eclipse: **Linker(Or Intel C++ Linker) > General > Pass <arg> to the backend of target device compiler specified by <triple> for OpenMP offload**

Intel(R) oneAPI DPC++ Compiler > Language > Pass <arg> to the frontend of target device compiler for OpenMP offload

Intel C++ Compiler > Language > Pass <arg> to the frontend of target device compiler for OpenMP offload

Linker(Or Intel C++ Linker) > General > Pass <arg> to the device code linker for OpenMP offload

Alternate Options

None

Xs

Passes options to the backend tool.

Syntax

Linux OS:

`-Xs -`*`option`* or `-Xs`*`option`*

Windows OS:

`-Xs -`*`option`* or `-Xs`*`option`*

Arguments

option	Is the option that you want to pass to the backend tool in device compilation. To see the values you can use for *option*, specify compiler option `-fsycl-help` to display the help information for the offline tools.

Default

OFF	No options are passed to the backend tool.

Description

This option passes options to the backend tool. It is a shortcut for option `Xsycl-target-backend`.

For example, the following option (using syntax form `-Xs`*option*):

```
-Xsversion
```

and the following option (using syntax form `-Xs` *-option*):

```
-Xs -version
```

are both equivalent to specifying:

```
-Xsycl-target-backend -version
```

NOTE
When using Ahead of Time (AOT) compilation, the options passed with `-Xs` are not compiler options.

To see a list of the options you can pass with `-Xs` when using AOT, specify `-fsycl-help=gen`, `-fsycl-help=x86_64`, or `-fsycl-help=fpga` on the command line.

NOTE
When offloading is enabled, this option only applies to device-specific compilation.

IDE Equivalent

Visual Studio

Visual Studio: **Linker > General > Enable FPGA hardware build**

Eclipse

Eclipse: **Linker > General > Enable FPGA hardware build**

Alternate Options

None

See Also

Xsycl-target
compiler option

Xsycl-target

Enables options to be passed to the specified tool in the device compilation tool chain for the target.

Syntax

Linux OS:

`-Xsycl-target-`*`tool`*`=`*`T`* `"`*`options`*`"`

Windows OS:

`-Xsycl-target-`*`tool`*`=`*`T`* `"`*`options`*`"`

Arguments

`tool` Can be one of the following:

`frontend`	Indicates the frontend + middle end of the Standard Portable Intermediate Representation (SPIR-V*)-based device compiler for target triple *T*.

The middle end is the part of a SPIR-V*-based device compiler that generates SPIR-V*. This SPIR-V* is then passed by the dpcpp driver to the backend of target *T*.

`backend` — Indicates Ahead of Time (AOT) compilation for target triple *T* and Just in Time (JIT) compilation for target *T* at runtime.

`linker` — Indicates the device code linker for target triple *T*.

Some targets may have *frontend* and *backend* in one component; in that case, options are merged.

T — Is the target triple device.

options — Are the options you want to pass to *tool*.

Default

OFF — No options are passed to a tool.

Description

This option enables options to be passed to the specified tool in the device compilation tool chain for the target.

NOTE
When SYCL offloading is enabled, this option only applies to device-specific compilation.

IDE Equivalent

Visual Studio

Visual Studio: **Linker > General > Pass <arg> to the backend of target device compiler specified by <triple>** (target-backend)

DPC++ > General > Pass <arg> to the frontend of target device compiler (target-frontend)

Linker > General > Pass <arg> to the device code linker (target-linker)

Eclipse

Eclipse: **Linker > General > Pass <arg> to the backend of target device compiler specified by <triple>** (target-backend)

Intel(R) oneAPI DPC++ Compiler > General > Pass <arg> to the frontend of target device compiler (target-frontend)

Linker > General > Pass <arg> to the device code linker (target-linker)

Alternate Options

None.

See Also

Xs
compiler option

Floating-Point Options

This section contains descriptions for compiler options that pertain to floating-point calculations. They are listed in alphabetical order.

ffp-contract

Controls when the compiler is permitted to form fused floating-point operations, such as fused multiply-add (FMA). Fused operations are allowed to produce more precise results than performing the individual operations separately.

Syntax

Linux OS:

`-ffp-contract=`*`keyword`*

Windows OS:

None

Arguments

keyword	Possible values are:	
	`fast`	Fuses floating-point operations across statements.
	`on`	Fuses floating-point operations within the same statement.
	`off`	Does not fuse floating-point operations.

Default

`-ffp-contract=fast`	Fuses floating-point operations across statements. However, if option `-fp-model=strict` is specified, the default is `-ffp-contract=off`.

Description

This option controls when the compiler is permitted to form fused floating-point operations, such as fused multiply-add (FMA). Fused operations are allowed to produce more precise results than performing the individual operations separately.

IDE Equivalent

None

Alternate Options

None

See Also

`fp-model, fp` compiler option

fimf-absolute-error, Qimf-absolute-error

Defines the maximum allowable absolute error for math library function results.

Syntax

Linux OS:

`-fimf-absolute-error=`*value*`[:`*funclist*`]`

Windows OS:

`/Qimf-absolute-error:`*value*`[:`*funclist*`]`

Arguments

value	Is a positive, floating-point number. Errors in math library function results may exceed the maximum relative error (max-error) setting if the absolute-error is less than or equal to *value*. The format for the number is [digits] [.digits] [{ e | E }[sign]digits]
funclist	Is an optional list of one or more math library functions to which the attribute should be applied. If you specify more than one function, they must be separated with commas. Precision-specific variants like sin and sinf are considered different functions, so you would need to use `-fimf-absolute-error=0.00001:sin,sinf` (or `/Qimf-absolute-error:0.00001:sin,sinf`) to specify the maximum allowable absolute error for both the single-precision and double-precision sine functions. You also can specify the symbol /f to denote single-precision divides, symbol / to denote double-precision divides, symbol /l to denote extended-precision divides, and symbol /q to denote quad-precision divides. For example you can specify `-fimf-absolute-error=0.00001:/` or `/Qimf-absolute-error: 0.00001:/`.

Default

Zero ("0")	An absolute-error setting of 0 means that the function is bound by the relative error setting. This is the default behavior.

Description

This option defines the maximum allowable absolute error for math library function results.

This option can improve run-time performance, but it may decrease the accuracy of results.

This option only affects functions that have zero as a possible return value, such as log, sin, asin, etc.

The relative error requirements for a particular function are determined by options that set the maximum relative error (max-error) and precision. The return value from a function must have a relative error less than the max-error value, or an absolute error less than the absolute-error value.

If you need to define the accuracy for a math function of a certain precision, specify the function name of the precision that you need. For example, if you want double precision, you can specify :sin; if you want single precision, you can specify :sinf, as in `-fimf-absolute-error=0.00001:sin` or `/Qimf-absolute-error:0.00001:sin`, or `-fimf-absolute-error=0.00001:sqrtf` or `/Qimf-absolute-error:0.00001:sqrtf`.

If you do not specify any function names, then the setting applies to all functions (and to all precisions). However, as soon as you specify an individual function name, the setting applies only to the function of corresponding precision. So, for example, sinf applies only to the single-precision sine function, sin applies only to the double-precision sine function, sinl applies only to the extended-precision sine function, etc.

> **NOTE**
> Many routines in libraries LIBM (Math Library) and SVML (Short Vector Math Library) are more highly optimized for Intel® microprocessors than for non-Intel microprocessors.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

IDE Equivalent

None

Alternate Options

None

See Also

`fimf-accuracy-bits, Qimf-accuracy-bits` compiler option

`fimf-arch-consistency, Qimf-arch-consistency` compiler option

`fimf-domain-exclusion, Qimf-domain-exclusion` compiler option

`fimf-max-error, Qimf-max-error` compiler option

`fimf-precision, Qimf-precision` compiler option

`fimf-use-svml, Qimf-use-svml` compiler option

fimf-accuracy-bits, Qimf-accuracy-bits

Defines the relative error for math library function results, including division and square root.

Syntax

Linux OS:

`-fimf-accuracy-bits=`*`bits`*`[:`*`funclist`*`]`

Windows OS:

`/Qimf-accuracy-bits:`*`bits`*`[:`*`funclist`*`]`

Arguments

bits	Is a positive, floating-point number indicating the number of correct bits the compiler should use. The format for the number is [digits] [.digits] [{ e | E }[sign]digits].
funclist	Is an optional list of one or more math library functions to which the attribute should be applied. If you specify more than one function, they must be separated with commas. Precision-specific variants like sin and sinf are considered different functions, so you would need to use `-fimf-accuracy-bits=23:sin,sinf` (or `/Qimf-accuracy-bits:23:sin,sinf`) to specify the relative error for both the single-precision and double-precision sine functions. You also can specify the symbol /f to denote single-precision divides, symbol / to denote double-precision divides, symbol /l to denote extended-precision divides, and symbol /q to denote quad-precision divides. For example you can specify `-fimf-accuracy-bits=10.0:/f` or `/Qimf-accuracy-bits:10.0:/f`.

Default

-fimf-precision=medium or /Qimf-precision:medium	The compiler uses medium precision when calling math library functions. Note that other options can affect precision; see below for details.

Description

This option defines the relative error, measured by the number of correct bits, for math library function results.

The following formula is used to convert bits into ulps: ulps = $2^{p-1-bits}$, where p is the number of the target format mantissa bits (24, 53, and 64 for single, double, and long double, respectively).

This option can affect run-time performance and the accuracy of results.

If you need to define the accuracy for a math function of a certain precision, specify the function name of the precision that you need. For example, if you want double precision, you can specify :sin; if you want single precision, you can specify :sinf, as in the following:

Linux

- `-fimf-accuracy-bits=23:sinf,cosf,logf`
- `-fimf-accuracy-bits=52:sqrt,/,trunc`
- `-fimf-accuracy-bits=10:powf`

Windows

- `/Qimf-accuracy-bits:23:sinf,cosf,logf`
- `/Qimf-accuracy-bits:52:sqrt,/,trunc`
- `/Qimf-accuracy-bits:10:powf`

If you do not specify any function names, then the setting applies to all functions (and to all precisions). However, as soon as you specify an individual function name, the setting applies only to the function of corresponding precision. So, for example, sinf applies only to the single-precision sine function, sin applies only to the double-precision sine function, sinl applies only to the extended-precision sine function, etc.

There are three options you can use to express the maximum relative error. They are as follows:

Linux

- `-fimf-precision`
- `-fimf-max-error`
- `-fimf-accuracy-bits`

Windows

- `/Qimf-precision`
- `/Qimf-max-error`
- `/Qimf-accuracy-bits`

If more than one of these options are specified, the default value for the maximum relative error is determined by the last one specified on the command line.

If none of the above options are specified, the default values for the maximum relative error are determined by the setting of the following options:

- `-fp-model` (Linux) or `/fp` (Windows)

NOTE
Many routines in libraries LIBM (Math Library) and SVML (Short Vector Math Library) are more highly optimized for Intel® microprocessors than for non-Intel microprocessors.

NOTE
This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

IDE Equivalent

None

Alternate Options

None

See Also

`fimf-absolute-error, Qimf-absolute-error` compiler option

`fimf-arch-consistency, Qimf-arch-consistency` compiler option

`fimf-domain-exclusion, Qimf-domain-exclusion` compiler option

`fimf-max-error, Qimf-max-error` compiler option

`fimf-precision, Qimf-precision` compiler option

`fimf-use-svml, Qimf-use-svml` compiler option

fimf-arch-consistency, Qimf-arch-consistency

Ensures that the math library functions produce consistent results across different microarchitectural implementations of the same architecture.

Syntax

Linux OS:

`-fimf-arch-consistency=`*`value`*`[:`*`funclist`*`]`

Windows OS:

`/Qimf-arch-consistency:`*`value`*`[:`*`funclist`*`]`

Arguments

value	Is one of the logical values "true" or "false".
funclist	Is an optional list of one or more math library functions to which the attribute should be applied. If you specify more than one function, they must be separated with commas. Precision-specific variants like sin and sinf are considered different functions, so you would need to use `-fimf-arch-consistency=true:sin,sinf` (or `/Qimf-arch-consistency:true:sin,sinf`) to specify consistent results for both the single-precision and double-precision sine functions. You also can specify the symbol /f to denote single-precision divides, symbol / to denote double-precision divides, symbol /l to denote extended-precision divides, and symbol /q to denote quad-precision divides. For example you can specify `-fimf-arch-consistency=true:/` or `/Qimf-arch-consistency:true:/`.

Default

false	Implementations of some math library functions may produce slightly different results on implementations of the same architecture.

Description

This option ensures that the math library functions produce consistent results across different microarchitectural implementations of the same architecture (for example, across different microarchitectural implementations of IA-32 architecture). Consistency is only guaranteed for a single binary. Consistency is not guaranteed across different architectures. For example, consistency is not guaranteed across IA-32 architecture and Intel® 64 architecture.

If you need to define the accuracy for a math function of a certain precision, specify the function name of the precision that you need. For example:

Linux

If you want double precision, you can specify :sin; if you want single precision, you can specify :sinf, as in `-fimf-arch-consistency=true:sin` or `-fimf-arch-consistency=false:sqrtf`.

Windows

If you want double precision, you can specify :sin; if you want single precision, you can specify :sinf, as in `/Qimf-arch-consistency:true:sin` or `/Qimf-arch-consistency:false:sqrtf`.

If you do not specify any function names, then the setting applies to all functions (and to all precisions). However, as soon as you specify an individual function name, the setting applies only to the function of corresponding precision. So, for example, sinf applies only to the single-precision sine function, sin applies only to the double-precision sine function, sinl applies only to the extended-precision sine function, etc.

The `-fimf-arch-consistency` (Linux*) and `/Qimf-arch-consistency` (Windows*) option may decrease run-time performance, but the option will provide bit-wise consistent results on all Intel® processors and compatible, non-Intel processors, regardless of micro-architecture. This option may not provide bit-wise consistent results between different architectures.

> **NOTE**
> Many routines in libraries LIBM (Math Library) and SVML (Short Vector Math Library) are more highly optimized for Intel® microprocessors than for non-Intel microprocessors.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

IDE Equivalent

None

Alternate Options

None

See Also

`fimf-absolute-error, Qimf-absolute-error` compiler option

`fimf-accuracy-bits, Qimf-accuracy-bits` compiler option

`fimf-domain-exclusion, Qimf-domain-exclusion` compiler option

`fimf-max-error, Qimf-max-error` compiler option

`fimf-precision, Qimf-precision` compiler option

`fimf-use-svml Qimf-use-svml` compiler option

fimf-domain-exclusion, Qimf-domain-exclusion

Indicates the input arguments domain on which math functions must provide correct results.

Syntax

Linux OS:

`-fimf-domain-exclusion=`*`classlist`*`[:`*`funclist`*`]`

Windows OS:

`/Qimf-domain-exclusion:`*`classlist`*`[:`*`funclist`*`]`

Arguments

classlist

Is one of the following:

- One or more of the following floating-point value classes you can exclude from the function domain without affecting the correctness of your program. The supported class names are:

  `extremes` This class is for values which do not lie within the usual domain of arguments for a given function.

  `nans` This means "x=Nan".

  `infinities` This means "x=infinities".

  `denormals` This means "x=denormal".

  `zeros` This means "x=0".

  Each *classlist* element corresponds to a power of two. The exclusion attribute is the logical or of the associated powers of two (that is, a bitmask).

  The following shows the current mapping from *classlist* mnemonics to numerical values:

extremes	1
nans	2
infinities	4
denormals	8
zeros	16
none	0
all	31
common	15
other combinations	bitwise OR of the used values

  You must specify the integer value that corresponds to the class that you want to exclude.

  Note that on excluded values, unexpected results may occur.

- One of the following short-hand tokens:

  `none` This means that none of the supported classes are excluded from the domain. To indicate this token, specify 0, as in `-fimf-domain-exclusion=0` (or `/Qimf-domain-exclusion:0`).

  `all` This means that all of the supported classes are excluded from the domain. To indicate this token, specify 31, as in `-fimf-domain-exclusion=31` (or `/Qimf-domain-exclusion:31`).

`common`	This is the same as specifying extremes,nans,infinities,denormals. To indicate this token, specify 15 (1 + 2+ 4 + 8), as in `-fimf-domain-exclusion=15` (or `/Qimf-domain-exclusion:15`)

funclist

Is an optional list of one or more math library functions to which the attribute should be applied. If you specify more than one function, they must be separated with commas.

Precision-specific variants like sin and sinf are considered different functions, so you would need to use `-fimf-domain-exclusion=4:sin,sinf` (or `/Qimf-domain-exclusion:4:sin,sinf`) to specify infinities for both the single-precision and double-precision sine functions.

You also can specify the symbol /f to denote single-precision divides, symbol / to denote double-precision divides, symbol /l to denote extended-precision divides, and symbol /q to denote quad-precision divides. For example, you can specify:

`-fimf-domain-exclusion=4` or `/Qimf-domain-exclusion:4`

`-fimf-domain-exclusion=5:/,powf` or `/Qimf-domain-exclusion:5:/,powf`

`-fimf-domain-exclusion=23:log,logf,/,sin,cosf` or `/Qimf-domain-exclusion:23:log,logf,/,sin,cosf`

If you don't specify argument *funclist*, the domain restrictions apply to all math library functions.

Default

Zero ("0")	The compiler uses default heuristics when calling math library functions.

Description

This option indicates the input arguments domain on which math functions must provide correct results. It specifies that your program will function correctly if the functions specified in *funclist* do not produce standard conforming results on the number classes.

This option can affect run-time performance and the accuracy of results. As more classes are excluded, faster code sequences can be used.

If you need to define the accuracy for a math function of a certain precision, specify the function name of the precision that you need. For example, if you want double precision, you can specify :sin; if you want single precision, you can specify :sinf, as in `-fimf-domain-exclusion=denormals:sin` or `/Qimf-domain-exclusion:denormals:sin`, or `-fimf-domain-exclusion=extremes:sqrtf` or `/Qimf-domain-exclusion:extremes:sqrtf`.

If you do not specify any function names, then the setting applies to all functions (and to all precisions). However, as soon as you specify an individual function name, the setting applies only to the function of corresponding precision. So, for example, sinf applies only to the single-precision sine function, sin applies only to the double-precision sine function, sinl applies only to the extended-precision sine function, etc.

NOTE
Many routines in libraries LIBM (Math Library) and SVML (Short Vector Math Library) are more highly optimized for Intel® microprocessors than for non-Intel microprocessors.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.   Notice revision #20201201

IDE Equivalent

None

Alternate Options

None

Example

Consider the following single-precision sequence for function exp2f:

Operation:	y = exp2f(x)
Accuracy:	1.014 ulp
Instructions:	4 (2 without fix-up)

The following shows the 2-instruction sequence without the fix-up:

```
vcvtfxpntps2dq  zmm1 {k1}, zmm0, 0x50      // zmm1 <-- rndToInt(2^24 * x)
vexp223ps       zmm1 {k1}, zmm1            // zmm1 <-- exp2(x)
```

However, the above 2-instruction sequence will not correctly process NaNs. To process Nans correctly, the following fix-up must be included following the above instruction sequence:

```
vpxord          zmm2, zmm2, zmm2             // zmm2 <-- 0
vfixupnanps     zmm1 {k1}, zmm0, zmm2 {aaaa} // zmm1 <-- QNaN(x) if x is NaN <F>
```

If the `vfixupnanps` instruction is not included, the sequence correctly processes any arguments except NaN values. For example, the following options generate the 2-instruction sequence:

```
-fimf-domain-exclusion=2:exp2f      <- NaN's are excluded (2 corresponds to NaNs)
-fimf-domain-exclusion=6:exp2f      <- NaN's and infinities are excluded (4 corresponds to
infinities; 2 + 4 = 6)
-fimf-domain-exclusion=7:exp2f      <- NaN's, infinities, and extremes are excluded (1
corresponds to extremes; 2 + 4 + 1 = 7)
-fimf-domain-exclusion=15:exp2f     <- NaN's, infinities, extremes, and denormals are excluded
(8 corresponds to denormals;  2 + 4 + 1 + 8=15)
```

If the `vfixupnanps` instruction is included, the sequence correctly processes any arguments including NaN values. For example, the following options generate the 4-instruction sequence:

```
-fimf-domain-exclusion=1:exp2f      <- only extremes are excluded (1 corresponds to extremes)
-fimf-domain-exclusion=4:exp2f      <- only infinities are excluded (4 corresponds to infinities)
-fimf-domain-exclusion=8:exp2f      <- only denormals are excluded (8 corresponds to denormals)
-fimf-domain-exclusion=13:exp2f     <- only extremes, infinities and denormals are excluded (1 +
4 + 8 = 13)
```

See Also

`fimf-absolute-error, Qimf-absolute-error` compiler option

`fimf-accuracy-bits, Qimf-accuracy-bits` compiler option
`fimf-arch-consistency, Qimf-arch-consistency` compiler option
`fimf-max-error, Qimf-max-error` compiler option
`fimf-precision, Qimf-precision` compiler option
`fimf-use-svml_Qimf-use-svml` compiler option

fimf-force-dynamic-target, Qimf-force-dynamic-target

Instructs the compiler to use run-time dispatch in calls to math functions.

Syntax

Linux OS:

`-fimf-force-dynamic-target[=`*`funclist`*`]`

Windows OS:

`/Qimf-force-dynamic-target[:`*`funclist`*`]`

Arguments

funclist	Is an optional list of one or more math library functions to which the attribute should be applied. If you specify more than one function, they must be separated with commas. Precision-specific variants like sin and sinf are considered different functions, so you would need to use `-fimf-dynamic-target=sin,sinf` (or `/Qimf-dynamic-target:sin,sinf`) to specify run-time dispatch for both the single-precision and double-precision sine functions. You also can specify the symbol /f to denote single-precision divides, symbol / to denote double-precision divides, symbol /l to denote extended-precision divides, and symbol /q to denote quad-precision divides. For example, you can specify `-fimf-dynamic-target=/` or `/Qimf-dynamic-target:/`.

Default

OFF	Run-time dispatch is not forced in math libraries calls. The compiler can choose to call a CPU-specific version of a math function if one is available.

Description

This option instructs the compiler to use run-time dispatch in calls to math functions. When this option set to ON, it lets you force run-time dispatch in math libraries calls.

If you want to target multiple CPU families with a single application or you prefer to choose a target CPU at run time, you can force run-time dispatch in math libraries by using this option.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

IDE Equivalent

None

Alternate Options

None

fimf-max-error, Qimf-max-error

Defines the maximum allowable relative error for math library function results, including division and square root.

Syntax

Linux OS:

`-fimf-max-error=`*`ulps`*`[:`*`funclist`*`]`

Windows OS:

`/Qimf-max-error:`*`ulps`*`[:`*`funclist`*`]`

Arguments

ulps	Is a positive, floating-point number indicating the maximum allowable relative error the compiler should use. The format for the number is [digits] [.digits] [{ e | E }[sign]digits].
funclist	Is an optional list of one or more math library functions to which the attribute should be applied. If you specify more than one function, they must be separated with commas. Precision-specific variants like sin and sinf are considered different functions, so you would need to use `-fimf-max-error=4.0:sin,sinf` (or `/Qimf-max-error=4.0:sin,sinf`) to specify the maximum allowable relative error for both the single-precision and double-precision sine functions. You also can specify the symbol /f to denote single-precision divides, symbol / to denote double-precision divides, symbol /l to denote extended-precision divides, and symbol /q to denote quad-precision divides. For example you can specify `-fimf-max-error=4.0:/` or `/Qimf-max-error:4.0:/`.

Default

-fimf-precision=medium or /Qimf-precision:medium	The compiler uses medium precision when calling math library functions. Note that other options can affect precision; see below for details.

Description

This option defines the maximum allowable relative error, measured in ulps, for math library function results.

This option can affect run-time performance and the accuracy of results.

If you need to define the accuracy for a math function of a certain precision, specify the function name of the precision that you need. For example, if you want double precision, you can specify :sin; if you want single precision, you can specify :sinf, as in `-fimf-max-error=4.0:sin` or `/Qimf-max-error:4.0:sin`, or `-fimf-max-error=4.0:sqrtf` or `/Qimf-max-error:4.0:sqrtf`.

If you do not specify any function names, then the setting applies to all functions (and to all precisions). However, as soon as you specify an individual function name, the setting applies only to the function of corresponding precision. So, for example, sinf applies only to the single-precision sine function, sin applies only to the double-precision sine function, sinl applies only to the extended-precision sine function, etc.

There are three options you can use to express the maximum relative error. They are as follows:

Linux

- `-fimf-precision`
- `-fimf-max-error`
- `-fimf-accuracy-bits`

Windows

- `/Qimf-precision`
- `/Qimf-max-error`
- `/Qimf-accuracy-bits`

If more than one of these options are specified, the default value for the maximum relative error is determined by the last one specified on the command line.

If none of the above options are specified, the default values for the maximum relative error are determined by the setting of the following options:

- `-fp-model` (Linux) or `/fp` (Windows)

> **NOTE**
> Many routines in libraries LIBM (Math Library) and SVML (Short Vector Math Library) are more highly optimized for Intel® microprocessors than for non-Intel microprocessors.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

IDE Equivalent

None

Alternate Options

None

See Also

`fimf-absolute-error, Qimf-absolute-error` compiler option
`fimf-accuracy-bits, Qimf-accuracy-bits` compiler option
`fimf-arch-consistency, Qimf-arch-consistency` compiler option
`fimf-domain-exclusion, Qimf-domain-exclusion` compiler option
`fimf-precision, Qimf-precision` compiler option
`fimf-use-svml_Qimf-use-svml` compiler option

fimf-precision, Qimf-precision

Lets you specify a level of accuracy (precision) that the compiler should use when determining which math library functions to use.

Syntax

Linux OS:

`-fimf-precision[=`*value*`[:`*funclist*`]]`

Windows OS:

`/Qimf-precision[:`*value*`[:`*funclist*`]]`

Arguments

value

Is one of the following values denoting the desired accuracy:

`high`	This is equivalent to max-error = 1.0.
`medium`	This is equivalent to max-error = 4; this is the default setting if the option is specified and *value* is omitted.
`low`	This is equivalent to accuracy-bits = 11 for single-precision functions; accuracy-bits = 26 for double-precision functions.

Linux

In the above explanations, max-error means option `-fimf-max-error`; accuracy-bits means option `-fimf-accuracy-bits`.

Windows

In the above explanations, max-error means option `/Qimf-max-error` (Windows*); accuracy-bits means option `/Qimf-accuracy-bits`.

funclist

Is an optional list of one or more math library functions to which the attribute should be applied.

If you specify more than one function, they must be separated with commas.

Precision-specific variants like sin and sinf are considered different functions, so you would need to use `-fimf-precision=high:sin,sinf` (or `/Qimf-precision:high:sin,sinf`) to specify high precision for both the single-precision and double-precision sine functions.

You also can specify the symbol /f to denote single-precision divides, symbol / to denote double-precision divides, symbol /l to denote extended-precision divides, and symbol /q to denote quad-precision divides. For example you can specify `-fimf-precision=low:/` or `/Qimf-precision:low:/` and `-fimf-precision=low:/f` or `/Qimf-precision:low:/f`.

Default

medium	The compiler uses medium precision when calling math library functions. Note that other options can affect precision; see below for details.

Description

This option lets you specify a level of accuracy (precision) that the compiler should use when determining which math library functions to use.

This option can be used to improve run-time performance if reduced accuracy is sufficient for the application, or it can be used to increase the accuracy of math library functions selected by the compiler.

In general, using a lower precision can improve run-time performance and using a higher precision may reduce run-time performance.

If you need to define the accuracy for a math function of a certain precision, specify the function name of the precision that you need. For example, if you want double precision, you can specify :sin; if you want single precision, you can specify :sinf, as in `-fimf-precision=low:sin` or `/Qimf-precision:low:sin`, or `-fimf-precision=high:sqrtf` or `/Qimf-precision:high:sqrtf`.

If you do not specify any function names, then the setting applies to all functions (and to all precisions). However, as soon as you specify an individual function name, the setting applies only to the function of corresponding precision. So, for example, sinf applies only to the single-precision sine function, sin applies only to the double-precision sine function, sinl applies only to the extended-precision sine function, etc.

There are three options you can use to express the maximum relative error. They are as follows:

Linux

- `-fimf-precision`
- `-fimf-max-error`
- `-fimf-accuracy-bits`

Windows

- `/Qimf-precision`
- `/Qimf-max-error`
- `/Qimf-accuracy-bits`

If more than one of these options are specified, the default value for the maximum relative error is determined by the last one specified on the command line.

If none of the above options are specified, the default values for the maximum relative error are determined by the setting of the following options:

- `-fp-model` (Linux) or `/fp` (Windows)

NOTE
Many routines in libraries LIBM (Math Library) and SVML (Short Vector Math Library) are more highly optimized for Intel® microprocessors than for non-Intel microprocessors.

NOTE
This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

IDE Equivalent

None

Alternate Options

None

See Also

`fimf-absolute-error, Qimf-absolute-error` compiler option
`fimf-accuracy-bits, Qimf-accuracy-bits` compiler option
`fimf-arch-consistency, Qimf-arch-consistency` compiler option
`fimf-domain-exclusion, Qimf-domain-exclusion` compiler option
`fimf-max-error, Qimf-max-error` compiler option
`fp-model, fp` compiler option
`fimf-use-svml_Qimf-use-svml` compiler option

fimf-use-svml, Qimf-use-svml

Instructs the compiler to use the Short Vector Math Library (SVML) rather than the Intel® oneAPI DPC++/C++ Compiler Math Library (LIBM) to implement math library functions.

Syntax

Linux OS:

-fimf-use-svml=*value*[:*funclist*]

Windows OS:

/Qimf-use-svml:*value*[:*funclist*]

Arguments

funclist	Is an optional list of one or more math library functions to which the attribute should be applied. If you specify more than one function, they must be separated with commas.

Precision-specific variants like sin and sinf are considered different functions, so you would need to use `-fimf-use-svmlt=true:sin,sinf` (or `/Qimf-use-svml:true:sin,sinf`) to specify that both the single-precision and double-precision sine functions should use SVML.

Default

false	Math library functions are implemented using the Intel® oneAPI DPC++/C++ Compiler Math Library, though other compiler options may give the compiler the flexibility to implement math library functions with either LIBM or SVML.

Description

This option instructs the compiler to implement math library functions using the Short Vector Math Library (SVML).

Linux

When you specify option `-fimf-use-svml=true`, the specific SVML variant chosen is influenced by other compiler options such as `-fimf-precision` and `-fp-model`.

Windows

When you specify option `/Qimf-use-svml:true`, the specific SVML variant chosen is influenced by other compiler options such as `/Qimf-precision` and `/fp`.

This option has no effect on math library functions that are implemented in LIBM but not in SVML.

In value-safe settings of option `-fp-model` (Linux) or option `/fp` (Windows) such as precise, this option causes a slight decrease in the accuracy of math library functions, because even the high accuracy SVML functions are slightly less accurate than the corresponding functions in LIBM. Additionally, the SVML functions might not accurately raise floating-point exceptions, do not maintain errno, and are designed to work correctly only in round-to-nearest-even rounding mode.

The benefit of using `-fimf-use-svml=true` or `/Qimf-use-svml:true` with value-safe settings of `-fp-model` (Linux) or /fp (Windows) is that it can significantly improve performance by enabling the compiler to efficiently vectorize loops containing calls to math library functions.

If you need to use SVML for a specific math function of a certain precision, specify the function name of the precision that you need. For example, if you want double precision, you can specify :sin; if you want single precision, you can specify :sqrtf, as in `-fimf-use-svml=true:sin` or `/Qimf-use-svml:true:sin`, or `-fimf-use-svml =false:sqrtf` or `/Qimf-use-svml:false:sqrtf`.

If you do not specify any function names, then the setting applies to all functions (and to all precisions). However, as soon as you specify an individual function name, the setting applies only to the function of corresponding precision. So, for example, sinf applies only to the single-precision sine function, sin applies only to the double-precision sine function, sinl applies only to the extended-precision sine function, etc.

NOTE
Since SVML functions may raise unexpected floating-point exceptions, be cautious about using features that enable trapping on floating-point exceptions.

NOTE
This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

IDE Equivalent

None

Alternate Options

None

See Also

`fp-model, fp` compiler option

fma, Qfma

Determines whether the compiler generates fused multiply-add (FMA) instructions if such instructions exist on the target processor.

Syntax

Linux OS:

`-fma`

`-no-fma`

Windows OS:

`/Qfma`

`/Qfma-`

Arguments

None

Default

`-fma` or `/Qfma`	If the instructions exist on the target processor, the compiler generates fused multiply-add (FMA) instructions. However, if you specify `-fp-model strict` (Linux*) or `/fp:strict` (Windows*), but do not explicitly specify `-fma` or `/Qfma`, the default is `-no-fma` or `/Qfma-`.

Description

This option determines whether the compiler generates fused multiply-add (FMA) instructions if such instructions exist on the target processor. When the `[Q]fma` option is specified, the compiler may generate FMA instructions for combining multiply and add operations. When the negative form of the `[Q]fma` option is specified, the compiler must generate separate multiply and add instructions with intermediate rounding.

This option has no effect unless setting CORE-AVX2 or higher is specified for option [Q]x,`-march` (Linux), or `/arch` (Windows).

IDE Equivalent

None

See Also

`fp-model, fp` compiler option
`x, Qx` compiler option
`march` compiler option
`arch` compiler option

fp-model, fp

Controls the semantics of floating-point calculations.

Syntax

Linux OS:

`-fp-model=`*`keyword`*

Windows OS:

`/fp:`*`keyword`*

Arguments

keyword	Specifies the semantics to be used. Possible values are:	
	`precise`	Disables optimizations that are not value-safe on floating-point data.
	`fast[=1|2]`	Enables more aggressive optimizations on floating-point data. There is currently no difference between `fast=1` and `fast=2`.
	`strict`	Enables `precise`, disables contractions, and enables pragma `stdc fenv_access`.

Default

`-fp-model=fast` or `/fp:fast`	The compiler uses more aggressive optimizations on floating-point calculations.

Description

This option controls the semantics of floating-point calculations.

The floating-point (FP) environment is a collection of registers that control the behavior of FP machine instructions and indicate the current FP status. The floating-point environment may include rounding-mode controls, exception masks, flush-to-zero controls, exception status flags, and other floating-point related features.

Option	**Description**
`-fp-model=precise` or `/fp:precise`	Tells the compiler to strictly adhere to value-safe optimizations when implementing floating-point calculations. It disables optimizations that can change the result of floating-point calculations, which is required for strict ANSI conformance. These semantics ensure the reproducibility of floating-point computations for serial code, including code vectorized or auto-parallelized by the compiler, but they may slow performance. They do not ensure value safety or run-to-run reproducibility of other parallel code. Run-to-run reproducibility for floating-point reductions in OpenMP* code may be obtained for a fixed number of threads through the KMP_DETERMINISTIC_REDUCTION environment variable. For more information about this environment variable, see topic "Supported Environment Variables". The compiler assumes the default floating-point environment; you are not allowed to modify it.
`-fp-model=fast[=1|2]` or `/fp:fast[=1|2]`	Tells the compiler to use more aggressive optimizations when implementing floating-point calculations. These optimizations increase speed, but may affect the accuracy or reproducibility of floating-point computations. There is currently no difference between fast=1 and fast=2.
`-fp-model=strict` or `/fp:strict`	Tells the compiler to strictly adhere to value-safe optimizations when implementing floating-point calculations and enables floating-point exception semantics. This is the strictest floating-point model. The compiler does not assume the default floating-point environment; you are allowed to modify it.

The `-fp-model` and `/fp` options determine the setting for the maximum allowable relative error for math library function results (max-error) if none of the following options are specified:

- `-fimf-accuracy-bits` (Linux*) or `/Qimf-accuracy-bits` (Windows*)
- `-fimf-max-error` (Linux) or `/Qimf-max-error` (Windows)
- `-fimf-precision` (Linux) or `/Qimf-precision` (Windows)

Option `-fp-model=fast` (and `/fp:fast`) sets option `-fimf-precision=medium` (`/Qimf-precision:medium`) and option `-fp-model=precise` (and `/fp:precise`); it implies `-fimf-precision=high` (and `/Qimf-precision:high`).

Option `-fp-model=fast=2` (and `/fp:fast2`) sets option `-fimf-precision=medium` (and `/Qimf-precision:medium`) and option `-fimf-domain-exclusion=15` (and `/Qimf-domain-exclusion=15`).

NOTE
In Microsoft* Visual Studio, when you create a Microsoft* Visual C++ project, option `/fp:precise` is set by default. It sets the floating-point model to improve consistency for floating-point operations by disabling certain optimizations that may reduce performance. To set the option back to the general default `/fp:fast`, change the IDE project property for Floating Point Model to Fast.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

IDE Equivalent

Visual Studio

Visual Studio: **Code Generation>Floating Point Model**

Code Generation>Enable Floating Point Exceptions

Code Generation> Floating Point Expression Evaluation

Eclipse

Eclipse: **Floating Point > Floating Point Model**

Alternate Options

None

See Also

`O` compiler option (specifically O0)

`Od` compiler option

`fimf-absolute-error, Qimf-absolute-error` compiler option

`fimf-accuracy-bits, Qimf-accuracy-bits` compiler option

`fimf-max-error, Qimf-max-error` compiler option

`fimf-precision, Qimf-precision` compiler option

`fimf-domain-exclusion, Qimf-domain-exclusion` compiler option

Supported Environment Variables

The article titled: Consistency of Floating-Point Results using the Intel® Compiler

fp-speculation, Qfp-speculation

Tells the compiler the mode in which to speculate on floating-point operations.

Syntax

Linux OS:

`-fp-speculation=`*mode*

Windows OS:

`/Qfp-speculation:`*mode*

Arguments

mode	Is the mode for floating-point operations. Possible values are:	
	`fast`	Tells the compiler to speculate on floating-point operations.
	`safe`	Tells the compiler to disable speculation if there is a possibility that the speculation may cause a floating-point exception.
	`strict`	Tells the compiler to disable speculation on floating-point operations.
	`off`	This is the same as specifying strict.

Default

`-fp-speculation=fast` or `/Qfp-speculation:fast`

The compiler speculates on floating-point operations. This is also the behavior when optimizations are enabled.

However, if you specify no optimizations (`-O0`), the default changes:

Linux

In this case, the default is `-fp-speculation=safe`.

Windows

In this case, the default is `/Qfp-speculation:safe`.

Description

This option tells the compiler the mode in which to speculate on floating-point operations.

Disabling speculation may prevent the vectorization of some loops containing conditionals.

IDE Equivalent

Visual Studio

Visual Studio: **Optimization > Floating-Point Speculation**

Eclipse

Eclipse: **Floating Point > Floating-Point Speculation**

Alternate Options

None

pc, Qpc

Enables control of floating-point significand precision.

Syntax

Linux OS:

`-pc`*`n`*

Windows OS:

`/Qpc`*`n`*

Arguments

n	Is the floating-point significand precision. Possible values are:	
	`32`	Rounds the significand to 24 bits (single precision).
	`64`	Rounds the significand to 53 bits (double precision).
	`80`	Rounds the significand to 64 bits (extended precision).

Default

`-pc80` or `/Qpc64`	On Linux* systems, the floating-point significand is rounded to 64 bits. On Windows* systems, the floating-point significand is rounded to 53 bits.

Description

This option enables control of floating-point significand precision.

Some floating-point algorithms are sensitive to the accuracy of the significand, or fractional part of the floating-point value. For example, iterative operations like division and finding the square root can run faster if you lower the precision with this option.

Note that a change of the default precision control or rounding mode, for example, by using the `[Q]pc32` option or by user intervention, may affect the results returned by some of the mathematical functions.

IDE Equivalent

None

Alternate Options

None

Inlining Options

This section contains descriptions for compiler options that pertain to inlining. They are listed in alphabetical order.

fgnu89-inline

Tells the compiler to use C89 semantics for inline functions when in C99 mode.

Syntax

Linux OS:

`-fgnu89-inline`

Windows OS:

`None`

Arguments

None

Default

OFF

Description

This option tells the compiler to use C89 semantics for inline functions when in C99 mode.

IDE Equivalent

None

Alternate Options

None

finline

Tells the compiler to inline functions declared with __inline and perform C++ inlining.

Syntax

Linux OS:

`-finline`

`-fno-inline`

Windows OS:

None

Arguments

None

Default

`-fno-inline`	The compiler does not inline functions declared with `__inline`.

Description

This option tells the compiler to inline functions declared with `__inline` and perform C++ inlining.

IDE Equivalent

None

Alternate Options

None

finline-functions

Enables function inlining for single file compilation.

Syntax

Linux OS:

`-finline-functions`

`-fno-inline-functions`

Windows OS:

`None`

Arguments

None

Default

`-finline-functions`	Interprocedural optimizations occur. However, if you specify `-O0`, the default is OFF.

Description

This option enables function inlining for single file compilation.

It enables the compiler to perform inline function expansion for calls to functions defined within the current source file.

The compiler applies a heuristic to perform the function expansion.

IDE Equivalent

None

Alternate Options

None

Output, Debug, and Precompiled Header Options

This section contains descriptions for compiler options that pertain to output, debugging, or precompiled headers (PCH). They are listed in alphabetical order.

c

Prevents linking.

Syntax

Linux OS:

`-c`

Windows OS:

`/c`

Arguments

None

Default

OFF	Linking is performed.

Description

This option prevents linking. Compilation stops after the object file is generated.

The compiler generates an object file for each C or C++ source file or preprocessed source file. It also takes an assembler file and invokes the assembler to generate an object file.

IDE Equivalent

None

Alternate Options

None

debug (Linux*)

Enables or disables generation of debugging information.

Syntax

Linux OS:

`-debug [`*`keyword`*`]`

Windows OS:

`None`

Arguments

keyword — Is the type of debugging information to be generated. Possible values are:

`none`	Disables generation of debugging information.
`full` or `all`	Generates complete debugging information.
`minimal`	Generates line number information for debugging.
`[no]emit_column`	Determines whether the compiler generates column number information for debugging.
`[no]expr-source-pos`	Determines whether the compiler generates source position information at the expression level of granularity.
`[no]inline-debug-info`	Determines whether the compiler generates enhanced debug information for inlined code.
`[no]pubnames`	Determines whether the compiler generates a DWARF debug_pubnames section.
`[no]semantic-stepping`	Determines whether the compiler generates enhanced debug information useful for breakpoints and stepping.
`[no]variable-locations`	Determines whether the compiler generates enhanced debug information useful in finding scalar local variables.
`extended`	Generates complete debugging information and also sets keyword values `semantic-stepping` and `variable-locations`.
`[no]parallel` (Linux only)	Determines whether the compiler generates parallel debug code instrumentations useful for thread data sharing and reentrant call detection.

For information on the non-default settings for these keywords, see the Description section.

Default

varies	Normally, the default is `-debug none` and no debugging information is generated. However, on Linux*, the `-debug inline-debug-info` option will be enabled by default if you compile with optimizations (option `-O2` or higher) and debugging is enabled (option `-g`).

Description

This option enables or disables generation of debugging information.

By default, enabling debugging, will disable optimization. To enable both debugging and optimization use the `-debug` option together with one of the optimization level options (`-O3`, `-O2` or `-O3`).

Keywords `semantic-stepping`, `inline-debug-info`, `variable-locations`, and `extended` can be used in combination with each other. If conflicting keywords are used in combination, the last one specified on the command line has precedence.

Option	Description
`-debug none`	Disables generation of debugging information.
`-debug full` or `-debug all`	Generates complete debugging information. It is the same as specifying `-debug` with no keyword.
`-debug minimal`	Generates line number information for debugging.
`-debug emit_column`	Generates column number information for debugging.
`-debug expr-source-pos`	Generates source position information at the statement level of granularity.
`-debug inline-debug-info`	Generates enhanced debug information for inlined code. On inlined functions, symbols are (by default) associated with the caller. This option causes symbols for inlined functions to be associated with the source of the called function.
`-debug pubnames`	The compiler generates a DWARF debug_pubnames section. This provides a means to list the names of global objects and functions in a compilation unit.
`-debug semantic-stepping`	Generates enhanced debug information useful for breakpoints and stepping. It tells the debugger to stop only at machine instructions that achieve the final effect of a source statement. For example, in the case of an assignment statement, this might be a store instruction that assigns a value to a program variable; for a function call, it might be the machine instruction that executes the call. Other instructions generated for those source statements are not displayed during stepping. This option has no impact unless optimizations have also been enabled.
`-debug variable-locations`	Generates enhanced debug information useful in finding scalar local variables. It uses a feature of the Dwarf object module known as "location lists".

Option	Description
	This feature allows the run-time locations of local scalar variables to be specified more accurately; that is, whether, at a given position in the code, a variable value is found in memory or a machine register.
`-debug extended`	Sets keyword values `semantic-stepping` and `variable-locations`. It also tells the compiler to include column numbers in the line information.
	Generates complete debugging information and also sets keyword values `semantic-stepping` and `variable-locations`. This is a more powerful setting than `-debug full` or `-debug all`.
`-debug parallel`	Generates parallel debug code instrumentations needed for the thread data sharing and reentrant call detection.
	This content does not apply to SYCL.
	For shared data and reentrancy detection, option `-qopenmp` must be set.

On Linux* systems, debuggers read debug information from executable images. As a result, information is written to object files and then added to the executable by the linker.

IDE Equivalent

Windows

Visual Studio: None

Linux

Eclipse: **Advanced Debugging > Enable Parallel Debug Checks** (`-debug parallel`)

Debug > Enable Expanded Line Number Information (`-debug expr-source-pos`)

Alternate Options

For `-debug full`, `-debug all`, or `-debug`	Linux: `-g` Windows: `/debug:full`, `/debug:all`, or `/debug`

See Also

`debug` (Windows*) compiler option
`qopenmp, Qopenmp` compiler option

debug (Windows*)

Enables or disables generation of debugging information.

Syntax

Linux OS:

`None`

Windows OS:

`/debug[:`*`keyword`*`]`

Arguments

keyword Is the type of debugging information to be generated. Possible values are:

`none`	Disables generation of debugging information.
`full` or `all`	Generates complete debugging information.
`minimal`	Generates line number information for debugging.
`partial`	Deprecated. Generates global symbol table information needed for linking.
`[no]expr-source-pos`	Determines whether the compiler generates source position information at the expression level of granularity.
`[no]inline-debug-info`	Determines whether the compiler generates enhanced debug information for inlined code.

For information on the non-default settings for these keywords, see the Description section.

Default

`/debug:none`	This is the default on the command line and for a release configuration in the IDE.
`/debug:all`	This is the default for a debug configuration in the IDE.

Description

This option enables or disables generation of debugging information. It is passed to the linker.

By default, enabling debugging, will disable optimization. To enable both debugging and optimization use the `/debug` option together with one of the optimization level options (`/O3`, `/O2` or `/O3`).

If conflicting keywords are used in combination, the last one specified on the command line has precedence.

Option	**Description**
`/debug:none`	Disables generation of debugging information.
`/debug:full` or `/debug:all`	Generates complete debugging information. It produces symbol table information needed for full symbolic debugging of unoptimized code and global symbol information needed for linking. It is the same as specifying `/debug` with no keyword.
`/debug:minimal`	Generates line number information for debugging.
`/debug:partial`	Generates global symbol table information needed for linking, but not local symbol table information needed for debugging. This option is deprecated and is not available in the IDE.
`/debug:expr-source-pos`	Generates source position information at the statement level of granularity.
`/debug:inline-debug-info`	Generates enhanced debug information for inlined code. On inlined functions, symbols are (by default) associated with the caller. This option causes symbols for inlined functions to be associated with the source of the called function.

IDE Equivalent

Windows

Visual Studio: **Debugging > Enable Expanded Line Number Information** (/debug:expr-source-pos)

Linux

Eclipse: None

Alternate Options

For /debug:all or /debug	Linux: None Windows: /Zi

See Also

debug (Linux*) compiler option

Fa

Specifies that an assembly listing file should be generated.

Syntax

Linux OS:

-Fa[*filename*|*dir*]

Windows OS:

/Fa[*filename*|*dir*]

Arguments

filename	Is the name of the assembly listing file.
dir	Is the directory where the file should be placed. It can include *filename*.

Default

OFF	No assembly listing file is produced.

Description

This option specifies that an assembly listing file should be generated (optionally named *filename*).

NOTE
This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

Windows

Visual Studio: **Output Files > ASM List Location**

Linux

Eclipse: **Output > Generate Assembler Source and Binary Files**

Alternate Options

Linux: `-S`

Windows: `/S`

fasm-blocks

Enables the use of blocks and entire functions of assembly code within a C or C++ file.

Syntax

Linux OS:

`-fasm-blocks`

Windows OS:

`None`

Arguments

None

Default

OFF	The compiler allows a GNU*-style inline assembly format.

Description

This option enables the use of blocks and entire functions of assembly code within a C or C++ file.

It allows a Microsoft* MASM-style inline assembly block not a GNU*-style inline assembly block.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

`-use-msasm`

FD

Generates file dependencies related to the Microsoft C/C++ compiler.*

Syntax

Linux OS:

`None`

Windows OS:

`/FD`

Arguments

None

Default

OFF	The compiler does not generate Microsoft C/C++-related file dependencies.

Description

This option generates file dependencies related to the Microsoft* C/C++ compiler. It invokes the Microsoft C/C++ compiler and passes the option to it.

IDE Equivalent

None

Alternate Options

None

Fe

Specifies the name for a built program or dynamic-link library.

Syntax

Linux OS:

`None`

Windows OS:

`/Fe[[:]`*`filename|dir`*`]`

Arguments

filename	Is the name for the built program or dynamic-link library.
dir	Is the directory where the built program or dynamic-link library should be placed. It can include *file*.

Default

OFF	The name of the file is the name of the first source file on the command line with file extension `.exe`, so `file.f` becomes `file.exe`.

Description

This option specifies the name for a built program (`.EXE`) or a dynamic-link library (`.DLL`).

You can use this option to specify an alternate name for an executable file. This is especially useful when compiling and linking a set of input files. You can use the option to give the resulting file a name other than that of the first input file (source or object) on the command line.

NOTE
This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

Linux: `-o`

Windows: None

Example

In the following example, the command produces an executable file named outfile.exe as a result of compiling and linking three files: one object file and two C++ source files.

```
prompt> icx /Feoutfile.exe file1.obj file2.cpp file3.cpp
```

This command produces an executable file named file1.exe when the `/Fe` option is not used.

See Also

- compiler option

Fo

Specifies the name for an object file.

Syntax

Linux OS:

`See option o.`

Windows OS:

`/Fo[[:]`*`filename`*`|`*`dir`*`]`

Arguments

filename	Is the name for the object file.
dir	Is the directory where the object file should be placed. It can include *filename*.

Default

OFF	An object file has the same name as the name of the first source file and a file extension of .obj.

Description

This option specifies the name for an object file.

NOTE
This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

Windows

Visual Studio: **Output Files > Object File Name**

Alternate Options

None

See Also

- compiler option

Fp

Lets you specify an alternate path or file name for precompiled header files.

Syntax

Linux OS:

`None`

Windows OS:

`/Fp{filename|dir}`

Arguments

filename	Is the name for the precompiled header file.
dir	Is the directory where the precompiled header file should be placed. It can include *filename*.

Default

OFF	The compiler does not create or use precompiled headers unless you tell it to do so.

Description

This option lets you specify an alternate path or file name for precompiled header files.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

Windows

Visual Studio: **Precompiled Headers > Precompiled Header Output File**

Linux

Eclipse: None

Alternate Options

None

ftrapuv, Qtrapuv

Initializes stack local variables to an unusual value to aid error detection.

Syntax

Linux OS:

`-ftrapuv`

Windows OS:

`/Qtrapuv (C++ only)`

Windows OS:

`None (SYCL only)`

Arguments

None

Default

OFF	The compiler does not initialize local variables.

Description

This option initializes stack local variables to an unusual value to aid error detection. Normally, these local variables should be initialized in the application. It also unmasks the floating-point invalid exception.

The option sets any uninitialized local variables that are allocated on the stack to a value that is typically interpreted as a very large integer or an invalid address. References to these variables are then likely to cause run-time errors that can help you detect coding errors.

This option sets option `-g` (Linux*) and `/Zi` or `/Z7` (Windows*), which changes the default optimization level from `O2` to `-O0` (Linux) or `/Od` (Windows). You can override this effect by explicitly specifying an `O` option setting.

For more details on using options `-ftrapuv` and `/Qtrapuv` (C++) with compiler option `O`, see the article titled Don't optimize when using -ftrapuv for uninitialized variable detection.

Another way to detect uninitialized local scalar variables is by specifying keyword `uninit` for option `check`.

IDE Equivalent

Windows

Visual Studio: None

Linux

Eclipse: **Run-Time > Initialize Stack Variables to an Unusual Value**

Alternate Options

None

See Also

g compiler option
Zi, Z7, ZI compiler option
O compiler option

fverbose-asm

Produces an assembly listing with compiler comments, including options and version information.

Syntax

Linux OS:

`-fverbose-asm`

`-fno-verbose-asm`

Windows OS:

`None`

Arguments

None

Default

`-fno-verbose-asm`	No source code annotations appear in the assembly listing file, if one is produced.

Description

This option produces an assembly listing file with compiler comments, including options and version information.

To use this option, you must also specify `-S`, which sets `-fverbose-asm`.

If you do not want this default when you specify `-S`, specify `-fno-verbose-asm`.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

None

See Also

`S` compiler option

g

Tells the compiler to generate a level of debugging information in the object file.

Syntax

Linux OS:

`-g[`*n*`]`

Windows OS:

See option Zi, Z7, ZI.

Arguments

n	Is the level of debugging information to be generated. Possible values are:

0	Disables generation of symbolic debug information.
1	Produces minimal debug information for performing stack traces.
2	Produces complete debug information. This is the same as specifying `-g` with no *n*.
3	Produces extra information that may be useful for some tools.

Default

-g or -g2	The compiler produces complete debug information.

Description

Option `-g` tells the compiler to generate symbolic debugging information in the object file, which increases the size of the object file.

The compiler does not support the generation of debugging information in assemblable files. If you specify this option, the resulting object file will contain debugging information, but the assemblable file will not.

This option turns off option `-O2` and makes option `-O0` the default unless option `-O2` (or higher) is explicitly specified in the same command line.

Specifying the `-g` or `-O0` option sets the `-fno-omit-frame-pointer` option.

Linux

For C++, the -debug inline-debug-info option will be enabled by default if you compile with optimizations (option `-O2` or higher) and debugging is enabled with option `-g`.

> **NOTE**
> When option `-g` is specified, debugging information is generated in the DWARF Version 3 format. Older versions of some analysis tools may require applications to be built with the `-gdwarf-2` option to ensure correct operation.

IDE Equivalent

Visual Studio

Visual Studio: None

Eclipse

Eclipse: **General > Include Debug Information**

Alternate Options

Linux: None

Windows: /Zi, /Z7, /ZI

See Also

gdwarf compiler option
Zi, Z7, ZI compiler option
debug (Linux*) compiler option

gdwarf

Lets you specify a DWARF Version format when generating debug information.

Syntax

Linux OS:

`-gdwarf-n`

Windows OS:

None

Arguments

n	Is a value denoting the DWARF Version format to use. Possible values are:	
	2	Generates debug information using the DWARF Version 2 format.
	3	Generates debug information using the DWARF Version 3 format.
	4	Generates debug information using the DWARF Version 4 format. This setting is only available on Linux*.

Default

OFF	No debug information is generated. However, if compiler option `-g` is specified, debugging information is generated in the DWARF Version 3 format.

Description

This option lets you specify a DWARF Version format when generating debug information.

Note that older versions of some analysis tools may require applications to be built with the -gdwarf-2 option to ensure correct operation.

IDE Equivalent

None

Alternate Options

None

See Also

`g` compiler option

grecord-gcc-switches

Causes the command line options that were used to invoke the compiler to be appended to the DW_AT_producer attribute in DWARF debugging information.

Syntax

Linux OS:

`-grecord-gcc-switches`

Windows OS:

`None`

Arguments

None

Default

OFF	The command line options that were used to invoke the compiler are not appended to the DW_AT_producer attribute in DWARF debugging information.

Description

This option causes the command line options that were used to invoke the compiler to be appended to the DW_AT_producer attribute in DWARF debugging information.

The options are concatenated with whitespace separating them from each other and from the compiler version.

IDE Equivalent

None

Alternate Options

None

gsplit-dwarf

Creates a separate object file containing DWARF debug information.

Syntax

Linux OS:

`-gsplit-dwarf`

Windows OS:

`None`

Arguments

None

Default

OFF	No separate object file containing DWARF debug information is created.

Description

This option creates a separate object file containing DWARF debug information. It causes debug information to be split between the generated object (.o) file and the new DWARF object (.dwo) file.

The DWARF object file is not used by the linker, so this reduces the amount of debug information the linker must process and it results in a smaller executable file.

For this option to perform correctly, you must use binutils-2.24 or higher. To debug the resulting executable, you must use gdb-7.6.1 or higher.

NOTE
If you use the split executable with a tool that does not support the split DWARF format, it will behave as though the DWARF debug information is absent.

NOTE
This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

None

o

Specifies the name for an output file.

Syntax

Linux OS:

`-o` *`filename`*

Windows OS:

`See option Fo.`

Arguments

filename	Is the name for the output file. The space before *filename* is optional.

Default

OFF	The compiler uses the default file name for an output file.

Description

This option specifies the name for an output file as follows:

- If `-c` is specified, it specifies the name of the generated object file.
- If `-S` is specified, it specifies the name of the generated assembly listing file.
- If `-P` is specified, it specifies the name of the generated preprocessor file.

Otherwise, it specifies the name of the executable file.

IDE Equivalent

None

Alternate Options

Linux: None

Windows: `/Fe`

See Also

`Fo` compiler option
`Fe` compiler option

RTC

Enables checking for certain run-time conditions.

Syntax

Linux OS:

`None`

Windows OS:

`/RTC`*`option`*

Arguments

option	Specifies the condition to check. Possible values are 1, s, u, or c.

Default

OFF	No checking is performed for these run-time conditions.

Description

This option enables checking for certain run-time conditions. Using the `/RTC` option sets `__MSVC_RUNTIME_CHECKS` = 1.

Option	Description
`/RTC1`	This is the same as specifying `/RTCsu`.
`/RTCs`	Enables run-time checks of the stack frame.
`/RTCu`	Enables run-time checks for unintialized variables.
`/RTCc`	Enables checks for converting to smaller types.

IDE Equivalent

Windows

Visual Studio: **Code Generation > Basic Runtime Checks / Smaller Type Check**

Linux

Eclipse: None

Alternate Options

None

S

Causes the compiler to compile to an assembly file only and not link.

Syntax

Linux OS:

`-S`

Windows OS:

`/S`

Arguments

None

Default

OFF	Normal compilation and linking occur.

Description

This option causes the compiler to compile to an assembly file only and not link.

On Linux* systems, the assembly file name has a .s suffix. On Windows* systems, the assembly file name has an .asm suffix.

IDE Equivalent

Windows

Visual Studio: None

Linux

Eclipse: **Output Files > Generate Assembler Source File**

Alternate Options

Linux: None

Windows: `/Fa`

See Also

Fa compiler option

use-msasm

Enables the use of blocks and entire functions of assembly code within a C or C++ file.

Syntax

Linux OS:

`-use-msasm`

Windows OS:

`None`

Arguments

None

Default

OFF	The compiler allows a GNU*-style inline assembly format.

Description

This option enables the use of blocks and entire functions of assembly code within a C or C++ file.

It allows a Microsoft* MASM-style inline assembly block not a GNU*-style inline assembly block.

IDE Equivalent

None

Alternate Options

`-fasm-blocks`

Y-

Tells the compiler to ignore all other precompiled header files.

Syntax

Linux OS:

`None`

Windows OS:

`/Y-`

Arguments

None

Default

OFF	The compiler recognizes precompiled header files when certain compiler options are specified.

Description

This option tells the compiler to ignore all other precompiled header files.

NOTE
This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

None

See Also

`Yc` compiler option
`Yu` compiler option

Yc

Tells the compiler to create a precompiled header file.

Syntax

Linux OS:

`None`

Windows OS:

`/Yc[`*`filename`*`]`

Arguments

filename	Is the name of a C/C++ header file, which is included in the source file using an #include preprocessor directive.

Default

OFF	The compiler does not create or use precompiled headers unless you tell it to do so.

Description

This option tells the compiler to create a precompiled header (PCH) file. It is supported only for single source file compilations.

When *filename* is specified, the compiler creates a precompiled header file from the headers in the C/C++ program up to and including the C/C++ header specified.

If you do not specify *filename*, the compiler compiles all code up to the end of the source file, or to the point in the source file where a hdrstop occurs. The default name for the resulting file is the name of the source file with extension .pch.

This option cannot be used in the same compilation as the `/Yu` option.

NOTE
This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

Windows

Visual Studio: **Precompiled Headers > Precompiled Header File**

Linux

Eclipse: None

Alternate Options

None

Example

If option /Fp is used, it names the PCH file. Consider the following example command:

```
icx /c /Ycheader.h /Fpprecomp foo.cpp
icx /c /Yc /Fpprecomp foo.cpp
```

The name of the PCH file is `precomp.pch`.

If the header file name is specified, the file name is based on the header file name. For example:

```
icx /c /Ycheader.h foo.cpp
```

The name of the PCH file is `header.pch`.

If no header file name is specified, the file name is based on the source file name. For example:

```
icx /c /Yc foo.cpp
```

The name of the PCH file is `foo.pch`.

See Also

`Yu` compiler option
`Fp` compiler option

Yu

Tells the compiler to use a precompiled header file.

Syntax

Linux OS:

`None`

Windows OS:

`/Yu[`*`filename`*`]`

Arguments

filename	Is the name of a C/C++ header file, which is included in the source file using an #include preprocessor directive.

Default

OFF	The compiler does not use precompiled header files unless it is told to do so.

Description

This option tells the compiler to use a precompiled header (PCH) file.

It is supported for multiple source files when all source files use the same .pch file.

The compiler treats all code occurring before the header file as precompiled. It skips to just beyond the #include directive associated with the header file, uses the code contained in the PCH file, and then compiles all code after *filename*.

If you do not specify *filename*, the compiler will use a PCH with a name based on the source file name. If you specify option `/Fp`, it will use the PCH specified by that option.

When this option is specified, the compiler ignores all text, including declarations preceding the #include statement of the specified file.

This option cannot be used in the same compilation as the `/Yc` option.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

Windows

Visual Studio: **Precompiled Headers > Precompiled Header**

Linux

Eclipse: None

Alternate Options

None

Example

Consider the following example command:

```
icx /c /Yuheader.h bar.cpp
```

The name of the PCH file used is `header.pch`.

In the following command line, no filename is specified:

```
icx /Yu bar.cpp
```

The name of the PCH file used is `bar.pch`.

In the following command line, no filename is specified, but option `/Fp` is specified:

```
icx /Yu /Fpprecomp bar.cpp
```

The name of the PCH file used is `precomp.pch`.

See Also

`Yc` compiler option

Zi, Z7, ZI

Tells the compiler to generate full debugging information in either an object (.obj) file or a project database (PDB) file.

Syntax

Linux OS:

`See option g.`

Windows OS:

`/Zi`

`/Z7`

`/ZI`

Arguments

None

Default

OFF	No debugging information is produced.

Description

Option `/Z7` tells the compiler to generate symbolic debugging information in the object (.obj) file for use with the debugger. No .pdb file is produced by the compiler.

Option `/ZI` is a synonym for option `/Zi`.

The `/Zi` option tells the compiler to generate symbolic debugging information in a program database (PDB) file for use with the debugger. Type information is placed in the .pdb file, and not in the .obj file, resulting in smaller object files in comparison to option `/Z7`.

When option `/Zi` is specified, two PDB files are created:

- The compiler creates the program database project.pdb. If you compile a file without a project, the compiler creates a database named vc*x*0.pdb, where *x* represents the major version of Visual C++, for example vc140.pdb.

  This file stores all debugging information for the individual object files and resides in the same directory as the project makefile. If you want to change this name, use option `/Fd`.
- The linker creates the program database executablename.pdb.

  This file stores all debug information for the .exe file and resides in the debug subdirectory. It contains full debug information, including function prototypes, not just the type information found in vc*x*0.pdb.

Both PDB files allow incremental updates. The linker also embeds the path to the .pdb file in the .exe or .dll file that it creates.

The compiler does not support the generation of debugging information in assemblable files. If you specify these options, the resulting object file will contain debugging information, but the assemblable file will not.

These options turn off option `/O2` and make option `/Od` the default unless option `/O2` (or higher) is explicitly specified in the same command line.

For more information about the `/Z7`, `/Zi`, and `/ZI` options, see the Microsoft documentation.

IDE Equivalent

Visual Studio

Visual Studio: **General > Generate Debug Information**

Eclipse

Eclipse: None

Alternate Options

Linux: -g

Windows: None

See Also

g compiler option
debug (Windows*) compiler option

Preprocessor Options

This section contains descriptions for compiler options that pertain to preprocessing. They are listed in alphabetical order.

B

Specifies a directory that can be used to find include files, libraries, and executables.

Syntax

Linux OS:

`-Bdir`

Windows OS:

`None`

Arguments

dir	Is the directory to be used. If necessary, the compiler adds a directory separator character at the end of *dir*.

Default

OFF	The compiler looks for files in the directories specified in your PATH environment variable.

Description

This option specifies a directory that can be used to find include files, libraries, and executables.

The compiler uses *dir* as a prefix.

For include files, the *dir* is converted to `-I/`*dir*/include. This command is added to the front of the includes passed to the preprocessor.

For libraries, the *dir* is converted to `-L/`*dir*. This command is added to the front of the standard `-L` inclusions before system libraries are added.

For executables, if *dir* contains the name of a tool, such as `ld` or `as`, the compiler will use it instead of those found in the default directories.

The compiler looks for include files in *dir* /include while library files are looked for in *dir*.

On Linux* systems, another way to get the behavior of this option is to use the environment variable `GCC_EXEC_PREFIX`.

IDE Equivalent

None

Alternate Options

None

C

Places comments in preprocessed source output.

Syntax

Linux OS:

`-C`

Windows OS:

`/C`

Arguments

None

Default

OFF	No comments are placed in preprocessed source output.

Description

This option places (or preserves) comments in preprocessed source output.

Comments following preprocessing directives, however, are not preserved.

IDE Equivalent

Windows

Visual Studio: **Preprocessor > Keep Comments**

Linux

Eclipse: None

Alternate Options

None

Example

The following commands cause the compiler to preserve comments in the prog1.i preprocessed file.

Linux

```
icpx -C -P prog1.cpp prog2.cpp
```

Windows

```
icx /C /P prog1.cpp prog2.cpp
```

D

Defines a macro name that can be associated with an optional value.

Syntax

Linux OS:

-D*name*[=*value*]

Windows OS:

/D*name*[=*value*]

Arguments

name	Is the name of the macro.
value	Is an optional integer or an optional character string delimited by double quotes; for example, D*name*=*string*.

Default

OFF	Only default symbols or macros are defined.

Description

Defines a macro name that can be associated with an optional value. This option is equivalent to a #define preprocessor directive.

If a *value* is not specified, *name* is defined as "1".

IDE Equivalent

Windows

Visual Studio: **Preprocessor > Preprocessor Definitions**

Linux

Eclipse: **Preprocessor > Preprocessor Definitions**

Alternate Options

None

Example

To define a macro called SIZE with the value 100, enter the following command:

Linux

```
icpx -DSIZE=100 prog1.cpp
```

Windows

```
icx /DSIZE=100 prog1.cpp
```

If you define a macro, but do not assign a value, the compiler defaults to 1 for the value of the macro.

See Also

Additional Predefined Macros

dD, QdD

Same as option -dM, but outputs #define directives in preprocessed source.

Syntax

Linux OS:

`-dD`

Windows OS:

`/QdD`

Arguments

None

Default

OFF	The compiler does not output #define directives.

Description

Same as `-dM`, but outputs `#define` directives in preprocessed source. To use this option, you must also specify the `E` option.

IDE Equivalent

None

Alternate Options

None

dM, QdM

Tells the compiler to output macro definitions in effect after preprocessing.

Syntax

Linux OS:

`-dM`

Windows OS:

`/QdM`

Arguments

None

Default

OFF	The compiler does not output macro definitions after preprocessing.

Description

This option tells the compiler to output macro definitions in effect after preprocessing. To use this option, you must also specify option `E`.

IDE Equivalent

None

Alternate Options

None

See Also

`E` compiler option

E

Causes the preprocessor to send output to stdout.

Syntax

Linux OS:

`-E`

Windows OS:

`/E`

Arguments

None

Default

OFF	Preprocessed source files are output to the compiler.

Description

This option causes the preprocessor to send output to `stdout`. Compilation stops when the files have been preprocessed.

When you specify this option, the compiler's preprocessor expands your source module and writes the result to `stdout`. The preprocessed source contains #line directives, which the compiler uses to determine the source file and line number.

IDE Equivalent

None

Alternate Options

None

Example

To preprocess two source files and write them to `stdout`, enter the following command:

Linux

```
icpx -E prog1.cpp prog2.cpp
```

Windows

```
icx /E prog1.cpp prog2.cpp
```

EP

Causes the preprocessor to send output to stdout, omitting #line directives.

Syntax

Linux OS:

-EP

Windows OS:

/EP

Arguments

None

Default

OFF	Preprocessed source files are output to the compiler.

Description

This option causes the preprocessor to send output to `stdout`, omitting #line directives.

If you also specify option `P` or Linux* option `F`, the preprocessor will write the results (without #line directives) to a file instead of `stdout`.

IDE Equivalent

Windows

Visual Studio: **Preprocessor > Preprocess Suppress Line Numbers**

Linux

Eclipse: None

Alternate Options

None

Example

To preprocess to `stdout` omitting #line directives, enter the following command:

Linux

```
icpx -EP prog1.cpp prog2.cpp
```

Windows

```
icx /EP prog1.cpp prog2.cpp
```

FI

Tells the preprocessor to include a specified file name as the header file.

Syntax

Linux OS:

`None`

Windows OS:

`/FI`*`filename`*

Arguments

filename	Is the file name to be included as the header file.

Default

OFF	The compiler uses default header files.

Description

This option tells the preprocessor to include a specified file name as the header file.

The file specified with `/FI` is included in the compilation before the first line of the primary source file.

IDE Equivalent

Windows

Visual Studio: **Advanced > Forced Include File**

Linux

Eclipse: None

Alternate Options

None

H, QH

Tells the compiler to display the include file order and continue compilation.

Syntax

Linux OS:

`-H`

Windows OS:

`/QH`

Arguments

None

Default

OFF	Compilation occurs as usual.

Description

This option tells the compiler to display the include file order and continue compilation.

IDE Equivalent

None

Alternate Options

None

I

Specifies an additional directory to search for include files.

Syntax

Linux OS:

`-Idir`

Windows OS:

`/Idir`

Arguments

dir	Is the additional directory for the search.

Default

OFF	The default directory is searched for include files.

Description

This option specifies an additional directory to search for include files. To specify multiple directories on the command line, repeat the include option for each directory.

IDE Equivalent

Windows

Visual Studio: **General > Additional Include Directories**

Linux

Eclipse: **Preprocessor > Additional Include Directories**

Alternate Options

None

I-

Splits the include path.

Syntax

Linux OS:

`-I-`

Windows OS:

`/I-`

Arguments

None

Default

OFF	The default directory is searched for include files.

Description

This option splits the include path. It prevents the use of the current directory as the first search directory for '#include "file"'.

If you specify directories using the `I` option *before* you specify option `I-`, the directories are searched only for the case of '#include "file"'; they are not searched for '#include <file>'.

If you specify directories using the `I` option *after* you specify option `I-`, these directories are searched for all '#include' directives.

This option has no effect on option `nostdinc++`, which searches the standard system directories for header files.

This option is provided for compatibility with gcc.

IDE Equivalent

None

Alternate Options

None

See Also

`I` compiler option

`nostdinc++` compiler option

idirafter

Adds a directory to the second include file search path.

Syntax

Linux OS:

`-idirafter`*`dir`*

Windows OS:

`None`

Arguments

dir	Is the name of the directory to add.

Default

OFF	Include file search paths include certain default directories.

Description

This option adds a directory to the second include file search path (after `-I`).

IDE Equivalent

None

Alternate Options

None

imacros

Allows a header to be specified that is included in front of the other headers in the translation unit.

Syntax

Linux OS:

`-imacros filename`

Windows OS:

`None`

Arguments

filename	Name of header file.

Default

OFF

Description

Allows a header to be specified that is included in front of the other headers in the translation unit.

IDE Equivalent

None

Alternate Options

None

iprefix

Lets you indicate the prefix for referencing directories that contain header files.

Syntax

Linux OS:

`-iprefix prefix`

Windows OS:

`None`

Arguments

prefix	Is the prefix to use.

Default

OFF	No prefix is included.

Description

Options for indicating the prefix for referencing directories containing header files. Use *prefix* with option `-iwithprefix` as a prefix.

IDE Equivalent

None

Alternate Options

None

iquote

Adds a directory to the front of the include file search path for files included with quotes but not brackets.

Syntax

Linux OS:

`-iquote` *`dir`*

Windows OS:

`None`

Arguments

dir	Is the name of the directory to add.

Default

OFF	The compiler does not add a directory to the front of the include file search path.

Description

Add directory to the front of the include file search path for files included with quotes but not brackets.

IDE Equivalent

None

Alternate Options

None

isystem

Specifies a directory to add to the start of the system include path.

Syntax

Linux OS:

`-isystemdir`

Windows OS:

None

Arguments

dir	Is the directory to add to the system include path.

Default

OFF	The default system include path is used.

Description

This option specifies a directory to add to the system include path. The compiler searches the specified directory for include files after it searches all directories specified by the `-I` compiler option but before it searches the standard system directories.

On Linux* systems, this option is provided for compatibility with gcc.

IDE Equivalent

None

Alternate Options

None

iwithprefix

Appends a directory to the prefix passed in by -iprefix and puts it on the include search path at the end of the include directories.

Syntax

Linux OS:

`-iwithprefixdir`

Windows OS:

None

Arguments

dir	Is the include directory.

Default

OFF

Description

This option appends a directory to the prefix passed in by `-iprefix` and puts it on the include search path at the end of the include directories.

IDE Equivalent

None

Alternate Options

None

iwithprefixbefore

Similar to -iwithprefix except the include directory is placed in the same place as -I command-line include directories.

Syntax

Linux OS:

`-iwithprefixbefore`*`dir`*

Windows OS:

`None`

Arguments

dir	Is the include directory.

Default

OFF

Description

Similar to `-iwithprefix` except the include directory is placed in the same place as `-I` command-line include directories.

IDE Equivalent

None

Alternate Options

None

Kc++, TP

Tells the compiler to process all source or unrecognized file types as C++ source files. This is a deprecated option that may be removed in a future release.

Syntax

Linux OS:

`-Kc++`

Windows OS:

`/TP`

Arguments

None

Default

OFF	The compiler uses default rules for determining whether a file is a C++ source file.

Description

This option tells the compiler to process all source or unrecognized file types as C++ source files.

This is a deprecated option that may be removed in a future release. The replacement option for `Kc++` is `-x c++`; the replacement option for `/TP` is `/Tp<file>`.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

Windows

Visual Studio: **Advanced > Compile As**

Linux

Eclipse: None

Alternate Options

Linux: `-x c++`

Windows: `/Tp`

M, QM

Tells the compiler to generate makefile dependency lines for each source file.

Syntax

Linux OS:

`-M`

Windows OS:

`/QM`

Arguments

None

Default

OFF	The compiler does not generate makefile dependency lines for each source file.

Description

This option tells the compiler to generate makefile dependency lines for each source file, based on the #include lines found in the source file.

IDE Equivalent

None

Alternate Options

None

MD, QMD

Preprocess and compile, generating output file containing dependency information ending with extension .d.

Syntax

Linux OS:

`-MD`

Windows OS:

`/QMD`

Arguments

None

Default

OFF	The compiler does not generate dependency information.

Description

Preprocess and compile, generating output file containing dependency information ending with extension .d.

IDE Equivalent

None

Alternate Options

None

MF, QMF

Tells the compiler to generate makefile dependency information in a file.

Syntax

Linux OS:

`-MF`*`filename`*

Windows OS:

`/QMF`*`filename`*

Arguments

filename	Is the name of the file where the makefile dependency information should be placed.

Default

OFF	The compiler does not generate makefile dependency information in files.

Description

This option tells the compiler to generate makefile dependency information in a file. To use this option, you must also specify `/QM` or `/QMM`.

IDE Equivalent

None

Alternate Options

None

See Also

`QM` compiler option

`QMM` compiler option

MG, QMG

Tells the compiler to generate makefile dependency lines for each source file.

Syntax

Linux OS:

`-MG`

Windows OS:

`/QMG`

Arguments

None

Default

OFF	The compiler does not generate makefile dependency information in files.

Description

This option tells the compiler to generate makefile dependency lines for each source file. It is similar to `/QM`, but it treats missing header files as generated files.

IDE Equivalent

None

Alternate Options

None

See Also

`QM` compiler option

MM, QMM

Tells the compiler to generate makefile dependency lines for each source file.

Syntax

Linux OS:

`-MM`

Windows OS:

`/QMM`

Arguments

None

Default

OFF	The compiler does not generate makefile dependency information in files.

Description

This option tells the compiler to generate makefile dependency lines for each source file. It is similar to `/QM`, but it does not include system header files.

IDE Equivalent

None

Alternate Options

None

See Also

`QM` compiler option

MMD, QMMD

Tells the compiler to generate an output file containing dependency information.

Syntax

Linux OS:

`-MMD`

Windows OS:

`/QMMD`

Arguments

None

Default

OFF	The compiler does not generate an output file containing dependency information.

Description

This option tells the compiler to preprocess and compile a file, then generate an output file (with extension `.d`) containing dependency information.

It is similar to `/QMD`, but it does not include system header files.

IDE Equivalent

None

Alternate Options

None

MQ

Changes the default target rule for dependency generation.

Syntax

Linux OS:

`-MQ`*`target`*

Windows OS:

`None`

Arguments

target	Is the target rule to use.

Default

OFF	The default target rule applies to dependency generation.

Description

This option changes the default target rule for dependency generation. It is similar to `-MT`, but quotes special Make characters.

IDE Equivalent

None

Alternate Options

None

MT, QMT

Changes the default target rule for dependency generation.

Syntax

Linux OS:

`-MTtarget`

Windows OS:

`/QMTtarget`

Arguments

target	Is the target rule to use.

Default

OFF	The default target rule applies to dependency generation.

Description

This option changes the default target rule for dependency generation.

IDE Equivalent

None

Alternate Options

None

nostdinc++

Do not search for header files in the standard directories for C++, but search the other standard directories.

Syntax

Linux OS:

`-nostdinc++`

Windows OS:

`None`

Arguments

None

Default

OFF

Description

Do not search for header files in the standard directories for C++, but search the other standard directories.

IDE Equivalent

None

Alternate Options

None

P

Tells the compiler to stop the compilation process and write the results to a file.

Syntax

Linux OS:

`-P`

Windows OS:

`/P`

Arguments

None

Default

OFF	Normal compilation is performed.

Description

This option tells the compiler to stop the compilation process after C or C++ source files have been preprocessed and write the results to files named according to the compiler's default file-naming conventions.

On Linux systems, this option causes the preprocessor to expand your source module and direct the output to a `.i` file instead of `stdout`. Unlike the `-E` option, the output from `-P` on Linux does not include #line number directives. By default, the preprocessor creates the name of the output file using the prefix of the source file name with a `.i` extension. You can change this by using the `-o` option.

IDE Equivalent

Windows

Visual Studio: **Preprocessor > Generate Preprocessed File**

Linux

Eclipse: None

Alternate Options

Linux: `-F`

Windows: None

pragma-optimization-level

Specifies which interpretation of the optimization_level pragma should be used if no prefix is specified.

Syntax

Linux OS:

`-pragma-optimization-level=`*`interpretation`*

Windows OS:

`None`

Arguments

interpretation	Compiler-specific interpretation of `optimization_level` pragma. Possible values are:
	`Intel` — Specify the Intel interpretation.
	`GCC` — Specify the GCC interpretation.

Default

`-pragma-optimization-level=Intel`	Use the Intel interpretation of the `optimization_level` pragma.

Description

Specifies which interpretation of the `optimization_level` pragma should be used if no prefix is specified.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

None

U

Undefines any definition currently in effect for the specified macro.

Syntax

Linux OS:

`-U`*`name`*

Windows OS:

/Uname

Arguments

name	Is the name of the macro to be undefined.

Default

OFF	Macro definitions are in effect until they are undefined.

Description

This option undefines any definition currently in effect for the specified macro. It is equivalent to an #undef preprocessing directive.

On Windows systems, use the `/u` option to undefine all previously defined preprocessor values.

IDE Equivalent

Windows

Visual Studio: **Preprocessor > Undefine Preprocessor Definitions**

Linux

Eclipse: **Preprocessor > Undefine Preprocessor Definitions**

Alternate Options

None

Example

To undefine a macro, enter the following command:

On Windows systems:

```
icx /Uia64 prog1.cpp
```

On Linux systems:

```
icpx -Uia64 prog1.cpp
```

If you attempt to undefine an ANSI C macro, the compiler will emit an error:

```
invalid macro undefinition: <name of macro>
```

See Also

undef

Disables all predefined macros.

Syntax

Linux OS:

`-undef`

Windows OS:

`None`

Arguments

None

Default

OFF	Defined macros are in effect until they are undefined.

Description

This option disables all predefined macros.

IDE Equivalent

None

Alternate Options

None

X

Removes standard directories from the include file search path.

Syntax

Linux OS:

`-X`

Windows OS:

`/X`

Arguments

None

Default

OFF	Standard directories are in the include file search path.

Description

This option removes standard directories from the include file search path. It prevents the compiler from searching the default path specified by the INCLUDE environment variable.

On Linux* systems, specifying `-X` (or `-noinclude`) prevents the compiler from searching in `/usr/include` for files specified in an INCLUDE statement.

You can use this option with the `I` option to prevent the compiler from searching the default path for include files and direct it to use an alternate path.

IDE Equivalent

Windows

Visual Studio: **Preprocessor > Ignore Standard Include Path**

Linux

Eclipse: **Preprocessor > Ignore Standard Include Path**

Alternate Options

Linux: `-nostdinc`

Windows: None

See Also

`I` compiler option

Component Control Options

This section contains descriptions for compiler options that pertain to component control. They are listed in alphabetical order.

Qoption

Passes options to a specified tool.

Syntax

Linux OS:

`-Qoption,`*`string`*`,`*`options`*

Windows OS:

`/Qoption,`*`string`*`,`*`options`*

Arguments

string	Is the name of the tool.
options	Are one or more comma-separated, valid options for the designated tool. Note that certain tools may require that options appear within quotation marks (" ").

Default

OFF	No options are passed to tools.

Description

This option passes options to a specified tool.

If an argument contains a space or tab character, you must enclose the entire argument in quotation marks (" "). You must separate multiple arguments with commas.

string can be any of the following:

- cpp - Indicates the preprocessor for the compiler.
- c - Indicates the Intel® oneAPI DPC++/C++ Compiler.
- asm - Indicates the assembler.
- link - Indicates the linker.
- prof - Indicates the profiler.
- On Windows* systems, the following is also available:
  - masm - Indicates the Microsoft assembler.
- On Linux* systems, the following are also available:

- as - Indicates the assembler.
- gas - Indicates the GNU assembler.
- ld - Indicates the loader.
- gld - Indicates the GNU loader.
- lib - Indicates an additional library.
- crt - Indicates the crt%.o files linked into executables to contain the place to start execution.

NOTE
This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

None

Language Options

This section contains descriptions for compiler options that pertain to language compatibility, conformity, etc.. They are listed in alphabetical order.

ansi

Enables language compatibility with the gcc option ansi.

Syntax

Linux OS:

`-ansi`

Windows OS:

`None`

Arguments

None

Default

OFF	GNU C++ is more strongly supported than ANSI C.

Description

This option enables language compatibility with the gcc option `-ansi` and provides the same level of ANSI standard conformance as that option.

This option sets option `fmath-errno`.

If you want strict ANSI conformance, use the -strict-ansi option.

IDE Equivalent

Windows

Visual Studio: None

Linux

Eclipse: **Language > ANSI Conformance**

Alternate Options

None

fno-gnu-keywords

Tells the compiler to not recognize typeof as a keyword.

Syntax

Linux OS:

`-fno-gnu-keywords`

Windows OS:

`None`

Arguments

None

Default

OFF	Keyword `typeof` is recognized.

Description

Tells the compiler to not recognize `typeof` as a keyword.

IDE Equivalent

None

Alternate Options

None

fno-operator-names

Disables support for the operator names specified in the standard.

Syntax

Linux OS:

`-fno-operator-names`

Windows OS:

`None`

Arguments

None

Default

OFF

Description

Disables support for the operator names specified in the standard.

IDE Equivalent

None

Alternate Options

None

fno-rtti

Disables support for run-time type information (RTTI).

Syntax

Linux OS:

`-fno-rtti`

Windows OS:

`None`

Arguments

None

Default

OFF	Support for run-time type information (RTTI) is enabled.

Description

This option disables support for run-time type information (RTTI).

IDE Equivalent

Windows

Visual Studio: None

Linux

Eclipse: None

Alternate Options

None

fpermissive

Tells the compiler to allow for non-conformant code.

Syntax

Linux OS:

`-fpermissive`

Windows OS:

`None`

Arguments

None

Default

OFF

Description

Tells the compiler to allow for non-conformant code.

IDE Equivalent

None

Alternate Options

None

fshort-enums

Tells the compiler to allocate as many bytes as needed for enumerated types.

Syntax

Linux OS:

`-fshort-enums`

Windows OS:

`None`

Arguments

None

Default

OFF	The compiler allocates a default number of bytes for enumerated types.

Description

This option tells the compiler to allocate as many bytes as needed for enumerated types.

IDE Equivalent

Windows

Visual Studio: None

Linux

Eclipse: **Data > Associate as Many Bytes as Needed for Enumerated Types**

Alternate Options

None

fsyntax-only

Tells the compiler to check only for correct syntax.

Syntax

Linux OS:

`-fsyntax-only`

Windows OS:

`None`

Arguments

None

Default

OFF	Normal compilation is performed.

Description

This option tells the compiler to check only for correct syntax. No object file is produced.

IDE Equivalent

None

Alternate Options

Linux: None

Windows: `/Zs`

funsigned-char

Change default char type to unsigned.

Syntax

Linux OS:

`-funsigned-char`

Windows OS:

`None`

Arguments

None

Default

OFF	Do not change default char type to unsigned.

Description

Change default char type to unsigned.

IDE Equivalent

Windows

Visual Studio: None

Linux

Eclipse: **Data > Change default char type to unsigned**

Alternate Options

None

J

Sets the default character type to unsigned.

Syntax

Linux OS:

`None`

Windows OS:

`/J`

Arguments

None

Default

OFF	The default character type is `signed`

Description

This option sets the default character type to `unsigned`. This option has no effect on character values that are explicitly declared `signed`. This option sets `_CHAR_UNSIGNED` = 1.

IDE Equivalent

Windows

Visual Studio: **Language > Default Char Unsigned**

Linux

Eclipse: None

Alternate Options

None

std, Qstd

Tells the compiler to conform to a specific language standard.

Syntax

Linux OS:

`-std=`*`val`*

Windows OS:

`/Qstd:`*`val`*

`/std:`*`val`* `(For Microsoft* compatibility)`

Arguments

val Specifies the specific language standard to conform to.

The following values apply to Linux* `-std` and Windows* `/Qstd`:

`c++2b`	Enables support for the Working Draft for ISO C++ 2023 DIS standard.
`c++20`	Enables support for the 2020 ISO C++ DIS standard.
`c++17`	Enables support for the 2017 ISO C++ standard with amendments.
`c++14`	Enables support for the 2014 ISO C++ standard with amendments.
`c++11`	Enables support for the 2011 ISO C++ standard with amendments.
`c++98` and `c++03`	Enables support for the 1998 ISO C++ standard with amendments.
`c2x`	Enables support for the Working Draft for ISO C2x standard.
`c18` and `c17`	Enables support for the 2017 ISO C standard. Support for c17 can also be enabled by value iso9899:2017. Support for c18 can also be enabled by value iso9899:2018.
`c11`	Enables support for the 2011 ISO C standard. Support for this standard can also be enabled by value iso9899:2011.
`c99`	Enables support for the 1999 ISO C standard. Support for this standard can also be enabled by value iso9899:1999.
`c90` and `c89`	Enables support for the 1990 ISO C standard. Support for this standard can also be enabled by value iso9899:1990.

The following values apply only to Linux `-std`:

`gnu++2b`	Enables support for the Working Draft for ISO C++ 2023 DIS standard plus GNU extensions.
`gnu++20`	Enables support for the 2020 ISO C++ DIS standard plus GNU extensions.
`gnu++17`	Enables support for the 2017 ISO C++ standard with amendments plus GNU extensions.
`gnu++14`	Enables support for the 2014 ISO C++ standard with amendments plus GNU extensions.
`gnu++11`	Enables support for the 2011 ISO C++ standard with amendments plus GNU extensions.
`gnu++98` and `gnu++03`	Enables support for the 1998 ISO C++ standard with amendments plus GNU extensions.
`gnu2x`	Enables support for the Working Draft for ISO C2x standard plus GNU extensions.
`gnu18` and `gnu17`	Enables support for the 2017 ISO C standard plus GNU extensions.
`gnu11`	Enables support for the 2011 ISO C standard plus GNU extensions.
`gnu99`	Enables support for the 1999 ISO C standard plus GNU extensions.
`gnu90` and `gnu89`	Enables support for the 1990 ISO C standard plus GNU extensions.

For possible values for Microsoft*-compatible Windows* /std, see the Microsoft* documentation.

Default

Default for Windows option `/Qstd`: OFF	The compiler does not conform to a specific language standard.
Default for Windows option `/std`: `c++14`	Currently, the compiler conforms to the 2014 ISO C++ standard. For the latest information, see the Microsoft* documentation.
Default for Linux option `-std` on dpcpp: `c++17`	The compiler conforms to the 2017 ISO C++ standard.
Default for Linux option `-std` on icx: `c++14`	The compiler conforms to the 2014 ISO C++ standard.

Description

This option tells the compiler to conform to a specific language standard.

IDE Equivalent

Visual Studio

Visual Studio: **Language > C/C++ Language Support**

Eclipse

Eclipse: **Language > ANSI Conformance**

Alternate Options

None

strict-ansi

Tells the compiler to implement strict ANSI conformance dialect.

Syntax

Linux OS:

`-strict-ansi`

Windows OS:

`None`

Arguments

None

Default

OFF	The compiler conforms to default standards.

Description

This option tells the compiler to implement strict ANSI conformance dialect. On Linux* systems, if you need to be compatible with gcc, use the `-ansi` option.

This option sets option `fmath-errno`, which tells the compiler to assume that the program tests errno after calls to math library functions. This restricts optimization because it causes the compiler to treat most math functions as having side effects.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

Windows

Visual Studio: None

Linux

Eclipse: **Language > ANSI Conformance**

Alternate Options

None

vd

Enables or suppresses hidden vtordisp members in C++ objects.

Syntax

Linux OS:

None

Windows OS:

/vd*n*

Arguments

n	Possible values are:	
	0	Suppresses the creation of the hidden `vtordisp` members in C++ objects.
	1	Enables the creation of hidden `vtordisp` members in C++ objects when they are necessary.
	2	Enables the hidden `vtordisp` members for all virtual base classes with virtual functions. This setting is recommended in the following cases: • When the only virtual function in your virtual base class is a destructor • When you want to ensure correct performance of the dynamic_cast operator on a partially-constructed object

Default

/vd1	The compiler enables the creation of hidden `vtordisp` members in C++ objects when they are necessary.

Description

This option enables or suppresses hidden `vtordisp` members in C++ objects.

This is a compatibility option for the Microsoft Visual C++* option `/vdn`. For full details about this compiler option, see the Microsoft* documentation.

IDE Equivalent

None

Alternate Options

None

vmg

Selects the general representation that the compiler uses for pointers to members.

Syntax

Linux OS:

None

Windows OS:

`/vmg`

Arguments

None

Default

OFF	The compiler uses default rules to represent pointers to members.

Description

This option selects the general representation that the compiler uses for pointers to members. Use this option if you declare a pointer to a member before you define the corresponding class.

IDE Equivalent

None

Alternate Options

None

x (type option)

All source files found subsequent to -x type will be recognized as a particular type.

Syntax

Linux OS:

`-x` *`type`*

Windows OS:

`None`

Arguments

type is the type of source file. Possible values are:

`c++`	C++ source file
`c++-header`	C++ header file
`c++-cpp-output`	C++ pre-processed file
`c`	C source file
`c-header`	C header file
`cpp-output`	C pre-processed file
`assembler`	Assembly file
`assembler-with-cpp`	Assembly file that needs to be preprocessed
`none`	Disable recognition, and revert to file extension

Default

`none`	Disable recognition and revert to file extension.

Description

All source files found subsequent to `-x`*type* will be recognized as a particular type.

IDE Equivalent

None

Alternate Options

None

Example

Suppose you want to compile the following C and C++ source files whose extensions are not recognized by the compiler:

File Name	Language
file1.c99	C
file2.cplusplus	C++

We will also include these files whose extensions are recognized:

File Name	Language
file3.c	C
file4.cpp	C++

The command-line invocation using the `-x` option follows:

```
icpx -x c file1.c99 -x c++ file2.cplusplus -x none file3.c file4.cpp
```

Zc

Lets you specify ANSI C standard conformance for certain language features.

Syntax

Linux OS:

`None`

Windows OS:

`/Zc:`*`arg1`*`[,`*`arg2`*`]`

Arguments

arg	Is the language feature for which you want standard conformance. The settings are compatible with Microsoft* settings for option /Zc. For a list of supported settings, see the table in the Description section of this topic.

Default

varies See the table in the Description section of this topic.

Description

This option lets you specify ANSI C standard conformance for certain language features.

If you do not want the default behavior for one or more of the settings, you must specify the negative form of the setting. For example, if you do not want the `threadSafeInit` or `sizedDealloc` default behavior, you should specify `/Zc:threadSafeInit-,sizedDealloc-`.

The following table shows the supported Microsoft settings for option `/Zc`.

`/Zc` setting name	Description
alignedNew[-]	Enables C++17 aligned allocation functions (default for C++17). Disabled by /Zc:alignedNew-.
char8_t[-]	Enables char8_t from C++2a. Disabled by /Zc:char8_t- (default).
dllexportInlines[-]	Enables dllexport/dllimport inline member functions of dllexport/import classes (default). Disabled by /Zc:dllexportInlines-.
sizedDealloc[-]	Enables C++14 sized global deallocation functions (default). Disabled by /Zc:sizedDealloc-
strictStrings[-]	Enforces const qualification for string literals. Disabled by /Zc:strictStrings- (default).
threadSafeInit[-]	Enables thread-safe initialization of local statics (default). Disabled by /Zc:threadSafeInit-.
trigraphs[-]	Enables trigraph character sequences. Disabled by /Zc:trigraphs- (default).
twoPhase[-]	Enables two-phase name lookup in templates. Disabled by /Zc:twoPhase- (default).

IDE Equivalent

Windows

Visual Studio: **Language > Treat wchar_t as Built-in Type / Force Conformance In For Loop Scope**

Language > Enforce type conversion rules (rvalueCast)

Linux

Eclipse: None

Alternate Options

None

Zg

Tells the compiler to generate function prototypes. This is a deprecated option that may be removed in a future release.

Syntax

Linux OS:

`None`

Windows OS:

`/Zg`

Arguments

None

Default

OFF	The compiler does not create function prototypes.

Description

This option tells the compiler to generate function prototypes.

This is a deprecated option that may be removed in a future release. There is no replacement option.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

None

Zp

Specifies alignment for structures on byte boundaries.

Syntax

Linux OS:

`-Zp[`*n*`]`

Windows OS:

`/Zp[`*n*`]`

Arguments

n	Is the byte size boundary. Possible values are 1, 2, 4, 8, or 16.

Default

`Zp16`	Structures are aligned on either size boundary 16 or the boundary that will naturally align them.

Description

This option specifies alignment for structures on byte boundaries.

If you do not specify *n*, you get `Zp16`.

IDE Equivalent

Windows

Visual Studio: **Code Generation > Struct Member Alignment**

Linux

Eclipse: **Data > Structure Member Alignment**

Alternate Options

None

Zs

Tells the compiler to check only for correct syntax.

Syntax

Linux OS:

`None`

Windows OS:

`/Zs`

Arguments

None

Default

OFF	Normal compilation is performed.

Description

This option tells the compiler to check only for correct syntax.

IDE Equivalent

None

Alternate Options

Linux: `-syntax`, `-fsyntax-only`

Windows: None

Data Options

This section contains descriptions for compiler options that pertain to the treatment of data. They are listed in alphabetical order.

align

Determines whether variables and arrays are naturally aligned.

Architecture Restrictions

Only available on IA-32 architecture

Syntax

Linux OS:

`-align`

`-noalign`

Windows OS:

`None`

Arguments

None

Default

`-noalign`	Variables and arrays are aligned according to the gcc model, which means they are aligned to 4-byte boundaries.

Description

This option determines whether variables and arrays are naturally aligned. Option `-align` forces the following natural alignment:

Type	Alignment
`double`	8 bytes
`long long`	8 bytes
`long double`	16 bytes

If you are not interacting with system libraries or other libraries that are compiled without `-align`, this option can improve performance by reducing misaligned accesses.

This option can also be specified as `-m[no-]align-double`. The options are equivalent.

> **Caution**
> If you are interacting with compatible libraries, this option can improve performance by reducing misaligned accesses. However, if you are interacting with noncompatible libraries or libraries that are compiled without option `-align`, your application may not perform as expected.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

None

fcommon

Determines whether the compiler treats common symbols as global definitions.

Syntax

Linux OS:

`-fcommon`

`-fno-common`

Windows OS:

`None`

Arguments

None

Default

`-fcommon`	The compiler does not treat common symbols as global definitions.

Description

This option determines whether the compiler treats common symbols as global definitions and to allocate memory for each symbol at compile time.

Option `-fno-common` tells the compiler to treat common symbols as global definitions. When using this option, you can only have a common variable declared in one module; otherwise, a link time error will occur for multiple defined symbols.

Normally, a file-scope declaration with no initializer and without the `extern` or `static` keyword "int i;" is represented as a common symbol. Such a symbol is treated as an external reference. However, if no other compilation unit has a global definition for the name, the linker allocates memory for it.

IDE Equivalent

Windows

Visual Studio: None

Linux

Eclipse: **Data > Allow gprel Addressing of Common Data Variables**

Alternate Options

None

fkeep-static-consts, Qkeep-static-consts

Tells the compiler to preserve allocation of variables that are not referenced in the source.

Syntax

Linux OS:

`-fkeep-static-consts`

`-fno-keep-static-consts`

Windows OS:

`/Qkeep-static-consts (C++ only)`

`/Qkeep-static-consts- (C++ only)`

Arguments

None

Default

`-fno-keep-static-consts` C++: or `/Qkeep-static-consts-`	If a variable is never referenced in a routine, the variable is discarded unless optimizations are disabled by option `-O0` (Linux*) or `/Od` (Windows*).

Description

This option tells the compiler to preserve allocation of variables that are not referenced in the source.

The negated form can be useful when optimizations are enabled to reduce the memory usage of static data.

IDE Equivalent

None

Alternate Options

None

fmath-errno

Tells the compiler that errno can be reliably tested after calls to standard math library functions.

Syntax

Linux OS:

`-fmath-errno`

`-fno-math-errno`

Windows OS:

`None`

Arguments

None

Default

`-fno-math-errno`	The compiler assumes that the program does not test `errno` after calls to standard math library functions.

Description

This option tells the compiler to assume that the program tests `errno` after calls to math library functions. This restricts optimization because it causes the compiler to treat most math functions as having side effects.

Option `-fno-math-errno` tells the compiler to assume that the program does not test `errno` after calls to math library functions. This frequently allows the compiler to generate faster code. Floating-point code that relies on IEEE exceptions instead of `errno` to detect errors can safely use this option to improve performance.

IDE Equivalent

None

Alternate Options

None

fpack-struct

Specifies that structure members should be packed together.

Syntax

Linux OS:

`-fpack-struct`

Windows OS:

`None`

Arguments

None

Default

OFF

Description

Specifies that structure members should be packed together.

NOTE
Using this option may result in code that is not usable with standard (system) c and C++ libraries.

IDE Equivalent

None

Alternate Options

Linux: `-Zp1`

Windows: None

fpascal-strings

Tells the compiler to allow for Pascal-style string literals.

Architecture Restrictions

Only available on IA-32 architecture

Syntax

Linux OS:

`-fpascal-strings`

Windows OS:

`None`

Arguments

None

Default

OFF	The compiler does not allow for Pascal-style string literals.

Description

Tells the compiler to allow for Pascal-style string literals.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

Windows

Visual Studio: None

Linux

Eclipse: None

Alternate Options

None

fpic

Determines whether the compiler generates position-independent code.

Syntax

Linux OS:

`-fpic`

`-fno-pic`

Windows OS:

`None`

Arguments

None

Default

`-fno-pic`	The compiler does not generate position-independent code.

Description

This option determines whether the compiler generates position-independent code.

Option `-fpic` specifies full symbol preemption. Global symbol definitions as well as global symbol references get default (that is, preemptable) visibility unless explicitly specified otherwise.

Option `-fpic` must be used when building shared objects.

This option can also be specified as `-fPIC`.

IDE Equivalent

Windows

Visual Studio: None

Linux

Eclipse: **Code Generation > Generate Position Independent Code**

Alternate Options

None

fpie

Tells the compiler to generate position-independent code. The generated code can only be linked into executables.

Syntax

Linux OS:

`-fpie`

Windows OS:

`None`

Arguments

None

Default

OFF	The compiler does not generate position-independent code for an executable-only object.

Description

This option tells the compiler to generate position-independent code. It is similar to `-fpic`, but code generated by `-fpie` can only be linked into an executable.

Because the object is linked into an executable, this option causes better optimization of some symbol references.

To ensure that run-time libraries are set up properly for the executable, you should also specify option `-pie` to the compiler driver on the link command line.

Option `-fpie` can also be specified as `-fPIE`.

IDE Equivalent

None

Alternate Options

None

See Also

fpic compiler option
pie compiler option

freg-struct-return

Tells the compiler to return struct and union values in registers when possible.

Architecture Restrictions

Only available on IA-32 architecture

Syntax

Linux OS:

`-freg-struct-return`

Windows OS:

None

Arguments

None

Default

OFF

Description

This option tells the compiler to return `struct` and `union` values in registers when possible.

NOTE
This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

None

fstack-protector

Enables or disables stack overflow security checks for certain (or all) routines.

Syntax

Linux OS:

`-fstack-protector[-`*`keyword`*`]`

`-fno-stack-protector[-`*`keyword`*`]`

Windows OS:

`None`

Arguments

`keyword`	Possible values are:	
	`strong`	When option `-fstack-protector-strong` is specified, it enables stack overflow security checks for routines with any type of buffer.
	`all`	When option `-fstack-protector-all` is specified, it enables stack overflow security checks for every routine.

If no `-keyword` is specified, option `-fstack-protector` enables stack overflow security checks for routines with a string buffer.

Default

`-fno-stack-protector`, `-fno-stack-protector-strong`	No stack overflow security checks are enabled for the relevant routines.
`-fno-stack-protector-all`	No stack overflow security checks are enabled for any routines.

Description

This option enables or disables stack overflow security checks for certain (or all) routines. A stack overflow occurs when a program stores more data in a variable on the execution stack than is allocated to the variable. Writing past the end of a string buffer or using an index for an array that is larger than the array bound could cause a stack overflow and security violations.

The `-fstack-protector` options are provided for compatibility with gcc. They use the gcc/glibc implementation when possible. If the gcc/glibc implementation is not available, they use the Intel implementation.

This content does not apply to SYCL.
For an Intel-specific version of this feature, see option `-fstack-security-check`.

IDE Equivalent

None

Alternate Options

None

See Also

`fstack-security-check` compiler option

`GS` compiler option

fstack-security-check

Determines whether the compiler generates code that detects some buffer overruns.

Syntax

Linux OS:

`-fstack-security-check`

`-fno-stack-security-check`

Windows OS:

None

Arguments

None

Default

`-fno-stack-security-check`	The compiler does not detect buffer overruns.

Description

This option determines whether the compiler generates code that detects some buffer overruns that overwrite the return address. This is a common technique for exploiting code that does not enforce buffer size restrictions.

This option always uses an Intel implementation.

For a gcc-compliant version of this feature, see option `fstack-protector`.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

Linux: None

Windows: `/GS`

See Also

`fstack-protector` compiler option

`GS` compiler option

fvisibility

Specifies the default visibility for global symbols or the visibility for symbols in declarations, functions, or variables.

Syntax

Linux OS:

`-fvisibility=`*arg*

`-fvisibility-global-new-delete-hidden`

`-fvisibility-inlines-hidden`

`-f[no]visibility-inlines-hidden-static-local-var`

`-fvisibility-ms-compat`

Windows OS:

None

Arguments

arg	Specifies the visibility setting. Possible values are:	
	`default`	Sets visibility to default. The symbol is visible outside this shared object. This means that other components can reference the symbols, and the symbol definitions can be overridden (preempted) by a definition of the same name in another component.
	`hidden`	Sets visibility to hidden. The symbol is *not* visible outside this shared object. This means that other components cannot directly reference the symbol.
	`internal`	This is the same as specifying hidden.
	`protected`	Sets visibility to protected. The symbol is seen by the dynamic linker but always dynamically resolves to an object within this shared object. This means that other components can reference the symbol, but it cannot be overridden by a definition of the same name in another component. This value is not supported on all targets.

Default

-fvisibility=default	The compiler sets visibility of symbols to default.

Description

This option specifies the default visibility for global symbols (syntax `-fvisibility=`*arg*) or the visibility for symbols in declarations, functions, or variables.

The following table shows supported `-fvisibility` options:

Option	Description
`-fvisibility=`*arg*	Sets visibility of symbols for all global declarations. As specified above in Arguments, *arg* can be one of the following: hidden internal default protected.
`-fvisibility-global-new-delete-hidden`	Sets hidden visibility for global C++ operator new and delete declarations.
`-fvisibility-inlines-hidden`	Sets hidden visibility by default for inline C++ member functions.
`-fvisibility-inlines-hidden-static-local-var` `-fno-visibility-inlines-hidden-static-local-var`	When `-fvisibility-inlines-hidden` is enabled, static variables in inline C++ member functions will also be given hidden visibility by default. To disable option `-fvisibility-inlines-hidden-static-local-var`, specify option `-fno-visibility-inlines-hidden-static-local-var`.
`-fvisibility-ms-compat`	Sets default visibility for global types and sets hidden visibility for global functions and variables.

If an -fvisibility option is specified more than once on the command line, the last specification takes precedence over any others.

The following shows the precedence of the visibility settings (from greatest to least visibility):

- `default`
- `protected`
- `hidden`

IDE Equivalent

None

Alternate Options

None

fzero-initialized-in-bss, Qzero-initialized-in-bss

Determines whether the compiler places in the DATA section any variables explicitly initialized with zeros.

Syntax

Linux OS:

`-fzero-initialized-in-bss`

`-fno-zero-initialized-in-bss`

Windows OS:

`/Qzero-initialized-in-bss`

`/Qzero-initialized-in-bss-`

Arguments

None

Default

`-fno-zero-initialized-in-bss` or `/Qzero-initialized-in-bss-`	Variables explicitly initialized with zeros are placed in the BSS section. This can save space in the resulting code.

Description

This option determines whether the compiler places in the DATA section any variables explicitly initialized with zeros.

If option `-fno-zero-initialized-in-bss` (Linux*) or `/Qzero-initialized-in-bss-` (Windows*) is specified, the compiler places in the DATA section any variables that are initialized to zero.

IDE Equivalent

Windows

Visual Studio: None

Linux

Eclipse: **Data > Disable Placement of Zero-Initialized Variables in .bss - place in .data instead**

Alternate Options

None

GA

Enables faster access to certain thread-local storage (TLS) variables.

Syntax

Linux OS:

`None`

Windows OS:

`/GA`

Arguments

None

Default

OFF	Default access to TLS variables is in effect.

Description

This option enables faster access to certain thread-local storage (TLS) variables. When you compile your main executable (`.EXE`) program with this option, it allows faster access to TLS variables declared with the `__declspec(thread)` specification.

Note that if you use this option to compile `.DLL`s, you may get program errors.

IDE Equivalent

Windows

Visual Studio: **Optimization > Optimize for Windows Applications**

Linux

Eclipse: None

Alternate Options

None

Gs

Lets you control the threshold at which the stack checking routine is called or not called.

Syntax

Linux OS:

`None`

Windows OS:

`/Gs[n]`

Arguments

n	Is the number of bytes that local variables and compiler temporaries can occupy before stack checking is activated. This is called the *threshold*.

Default

`/Gs`	Stack checking occurs for routines that require more than 4KB (4096 bytes) of stack space. This is also the default if you do not specify *n*.

Description

This option lets you control the threshold at which the stack checking routine is called or not called. If a routine's local stack allocation exceeds the threshold (*n*), the compiler inserts a __chkstk() call into the prologue of the routine.

NOTE
This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

None

GS

Determines whether the compiler generates code that detects some buffer overruns.

Syntax

Linux OS:

None

Windows OS:

/GS[:*keyword*]

/GS-

Arguments

keyword	Specifies the level of stack protection heuristics used by the compiler. Possible values are:	
	off	Tells the compiler to ignore buffer overruns. This is the same as specifying /GS-.
	partial	Tells the compiler to provide a stack protection level that is compatible with Microsoft* Visual Studio 2008.
	strong	Tells the compiler to provide full stack security level checking. This setting is compatible with more recent Microsoft* Visual Studio stack protection heuristics. This is the same as specifying /GS with no keyword.

Default

/GS-	The compiler does not detect buffer overruns.

Description

This option determines whether the compiler generates code that detects some buffer overruns that overwrite a function's return address, exception handler address, or certain types of parameters.

This option has been added for Microsoft compatibility.

Following Visual Studio 2008, the Microsoft implementation of option /GS became more extensive (for example, more routines are protected). The performance of some programs may be impacted by the newer heuristics. In such cases, you may see better performance if you specify /GS:partial.

For more details about option /GS, see the Microsoft documentation.

IDE Equivalent

Visual Studio

Visual Studio: **Code Generation > Security Check**

Eclipse

Eclipse: None

Alternate Options

SYCL: None

C++: Linux: `-fstack-security-check`

C++: Windows: None

See Also

`fstack-security-check` compiler option

`fstack-protector` compiler option

malign-double

Determines whether double, long double, and long long types are naturally aligned. This option is equivalent to specifying option align.

Architecture Restrictions

Only available on IA-32 architecture

Syntax

Linux OS:

`-malign-double`

`-mno-align-double`

Windows OS:

`None`

Arguments

None

Default

`-mno-align-double`	Types are aligned according to the gcc model, which means they are aligned to 4-byte boundaries.

Description

For details, see the align option.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

None

mcmodel

Tells the compiler to use a specific memory model to generate code and store data.

Architecture Restrictions

Only available on Intel® 64 architecture

Syntax

Linux OS:

`-mcmodel=`*`mem_model`*

Windows OS:

`None`

Arguments

mem_model	Is the memory model to use. Possible values are:	
	`small`	Tells the compiler to restrict code and data to the first 2GB of address space. All accesses of code and data can be done with Instruction Pointer (IP)-relative addressing.
	`medium`	Tells the compiler to restrict code to the first 2GB; it places no memory restriction on data. Accesses of code can be done with IP-relative addressing, but accesses of data must be done with absolute addressing.
	`large`	Places no memory restriction on code or data. All accesses of code and data must be done with absolute addressing.

Default

`-mcmodel=small`	On systems using Intel® 64 architecture, the compiler restricts code and data to the first 2GB of address space. Instruction Pointer (IP)-relative addressing can be used to access code and data.

Description

This option tells the compiler to use a specific memory model to generate code and store data. It can affect code size and performance. If your program has global and static data with a total size smaller than 2GB, `-mcmodel=small` is sufficient. Global and static data larger than 2GB requires`-mcmodel=medium` or `-mcmodel=large`. Allocation of memory larger than 2GB can be done with any setting of `-mcmodel`.

IP-relative addressing requires only 32 bits, whereas absolute addressing requires 64-bits. IP-relative addressing is somewhat faster. So, the `small` memory model has the least impact on performance.

NOTE
This content does not apply to SYCL.

When you specify option `-mcmodel=medium` or `-mcmodel=large`, it sets option `-shared-intel`. This ensures that the correct dynamic versions of the Intel run-time libraries are used.

If you specify option `-static-intel` while `-mcmodel=medium` or `-mcmodel=large` is set, an error will be displayed.

IDE Equivalent

None

Alternate Options

None

Example

The following example shows how to compile using `-mcmodel`:

This content does not apply to SYCL.

```
icx -shared-intel -mcmodel=medium -o prog prog.c
```

See Also

`shared-intel` compiler option

`fpic` compiler option

Qlong-double

Changes the default size of the long double data type.

Syntax

Linux OS:

`None`

Windows OS:

`/Qlong-double`

Arguments

None

Default

OFF	The default size of the long double data type is 64 bits.

Description

This option changes the default size of the long double data type to 80 bits.

However, the alignment requirement of the data type is 16 bytes, and its size must be a multiple of its alignment, so the size of a long double on Windows* is also 16 bytes. Only the lower 10 bytes (80 bits) of the 16 byte space will have valid data stored in it.

NOTE
Using the `Qlong-double` command-line option on Windows* platforms requires that any source code using `double extended precision` floating-point types (FP80) be carefully segregated from source code that was not written in a way that considers or supports their use. When this option is used, source code that makes assumptions or has requirements on the size or layout of an FP80 value may experience a variety of failures at compile time, link time, or run time.

The Microsoft* C Standard Library and Microsoft* C++ Standard Template Library do *not* support FP80 datatypes. In all circumstances where you want to use this option, please check with your library vendor to determine whether they support FP80 datatype formats.

For example, the Microsoft* compiler and Microsoft*-provided library routines (such as `printf` or `long double` math functions) do *not* provide support for 80-bit floating-point values and should not be called from code compiled with the `Qlong-double` command-line option.

Starting with the Microsoft Visual Studio 2019 version 16.10 release, you may get compilation errors when using options /std:c++latest together with /Qlong-double in programs that directly or indirectly include the <complex> header, <xutility> header, or the <cmath> header. To see an example of this, see the Example section below.

IDE Equivalent

None

Alternate Options

None

Example

In the Note above, we mention an issue with using the options /std:c++latest together with /Qlong-double in programs that directly or indirectly include the <complex>, <xutility>, or the <cmath> headers. The following shows an example of this issue:

```
#include <iostream>
#include <complex>

int main()
{long double ld2 = 1256789.98765432106L;int iNan = isnan(ld2);std::cout << "Hello World!\n"; }

ksh-3.2$ icl -c -EHsc -GR    -std:c++latest /Qlong-double /MD  test1.cpp
Intel(R) C++ Intel(R) 64 Compiler Classic for applications running on Intel(R) 64, Version xxx
Build xxxx
Copyright (C) 1985-2021 Intel Corporation.  All rights reserved.

test1.cpp
c:/Program files/Microsoft Visual Studio/2022/Preview/VC/Tools/MSVC/14.29.30130/include/
xutility(5918): error: no instance of function template "std::_Bit_cast" matches the argument
list
            argument types are: (const long double)
      const auto _Bits = _Bit_cast<_Uint_type>(_Xx);
                         ^
c:/Program files/Microsoft Visual Studio/2022/Preview/VC/Tools/MSVC/14.29.30130/include/
xutility(67): note: this candidate was rejected because at least one template argument could not
be deduced
  _NODISCARD _CONSTEXPR_BIT_CAST _To _Bit_cast(const _From& _Val) noexcept {
```

```
                                        ^
          detected during:
            instantiation of "auto std::_Float_abs_bits(const _Ty &) [with _Ty=long double,
<unnamed>=0]" at line 5967
            instantiation of "bool std::_Is_finite(_Ty) [with _Ty=long double, <unnamed>=0]" at
line 1307 of "c:/Program files/Microsoft Visual Studio/2022/Preview/VC/Tools/MSVC/14.29.30130/
include/cmath"
            instantiation of "_Ty std::_Common_lerp(_Ty, _Ty, _Ty) noexcept [with _Ty=long
double]" at line 1392 of "c:/Program files/Microsoft Visual Studio/2022/Preview/VC/Tools/MSVC/
14.29.30130/include/cmath"

compilation aborted for test1.cpp (code 2)
```

Compiler Diagnostic Options

This section contains descriptions for compiler options that pertain to compiler diagnostics. They are listed in alphabetical order.

w

Disables all warning messages.

Syntax

Linux OS:

`-w`

Windows OS:

`/w`

Arguments

None

Default

OFF	Default warning messages are enabled.

Description

This option disables all warning messages.

IDE Equivalent

Windows

Visual Studio: **General > Warning Level**

Linux

Eclipse: **General > Warning Level**

Alternate Options

Linux: `-w0`

Windows: `/W0`

w, W

Specifies the level of diagnostic messages to be generated by the compiler.

Syntax

Linux OS:

`-wn`

Windows OS:

`/Wn`

Arguments

n	Is the level of diagnostic messages to be generated. Possible values are:	
	`0`	Enables diagnostics for errors. Disables diagnostics for warnings.
	`1`	Enables diagnostics for warnings and errors.
	`2`	Enables diagnostics for warnings and errors. On Linux* systems, additional warnings are enabled. On Windows* systems, this setting is equivalent to level 1 (*n* = 1).
	`3`	Enables diagnostics for remarks, warnings, and errors. Additional warnings are also enabled above level 2 (*n* = 2). This level is recommended for production purposes.
	`4`	Enables diagnostics for all level 3 (*n* = 3) warnings plus informational warnings and remarks, which in most cases can be safely ignored. This value is only available on Windows* systems.
	`5`	Enables diagnostics for all remarks, warnings, and errors. This setting produces the most diagnostic messages. This value is only available on Windows* systems.

Default

n=1	The compiler displays diagnostics for warnings and errors.

Description

This option specifies the level of diagnostic messages to be generated by the compiler.

On Windows systems, option /W4 is equivalent to option /Wall.

The `-wn`, `/Wn`, and `Wall` options can override each other. The last option specified on the command line takes precedence.

IDE Equivalent

Windows

Visual Studio: **General > Warning Level**

Linux

Eclipse: **General > Warning Level**

Alternate Options

None

See Also

Wall compiler option

Wabi

Determines whether a warning is issued if generated code is not C++ ABI compliant.

Syntax

Linux OS:

`-Wabi`

`-Wno-abi`

Windows OS:

`None`

Arguments

None

Default

`-Wno-abi`	No warning is issued when generated code is not C++ ABI compliant.

Description

This option determines whether a warning is issued if generated code is not C++ ABI compliant.

IDE Equivalent

None

Alternate Options

None

Wall

Enables warning and error diagnostics.

Syntax

Linux OS:

`-Wall`

Windows OS:

`/Wall`

Arguments

None

Default

OFF	Only default warning diagnostics are enabled.

Description

This option enables many warning and error diagnostics.

On Windows* systems, this option is equivalent to the `/W4` option. It enables diagnostics for all level 3 warnings plus informational warnings and remarks.

However, on Linux* systems, this option is similar to gcc option `-Wall`. It displays all errors and some of the warnings that are typically reported by gcc option `-Wall`. If you want to display all warnings, specify the `-w2` or `-w3` option.

The `Wall`, `-w`*n*, and `/W`*n* options can override each other. The last option specified on the command line takes precedence.

IDE Equivalent

None

Alternate Options

None

See Also

w, W compiler option

Wcheck-unicode-security

Determines whether the compiler performs source code checking for Unicode vulnerabilities.

Syntax

Linux OS:

`-Wcheck-unicode-security`

`-Wno-check-unicode-security`

Windows OS:

`/Wcheck-unicode-security`

`/Wno-check-unicode-security`

Arguments

None

Default

`Wno-check-unicode-security`	The compiler does not perform source code checking for Unicode vulnerabilities.

Description

This option determines whether the compiler performs source code checking for Unicode vulnerabilities.

Option `Wcheck-unicode-security` enables Unicode checking. The compiler will detect and warn about Unicode constructs that can be exploited by using bi-directional formatting codes, zero-width characters in strings, and use of zero-width characters and homoglyphs in identifiers.

Option `Wno-check-unicode-security` disables Unicode checking.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

Windows

Visual Studio: DPC++: **DPC++ > Diagnostics > Check Unicode Security**

C/C++: **C/C++ > Diagnostics [Intel C++] > Check Unicode Security**

Linux

Eclipse: DPC++: **Intel(R) oneAPI DPC++ Compiler > Diagnostics > Check Unicode Security**

C/C++: **Intel C++ Compiler > Compilation Diagnostics > Check Unicode Security**

Alternate Options

None

Wcomment

Determines whether a warning is issued when / appears in the middle of a /* */ comment.*

Syntax

Linux OS:

`-Wcomment`

`-Wno-comment`

Windows OS:

`None`

Arguments

None

Default

`-Wno-comment`	No warning is issued when /* appears in the middle of a /* */ comment.

Description

This option determines whether a warning is issued when /* appears in the middle of a /* */ comment.

IDE Equivalent

None

Alternate Options

None

Wdeprecated

Determines whether warnings are issued for deprecated C++ headers.

Syntax

Linux OS:

`-Wdeprecated`

`-Wno-deprecated`

Windows OS:

None

Arguments

None

Default

`-Wdeprecated`	The compiler issues warnings for deprecated C++ headers.

Description

This option determines whether warnings are issued for deprecated C++ headers. It has no effect in C compilation mode.

Option `-Wdeprecated` enables these warnings by defining the __DEPRECATED macro for preprocessor.

To disable warnings for deprecated C++ headers, specify `-Wno-deprecated`.

IDE Equivalent

None

Alternate Options

None

Weffc++, Qeffc++

Enables warnings based on certain C++ programming guidelines.

Syntax

Linux OS:

`-Weffc++`

Windows OS:

`/Qeffc++`

Arguments

None

Default

OFF	Diagnostics are not enabled.

Description

This option enables warnings based on certain programming guidelines developed by Scott Meyers in his books on effective C++ programming. With this option, the compiler emits warnings for these guidelines:

- Use `const` and `inline` rather than `#define`. Note that you will only get this in user code, not system header code.
- Use `<iostream>` rather than `<stdio.h>`.
- Use `new` and `delete` rather than `malloc` and `free`.
- Use C++ style comments in preference to C style comments. C comments in system headers are not diagnosed.
- Use `delete` on pointer members in destructors. The compiler diagnoses any pointer that does not have a `delete`.
- Make sure you have a user copy constructor and assignment operator in classes containing pointers.
- Use initialization rather than assignment to members in constructors.
- Make sure the initialization list ordering matches the declartion list ordering in constructors.
- Make sure base classes have virtual destructors.
- Make sure `operator=` returns `*this`.
- Make sure prefix forms of increment and decrement return a `const` object.
- Never overload operators `&&`, `||`, and `,`.

NOTE
The warnings generated by this compiler option are based on the following books from Scott Meyers:

- Effective C++ Second Edition - 50 Specific Ways to Improve Your Programs and Designs
- More Effective C++ - 35 New Ways to Improve Your Programs and Designs

IDE Equivalent

Windows

Visual Studio: None

Linux

Eclipse: **Compilation Diagnostics > Enable Warnings for Style Guideline Violations**

Alternate Options

None

Werror, WX

Changes all warnings to errors.

Syntax

Linux OS:

`-Werror`

Windows OS:

`/WX`

Arguments

None

Default

OFF	The compiler returns diagnostics as usual.

Description

This option changes all warnings to errors.

IDE Equivalent

Windows

Visual Studio: **General > Treat Warnings As Errors**

Linux

Eclipse: **Compilation Diagnostics > Treat Warnings As Errors**

Alternate Options

None

Werror-all

Causes all warnings and currently enabled remarks to be reported as errors.

Syntax

Linux OS:

`-Werror-all`

Windows OS:

`/Werror-all`

Arguments

None

Default

OFF	The compiler returns diagnostics as usual.

Description

This option causes all warnings and currently enabled remarks to be reported as errors.

IDE Equivalent

None

Alternate Options

None

Wextra-tokens

Determines whether warnings are issued about extra tokens at the end of preprocessor directives.

Syntax

Linux OS:

`-Wextra-tokens`

`-Wno-extra-tokens`

Windows OS:

`None`

Arguments

None

Default

`-Wno-extra-tokens`	The compiler does not warn about extra tokens at the end of preprocessor directives.

Description

This option determines whether warnings are issued about extra tokens at the end of preprocessor directives.

IDE Equivalent

None

Alternate Options

None

Wformat

Determines whether argument checking is enabled for calls to printf, scanf, and so forth.

Syntax

Linux OS:

`-Wformat`

`-Wno-format`

Windows OS:

`None`

Arguments

None

Default

`-Wno-format`	Argument checking is not enabled for calls to `printf`, `scanf`, and so forth.

Description

This option determines whether argument checking is enabled for calls to `printf`, `scanf`, and so forth.

IDE Equivalent

None

Alternate Options

None

Wformat-security

Determines whether the compiler issues a warning when the use of format functions may cause security problems.

Syntax

Linux OS:

`-Wformat-security`

`-Wno-format-security`

Windows OS:

`None`

Arguments

None

Default

`-Wno-format-security`	No warning is issued when the use of format functions may cause security problems.

Description

This option determines whether the compiler issues a warning when the use of format functions may cause security problems.

When `-Wformat-security` is specified, it warns about uses of format functions where the format string is not a string literal and there are no format arguments.

IDE Equivalent

None

Alternate Options

None

Wmain

Determines whether a warning is issued if the return type of main is not expected.

Syntax

Linux OS:

`-Wmain`

`-Wno-main`

Windows OS:

`None`

Arguments

None

Default

`-Wno-main`	No warning is issued if the return type of `main` is not expected.

Description

This option determines whether a warning is issued if the return type of `main` is not expected.

IDE Equivalent

None

Alternate Options

None

Wmissing-declarations

Determines whether warnings are issued for global functions and variables without prior declaration.

Syntax

Linux OS:

`-Wmissing-declarations`

`-Wno-missing-declarations`

Windows OS:

`None`

Arguments

None

Default

`-Wno-missing-declarations`	No warnings are issued for global functions and variables without prior declaration.

Description

This option determines whether warnings are issued for global functions and variables without prior declaration.

IDE Equivalent

None

Alternate Options

None

Wmissing-prototypes

Determines whether warnings are issued for missing prototypes.

Syntax

Linux OS:

`-Wmissing-prototypes`

`-Wno-missing-prototypes`

Windows OS:

None

Arguments

None

Default

`-Wno-missing-prototypes`	No warnings are issued for missing prototypes.

Description

Determines whether warnings are issued for missing prototypes.

If `-Wmissing-prototypes` is specified, it tells the compiler to detect global functions that are defined without a previous prototype declaration.

IDE Equivalent

None

Alternate Options

None

Wpointer-arith

Determines whether warnings are issued for questionable pointer arithmetic.

Syntax

Linux OS:

`-Wpointer-arith`

`-Wno-pointer-arith`

Windows OS:

`None`

Arguments

None

Default

`-Wno-pointer-arith`	No warnings are issued for questionable pointer arithmetic.

Description

Determines whether warnings are issued for questionable pointer arithmetic.

IDE Equivalent

None

Alternate Options

None

Wreorder

Tells the compiler to issue a warning when the order of member initializers does not match the order in which they must be executed.

Syntax

Linux OS:

`-Wreorder`

Windows OS:

`None`

Arguments

None

Default

OFF	The compiler does not issue a warning.

Description

This option tells the compiler to issue a warning when the order of member initializers does not match the order in which they must be executed. This option is supported for C++ only.

IDE Equivalent

None

Alternate Options

None

Wreturn-type

Determines whether warnings are issued when a function is declared without a return type, when the definition of a function returning void contains a return statement with an expression, or when the closing brace of a function returning non-void is reached.

Syntax

Linux OS:

`-Wreturn-type`

`-Wno-return-type`

Windows OS:

`None`

Arguments

None

Default

ON for one condition	A warning is issued when the closing brace of a function returning non-void is reached.

Description

This option determines whether warnings are issued for the following:

- When a function is declared without a return type
- When the definition of a function returning void contains a return statement with an expression
- When the closing brace of a function returning non-void is reached

Specify `-Wno-return-type` if you do not want to see warnings about the above diagnostics.

IDE Equivalent

None

Alternate Options

None

Wshadow

Determines whether a warning is issued when a variable declaration hides a previous declaration.

Syntax

Linux OS:

`-Wshadow`

`-Wno-shadow`

Windows OS:

`None`

Arguments

None

Default

`-Wno-shadow`	No warning is issued when a variable declaration hides a previous declaration.

Description

This option determines whether a warning is issued when a variable declaration hides a previous declaration. Same as `-ww1599`.

IDE Equivalent

None

Alternate Options

None

Wsign-compare

Determines whether warnings are issued when a comparison between signed and unsigned values could produce an incorrect result when the signed value is converted to unsigned.

Syntax

Linux OS:

`-Wsign-compare`

`-Wno-sign-compare`

Windows OS:

`None`

Arguments

None

Default

`-Wno-sign-compare`	The compiler does not issue these warnings

Description

This option determines whether warnings are issued when a comparison between signed and unsigned values could produce an incorrect result when the signed value is converted to unsigned.

On Linux* systems, this option is provided for compatibility with gcc.

IDE Equivalent

None

Alternate Options

None

Wstrict-aliasing

Determines whether warnings are issued for code that might violate the optimizer's strict aliasing rules.

Syntax

Linux OS:

`-Wstrict-aliasing`

`-Wno-strict-aliasing`

Windows OS:

`None`

Arguments

None

Default

`-Wno-strict-aliasing`	No warnings are issued for code that might violate the optimizer's strict aliasing rules.

Description

This option determines whether warnings are issued for code that might violate the optimizer's strict aliasing rules. These warnings will only be issued if you also specify option `-fstrict-aliasing`.

IDE Equivalent

None

Alternate Options

None

Wstrict-prototypes

Determines whether warnings are issued for functions declared or defined without specified argument types.

Syntax

Linux OS:

`-Wstrict-prototypes`

`-Wno-strict-prototypes`

Windows OS:

`None`

Arguments

None

Default

`-Wno-strict-prototypes`	No warnings are issued for functions declared or defined without specified argument types.

Description

This option determines whether warnings are issued for functions declared or defined without specified argument types.

IDE Equivalent

None

Alternate Options

None

Wtrigraphs

Determines whether warnings are issued if any trigraphs are encountered that might change the meaning of the program.

Syntax

Linux OS:

`-Wtrigraphs`

`-Wno-trigraphs`

Windows OS:

`None`

Arguments

None

Default

`-Wno-trigraphs`	No warnings are issued if any trigraphs are encountered that might change the meaning of the program.

Description

This option determines whether warnings are issued if any trigraphs are encountered that might change the meaning of the program.

IDE Equivalent

None

Alternate Options

None

Wuninitialized

Determines whether a warning is issued if a variable is used before being initialized.

Syntax

Linux OS:

`-Wuninitialized`

`-Wno-uninitialized`

Windows OS:

`None`

Arguments

None

Default

`-Wno-uninitialized`	No warning is issued if a variable is used before being initialized.

Description

This option determines whether a warning is issued if a variable is used before being initialized. Equivalent to `-ww592` and `-wd592`.

IDE Equivalent

None

Alternate Options

`-ww592` and `-wd592`

Wunknown-pragmas

Determines whether a warning is issued if an unknown #pragma directive is used.

Syntax

Linux OS:

`-Wunknown-pragmas`

`-Wno-unknown-pragmas`

Windows OS:

`None`

Arguments

None

Default

`-Wunknown-pragmas`	A warning is issued if an unknown `#pragma` directive is used.

Description

This option determines whether a warning is issued if an unknown `#pragma` directive is used.

IDE Equivalent

None

Alternate Options

None

Wunused-function

Determines whether a warning is issued if a declared function is not used.

Syntax

Linux OS:

`-Wunused-function`

`-Wno-unused-function`

Windows OS:

`None`

Arguments

None

Default

`-Wno-unused-function`	No warning is issued if a declared function is not used.

Description

This option determines whether a warning is issued if a declared function is not used.

IDE Equivalent

None

Alternate Options

None

Wunused-variable

Determines whether a warning is issued if a local or non-constant static variable is unused after being declared.

Syntax

Linux OS:

`-Wunused-variable`

`-Wno-unused-variable`

Windows OS:

`None`

Arguments

None

Default

`-Wno-unused-variable`	No warning is issued if a local or non-constant static variable is unused after being declared.

Description

This option determines whether a warning is issued if a local or non-constant static variable is unused after being declared.

IDE Equivalent

None

Alternate Options

None

Wwrite-strings

*Issues a diagnostic message if const char * is converted to (non-const) char *.*

Syntax

Linux OS:

`-Wwrite-strings`

Windows OS:

`None`

Arguments

None

Default

OFF	No diagnostic message is issued if const char * is converted to (non-const) char*.

Description

This option issues a diagnostic message if const char* is converted to (non-const) char *.

IDE Equivalent

None

Alternate Options

None

Compatibility Options

This section contains descriptions for compiler options that pertain to language compatibility.

gcc-toolchain

Lets you specify the location of the base toolchain.

Syntax

Linux OS:

`--gcc-toolchain=dir`

Windows OS:

None

Arguments

dir	Is the location of the base toolchain.

Default

OFF	The compiler uses heuristics to locate the base toolchain.

Description

This option lets you specify the location of the base toolchain.

NOTE
This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

None

vmv

Enables pointers to members of any inheritance type.

Syntax

Linux OS:

None

Windows OS:

`/vmv`

Arguments

None

Default

OFF	The compiler uses default rules to represent pointers to members.

Description

This option enables pointers to members of any inheritance type. To use this option, you must also specify option `/vmg`.

IDE Equivalent

None

Alternate Options

None

Linking or Linker Options

This section contains descriptions for compiler options that pertain to linking or to the linker. They are listed in alphabetical order.

Bdynamic

Enables dynamic linking of libraries at run time.

Syntax

Linux OS:

`-Bdynamic`

Windows OS:

`None`

Arguments

None

Default

OFF	Limited dynamic linking occurs.

Description

This option enables dynamic linking of libraries at run time. Smaller executables are created than with static linking.

This option is placed in the linker command line corresponding to its location on the user command line. It controls the linking behavior of any library that is passed using the command line.

All libraries on the command line following option `-Bdynamic` are linked dynamically until the end of the command line or until a `-Bstatic` option is encountered. The `-Bstatic` option enables static linking of libraries.

NOTE
This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

None

See Also

`Bstatic` compiler option

Bstatic

Enables static linking of a user's library.

Syntax

Linux OS:

`-Bstatic`

Windows OS:

None

Arguments

None

Default

OFF	Default static linking occurs.

Description

This option enables static linking of a user's library.

This option is placed in the linker command line corresponding to its location on the user command line. It controls the linking behavior of any library that is passed using the command line.

All libraries on the command line following option `-Bstatic` are linked statically until the end of the command line or until a `-Bdynamic` option is encountered. The `-Bdynamic` option enables dynamic linking of libraries.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

None

See Also

`Bdynamic` compiler option

Bsymbolic

Binds references to all global symbols in a program to the definitions within a user's shared library.

Syntax

Linux OS:

`-Bsymbolic`

Windows OS:

`None`

Arguments

None

Default

OFF	When a program is linked to a shared library, it can override the definition within the shared library.

Description

This option binds references to all global symbols in a program to the definitions within a user's shared library.

This option is only meaningful on Executable Linkage Format (ELF) platforms that support shared libraries.

Caution
This option can have unintended side-effects of disabling symbol preemption in the shared library.

NOTE
This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

None

See Also

`Bsymbolic-functions` compiler option

Bsymbolic-functions

Binds references to all global function symbols in a program to the definitions within a user's shared library.

Syntax

Linux OS:

`-Bsymbolic-functions`

Windows OS:

`None`

Arguments

None

Default

OFF	When a program is linked to a shared library, it can override the definition within the shared library.

Description

This option binds references to all global function symbols in a program to the definitions within a user's shared library.

This option is only meaningful on Executable Linkage Format (ELF) platforms that support shared libraries.

> **Caution**
> This option can have unintended side-effects of disabling symbol preemption in the shared library.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

None

See Also

`Bsymbolic` compiler option

dynamic-linker

Specifies a dynamic linker other than the default.

Syntax

Linux OS:

`-dynamic-linker` *`file`*

Windows OS:

`None`

Arguments

file	Is the name of the dynamic linker to be used.

Default

OFF	The default dynamic linker is used.

Description

This option lets you specify a dynamic linker other than the default.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

None

F (Windows*)

Specifies the stack reserve amount for the program.

Syntax

Linux OS:

`None`

Windows OS:

`/Fn`

Arguments

n	Is the stack reserve amount. It can be specified as a decimal integer or as a hexadecimal constant by using a C-style convention (for example, /F0x1000).

Default

OFF	The stack size default is chosen by the operating system.

Description

This option specifies the stack reserve amount for the program. The amount (*n*) is passed to the linker.

Note that the linker property pages have their own option to do this.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

None

fixed

Causes the linker to create a program that can be loaded only at its preferred base address.

Syntax

Linux OS:

`None`

Windows OS:

`/fixed`

Arguments

None

Default

OFF	The compiler uses default methods to load programs.

Description

This option is passed to the linker, causing it to create a program that can be loaded only at its preferred base address.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

None

Fm

Tells the linker to generate a link map file. This is a deprecated option that may be removed in a future release.

Syntax

Linux OS:

`None`

Windows OS:

`/Fm[`*`filename|dir`*`]`

Arguments

filename	Is the name for the link map file.
dir	Is the directory where the link map file should be placed. It can include *file*.

Default

OFF	No link map is generated.

Description

This option tells the linker to generate a link map.

This is a deprecated option that may be removed in a future release. There is no replacement option.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

None

fuse-ld

Tells the compiler to use a different linker instead of the default linker (ld).

Syntax

Linux OS:

`-fuse-ld=`*`keyword`*

Windows OS:

`None`

Arguments

`keyword`	Possible values are:	
	`bfd`	Tells the compiler to use the bfd linker.
	`gold`	Tells the compiler to use the gold linker.

Default

ld	The compiler uses the ld linker by default.

Description

This option tells the compiler to use a different linker instead of default linker (ld).

This option is provided for compatibility with gcc.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

None

l

Tells the linker to search for a specified library when linking.

Syntax

Linux OS:

`-lstring`

Windows OS:

`None`

Arguments

string	Specifies the library (lib*string*) that the linker should search.

Default

OFF	The linker searches for standard libraries in standard directories.

Description

This option tells the linker to search for a specified library when linking.

When resolving references, the linker normally searches for libraries in several standard directories, in directories specified by the `L` option, then in the library specified by the `l` option.

The linker searches and processes libraries and object files in the order they are specified. So, you should specify this option following the last object file it applies to.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

None

See Also

`L` compiler option

L

Tells the linker to search for libraries in a specified directory before searching the standard directories.

Syntax

Linux OS:

`-Ldir`

Windows OS:

`None`

Arguments

dir	Is the name of the directory to search for libraries.

Default

OFF	The linker searches the standard directories for libraries.

Description

This option tells the linker to search for libraries in a specified directory before searching for them in the standard directories.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

None

See Also

`l` compiler option

LD

Specifies that a program should be linked as a dynamic-link (DLL) library.

Syntax

Linux OS:

`None`

Windows OS:

`/LD`

```
/LDd
```

Arguments

None

Default

OFF	The program is not linked as a dynamic-link (DLL) library.

Description

This option specifies that a program should be linked as a dynamic-link (DLL) library instead of an executable (.exe) file. You can also specify `/LDd`, where *d* indicates a debug version.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

None

link

Passes user-specified options directly to the linker at compile time.

Syntax

Linux OS:

```
None
```

Windows OS:

```
/link
```

Arguments

None

Default

OFF	No user-specified options are passed directly to the linker.

Description

This option passes user-specified options directly to the linker at compile time.

All options that appear following `/link` are passed directly to the linker.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

None

See Also

`Xlinker` compiler option

MD

Tells the linker to search for unresolved references in a multithreaded, dynamic-link run-time library.

Syntax

Linux OS:

`None`

Windows OS:

`/MD`

`/MDd`

Arguments

None

Default

OFF	The linker searches for unresolved references in a multi-threaded, static run-time library.

Description

This option tells the linker to search for unresolved references in a multithreaded, dynamic-link (DLL) run-time library. You can also specify `/MDd`, where `d` indicates a debug version.

This option is processed by the compiler, which adds directives to the compiled object file that are processed by the linker.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

Visual Studio

Visual Studio: **Code Generation > Runtime Library**

Eclipse

Eclipse: None

Alternate Options

None

MT

Tells the linker to search for unresolved references in a multithreaded, static run-time library.

Syntax

Linux OS:

`None`

Windows OS:

`/MT`

`/MTd`

Arguments

None

Default

`/MT`	The linker searches for unresolved references in a multithreaded, static run-time library.

Description

This option tells the linker to search for unresolved references in a multithreaded, static run-time library. You can also specify `/MTd`, where `d` indicates a debug version.

This option is processed by the compiler, which adds directives to the compiled object file that are processed by the linker.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

Visual Studio

Visual Studio: **Code Generation > Runtime Library**

Eclipse

Eclipse: None

Alternate Options

None

See Also

no-libgcc

Prevents the linking of certain gcc-specific libraries.

Syntax

Linux OS:

`-no-libgcc`

Windows OS:

`None`

Arguments

None

Default

OFF

Description

This option prevents the linking of certain gcc-specific libraries.

This option is not recommended for general use.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

None

nodefaultlibs

Prevents the compiler from using standard libraries when linking.

Syntax

Linux OS:

`-nodefaultlibs`

Windows OS:

`None`

Arguments

None

Default

OFF	The standard libraries are linked.

Description

This option prevents the compiler from using standard libraries when linking.

On Linux* systems, it is provided for GNU compatibility.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

Windows

Visual Studio: None

Linux

Eclipse: **Libraries > Use no system libraries**

Alternate Options

None

See Also

nostdlib compiler option

no-intel-lib, Qno-intel-lib

Disables linking to specified Intel® libraries, or to all Intel® libraries.

Syntax

Linux OS:

`-no-intel-lib[=`*`library`*`]`

Windows OS:

`/Qno-intel-lib[:`*`library`*`]`

Arguments

library — Indicates which Intel® library should *not* be linked. Possible values are:

`libirc`	Disables linking to the Intel® C/C++ library.
`libimf`	Disables linking to the Intel® oneAPI DPC++/C++ Compiler Math library. This value is only available for Linux*.
`libsvml`	Disables linking to the Intel® Short Vector Math library.
`libirng`	Disables linking to the Random Number Generator library.

If you specify more than one *library*, they must be separated by commas.

If *library* is omitted, the compiler will not link to any of the Intel® libraries shown above.

Default

OFF	If this option is not specified, the compiler uses default heuristics for linking to libraries.

Description

This option disables linking to specified Intel® libraries, or to all Intel® libraries.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

None

nostartfiles

Prevents the compiler from using standard startup files when linking.

Syntax

Linux OS:

`-nostartfiles`

Windows OS:

`None`

Arguments

None

Default

OFF	The compiler uses standard startup files when linking.

Description

This option prevents the compiler from using standard startup files when linking.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

None

See Also

`nostdlib` compiler option

nostdlib

Prevents the compiler from using standard libraries and startup files when linking.

Syntax

Linux OS:

`-nostdlib`

Windows OS:

`None`

Arguments

None

Default

OFF	The compiler uses standard startup files and standard libraries when linking.

Description

This option prevents the compiler from using standard libraries and startup files when linking.

This option is provided for GNU compatibility.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

None

See Also

`nodefaultlibs` compiler option

`nostartfiles` compiler option

pie

Determines whether the compiler generates position-independent code that will be linked into an executable.

Syntax

Linux OS:

`-pie`

`-no-pie`

Windows OS:

None

Arguments

None

Default

varies	On Linux*, the default is -no-pie.

Description

This option determines whether the compiler generates position-independent code that will be linked into an executable. To enable generation of position-independent code that will be linked into an executable, specify `-pie`.

To disable generation of position-independent code that will be linked into an executable, specify `-no-pie`.

IDE Equivalent

None

Alternate Options

None

See Also

`fpic` compiler option

pthread

Tells the compiler to use pthreads library for multithreading support.

Syntax

Linux OS:

`-pthread`

Windows OS:

None

Arguments

None

Default

OFF	The compiler does not use pthreads library for multithreading support.

Description

Tells the compiler to use pthreads library for multithreading support.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

None

shared

Tells the compiler to produce a dynamic shared object instead of an executable.

Syntax

Linux OS:

`-shared`

Windows OS:

`None`

Arguments

None

Default

OFF	The compiler produces an executable.

Description

This option tells the compiler to produce a dynamic shared object (DSO) instead of an executable. This includes linking in all libraries dynamically and passing `-shared` to the linker.

You must specify option `fpic` for the compilation of each object file you want to include in the shared library.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

None

See Also

`fpic` compiler option

`Xlinker` compiler option

shared-intel

Causes Intel-provided libraries to be linked in dynamically.

Syntax

Linux OS:

`-shared-intel`

Windows OS:

`None`

Arguments

None

Default

OFF	Intel® libraries are linked in statically, with the exception of Intel's OpenMP* runtime support library, which is linked in dynamically unless you specify option `-qopenmp-link=static`.

Description

This option causes Intel-provided libraries to be linked in dynamically. It is the opposite of `-static-intel`.

This option is processed by the compiler driver command that initiates linking, adding library names explicitly to the link command.

If you specify option `-mcmodel=medium` or `-mcmodel=large`, it sets option `-shared-intel`.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

Visual Studio

Visual Studio: None

Eclipse

Eclipse: None

Alternate Options

None

See Also

`static-intel` compiler option

`qopenmp-link` compiler option

shared-libgcc

Links the GNU libgcc library dynamically.

Syntax

Linux OS:

`-shared-libgcc`

Windows OS:

None

Arguments

None

Default

`-shared-libgcc`	The compiler links the `libgcc` library dynamically.

Description

This option links the GNU `libgcc` library dynamically. It is the opposite of option `static-libgcc`.

This option is processed by the compiler driver command that initiates linking, adding library names explicitly to the link command.

This option is useful when you want to override the default behavior of the `static` option, which causes all libraries to be linked statically.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

None

See Also

static-libgcc compiler option

static

Prevents linking with shared libraries.

Syntax

Linux OS:

`-static`

Windows OS:

None

Arguments

None

Default

OFF	The compiler links with shared libraries except as otherwise specified by `-static-intel` or its default.

Description

This option prevents linking with shared libraries. It causes the executable to link all libraries statically.

> **NOTE**
> This option does not cause static linking of libraries for which no static version is available. These libraries can only be linked dynamically.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

Visual Studio

Visual Studio: None

Eclipse

Eclipse: **Libraries > Link with static libraries**

Alternate Options

None

See Also

`static-intel` compiler option

static-intel

Causes Intel-provided libraries to be linked in statically.

Syntax

Linux OS:

`-static-intel`

Windows OS:

`None`

Arguments

None

Default

ON	Intel® libraries are linked in statically, with the exception of Intel's OpenMP* runtime support library, which is linked in dynamically unless you specify option `-qopenmp-link=static`.

Description

This option causes Intel-provided libraries to be linked in statically with certain exceptions (see the Default above). It is the opposite of `-shared-intel`.

This option is processed by the icx or icpx command that initiates linking, adding library names explicitly to the link command.

If you specify option `-static-intel` while option `-mcmodel=medium` or `-mcmodel=large` is set, an error will be displayed.

If you specify option `-static-intel` and any of the Intel-provided libraries have no static version, a diagnostic will be displayed.

NOTE
This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

Visual Studio

Visual Studio: None

Eclipse

Eclipse: None

Alternate Options

None

See Also

`shared-intel` compiler option
`qopenmp-link` compiler option

static-libgcc

Links the GNU libgcc library statically.

Syntax

Linux OS:

`-static-libgcc`

Windows OS:

None

Arguments

None

Default

OFF	The compiler links the GNU `libgcc` library dynamically.

Description

This option links the GNU `libgcc` library statically. It is the opposite of option `-shared-libgcc`.

This option is processed by the compiler driver command that initiates linking, adding library names explicitly to the link command.

This option is useful when you want to override the default behavior, which causes the library to be linked dynamically.

NOTE
If you want to use traceback, you must also link to the static version of the `libgcc` library. This library enables printing of backtrace information.

NOTE
This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

None

See Also

`shared-libgcc` compiler option
`static-libstdc++` compiler option

static-libstdc++

Links the GNU libstdc++ library statically.

Syntax

Linux OS:

`-static-libstdc++`

Windows OS:

`None`

Arguments

None

Default

OFF	The compiler links the GNU `libstdc++` library dynamically.

Description

This option links the GNU `libstdc++` library statically.

This option is processed by the compiler driver command that initiates linking, adding library names explicitly to the link command.

This option is useful when you want to override the default behavior, which causes the library to be linked dynamically.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

None

See Also

`static-libgcc` compiler option

T

Tells the linker to read link commands from a file.

Syntax

Linux OS:

`-T`*`filename`*

Windows OS:

`None`

Arguments

filename	Is the name of the file.

Default

OFF	The linker does not read link commands from a file.

Description

This option tells the linker to read link commands from a file.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

None

u (Linux*)

Tells the compiler the specified symbol is undefined.

Syntax

Linux OS:

`-u` *`symbol`*

Windows OS:

`None`

Arguments

None

Default

OFF	Standard rules are in effect for variables.

Description

This option tells the compiler the specified *symbol* is undefined.

IDE Equivalent

None

Alternate Options

None

v

Specifies that driver tool commands should be displayed and executed.

Syntax

Linux OS:

`-v [`*`filename`*`]`

Windows OS:

`None`

Arguments

filename	Is the name of a source file to be compiled. A space must appear before the file name.

Default

OFF	No tool commands are shown.

Description

This option specifies that driver tool commands should be displayed and executed.

If you use this option without specifying a source file name, the compiler displays only the version of the compiler.

IDE Equivalent

None

Alternate Options

None

See Also

`dryrun` compiler option

Wa

Passes options to the assembler for processing.

Syntax

Linux OS:

`-Wa,`*`option1`*`[,`*`option2`*`,...]`

Windows OS:

`None`

Arguments

option	Is an assembler option. This option is not processed by the driver and is directly passed to the assembler.

Default

OFF	No options are passed to the assembler.

Description

This option passes one or more options to the assembler for processing. If the assembler is not invoked, these options are ignored.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

None

Wl

Passes options to the linker for processing.

Syntax

Linux OS:

`-Wl,`*`option1`*`[,`*`option2`*`,...]`

Windows OS:

`None`

Arguments

option	Is a linker option. This option is not processed by the driver and is directly passed to the linker.

Default

OFF	No options are passed to the linker.

Description

This option passes one or more options to the linker for processing. If the linker is not invoked, these options are ignored.

This option is equivalent to specifying option `-Qoption,link,`*options*.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

None

See Also

`Qoption` compiler option

Wp

Passes options to the preprocessor.

Syntax

Linux OS:

`-Wp,`*`option1`*`[,`*`option2`*`,...]`

Windows OS:

`None`

Arguments

option	Is a preprocessor option. This option is not processed by the driver and is directly passed to the preprocessor.

Default

OFF	No options are passed to the preprocessor.

Description

This option passes one or more options to the preprocessor. If the preprocessor is not invoked, these options are ignored.

This option is equivalent to specifying option `-Qoption,`*cpp, options*.

NOTE
This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

None

See Also

`Qoption` compiler option

Xlinker

Passes a linker option directly to the linker.

Syntax

Linux OS:

`-Xlinker` *`option`*

Windows OS:

`None`

Arguments

option	Is a linker option.

Default

OFF	No options are passed directly to the linker.

Description

This option passes a linker option directly to the linker. If `-Xlinker -shared` is specified, only `-shared` is passed to the linker and no special work is done to ensure proper linkage for generating a shared object. `-Xlinker` just takes whatever arguments are supplied and passes them directly to the linker.

If you want to pass compound options to the linker, for example "`-L $HOME/lib`", you must use the following method:

```
-Xlinker -L -Xlinker $HOME/lib
```

NOTE
This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

Windows

Visual Studio: None

Linux

Eclipse: **Linker > Miscellaneous > Other Options**

Alternate Options

None

See Also

`shared` compiler option
`link` compiler option

Zl

Causes library names to be omitted from the object file.

Syntax

Linux OS:

`None`

Windows OS:

`/Zl`

Arguments

None

Default

OFF	Default or specified library names are included in the object file.

Description

This option causes library names to be omitted from the object file.

NOTE
This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

Windows

Visual Studio: **Advanced > Omit Default Library Names**

Linux

Eclipse: None

Alternate Options

None

Miscellaneous Options

This section contains descriptions for compiler options that do not pertain to a specific category. They are listed in alphabetical order.

dryrun

Specifies that driver tool commands should be shown but not executed.

Syntax

Linux OS:

`-dryrun`

Windows OS:

`None`

Arguments

None

Default

OFF	No tool commands are shown, but they are executed.

Description

This option specifies that driver tool commands should be shown but not executed.

IDE Equivalent

None

Alternate Options

None

See Also

`v` compiler option

dumpmachine

Displays the target machine and operating system configuration.

Syntax

Linux OS:

`-dumpmachine`

Windows OS:

`None`

Arguments

None

Default

OFF	The compiler does not display target machine or operating system information.

Description

This option displays the target machine and operating system configuration. No compilation is performed.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

None

See Also

`dumpversion` compiler option

dumpversion

Displays the version number of the compiler.

Syntax

Linux OS:

`-dumpversion`

Windows OS:

`None`

Arguments

None

Default

OFF	The compiler does not display the compiler version number.

Description

This option displays the version number of the compiler. It does not compile your source files.

NOTE
This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

None

See Also

`dumpmachine` compiler option

help

Displays a list of supported compiler options in alphabetical order.

Syntax

Linux OS:

`-help`

Windows OS:

`/help`

Arguments

None

Default

OFF	No list is displayed unless this compiler option is specified.

Description

This option displays a list of supported compiler options in alphabetical order.

NOTE
This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

Alternate Options

None

nologo

Tells the compiler to not display compiler version information.

Syntax

Linux OS:

`None`

Windows OS:

`/nologo`

Arguments

None

Default

OFF

Description

Tells the compiler to not display compiler version information.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

Windows

Visual Studio: **General > Suppress Startup Banner**

Linux

Eclipse: None

Alternate Options

None

save-temps, Qsave-temps

Tells the compiler to save intermediate files created during compilation.

Syntax

Linux OS:

`-save-temps`

`-no-save-temps`

Windows OS:

`/Qsave-temps (C++ only)`

`/Qsave-temps- (C++ only)`

Windows OS:

`None (SYCL only)`

Arguments

None

Default

SYCL: Linux systems: `-no-save-temps`	On Linux systems, the compiler deletes intermediate files after compilation is completed.
C++: Linux systems: `-no-save-temps` Windows systems: .obj files are saved	On Linux systems, the compiler deletes intermediate files after compilation is completed. On Windows systems, the compiler saves only intermediate object files after compilation is completed.

Description

This option tells the compiler to save intermediate files created during compilation. The names of the files saved are based on the name of the source file; the files are saved in the current working directory.

If option `[Q]save-temps` (C++) or `save-temps` (SYCL) is specified, the following occurs:

- The object .o file (Linux) is saved.
- C++: The .obj file (Windows) object .o file is saved.

If `-no-save-temps` is specified on Linux systems, the following occurs:

- The .o file is put into `/tmp` and deleted after calling `ld`.
- The preprocessed file is not saved after it has been used by the compiler.

This content does not apply to SYCL.
If `/Qsave-temps-` is specified on Windows systems, the following occurs:

- The .obj file is not saved after the linker step.
- The preprocessed file is not saved after it has been used by the compiler.

> **NOTE**
> This option only saves intermediate files that are normally created during compilation.

IDE Equivalent

None

Alternate Options

None

showIncludes

Tells the compiler to display a list of the include files.

Syntax

Linux OS:

`None`

Windows OS:

`/showIncludes`

Arguments

None

Default

OFF	The compiler does not display a list of the include files.

Description

This option tells the compiler to display a list of the include files. Nested include files (files that are included from the files that you include) are also displayed.

IDE Equivalent

Windows

Visual Studio: **Advanced > Show Includes**

Linux

Eclipse: None

Alternate Options

None

sox, Qsox

Tells the compiler to save the compilation options in the executable file.

Syntax

Linux OS:

`-sox`

Windows OS:

`/Qsox`

Arguments

None

Default

OFF	The compiler version number is saved in the object file.

Description

This option tells the compiler to save the compilation options in the executable file. The information is embedded as a string in each object file or assembly output.

When you specify this option, the size of the executable on disk is increased slightly. When you link the object files into an executable file, the linker places each of the information strings into the header of the executable. It is then possible to use a tool, such as a strings utility, to determine what options were used to build the executable file.

IDE Equivalent

None

Alternate Options

None

sysroot

Specifies the root directory where headers and libraries are located.

Syntax

Linux OS:

`--sysroot=dir`

Windows OS:

`None`

Arguments

dir	Specifies the local directory that contains copies of target libraries in the corresponding subdirectories.

Default

Off	The compiler uses default settings to search for headers and libraries.

Description

This option specifies the root directory where headers and libraries are located.

For example, if the headers and libraries are normally located in `/usr/include` and `/usr/lib` respectively, `--sysroot=/mydir` will cause the compiler to search in `/mydir/usr/include` and `/mydir/usr/lib` for the headers and libraries.

This option is provided for compatibility with gcc.

> **NOTE**
> Even though this option is not supported for a Windows-to-Windows native compiler, it is supported for a Windows-host to Linux-target compiler.

IDE Equivalent

None

Alternate Options

None

Tc

Tells the compiler to process a file as a C source file.

Syntax

Linux OS:

`None`

Windows OS:

`/Tc`*`filename`*

Arguments

filename	Is the file name to be processed as a C source file.

Default

OFF	The compiler uses default rules for determining whether a file is a C source file.

Description

This option tells the compiler to process a file as a C source file.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

None

See Also

`TC` compiler option
`Tp` compiler option

TC

Tells the compiler to process all source or unrecognized file types as C source files.

Syntax

Linux OS:

`None`

Windows OS:

`/TC`

Arguments

None

Default

OFF	The compiler uses default rules for determining whether a file is a C source file.

Description

This option tells the compiler to process all source or unrecognized file types as C source files.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

Windows

Visual Studio: **Advanced > Compile As**

Linux

Eclipse: None

Alternate Options

None

See Also

`TP` compiler option

`Tc` compiler option

Tp

Tells the compiler to process a file as a C++ source file.

Syntax

Linux OS:

`None`

Windows OS:

`/Tp`*`filename`*

Arguments

filename	Is the file name to be processed as a C++ source file.

Default

OFF	The compiler uses default rules for determining whether a file is a C++ source file.

Description

This option tells the compiler to process a file as a C++ source file.

IDE Equivalent

None

Alternate Options

None

See Also

`TP` compiler option

`Tc` compiler option

version

Tells the compiler to display GCC-style version information.

Syntax

Linux OS:

`--version`

Windows OS:

`None`

Arguments

None

Default

OFF

Description

Tells the compiler to display GCC-style version information.

> **NOTE**
> This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

None

watch

Tells the compiler to display certain information to the console output window.

Syntax

Linux OS:

`-watch[=keyword[, keyword...]]`

`-nowatch`

Windows OS:

`/watch[:keyword[, keyword...]]`

`/nowatch`

Arguments

keyword Determines what information is displayed. Possible values are:

none	Disables `cmd` and `source`.
[no]cmd	Determines whether driver tool commands are displayed and executed.
[no]source	Determines whether the name of the file being compiled is displayed.
all	Enables `cmd` and `source`.

Default

`nowatch` Pass information and source file names are not displayed to the console output window.

Description

Tells the compiler to display processing information (pass information and source file names) to the console output window.

Option **watch** *keyword*	Description
none	Tells the compiler to not display pass information and source file names to the console output window. This is the same as specifying `nowatch`.
cmd	Tells the compiler to display and execute driver tool commands.
source	Tells the compiler to display the name of the file being compiled.
all	Tells the compiler to display pass information and source file names to the console output window. This is the same as specifying `watch` with no *keyword*. For heterogeneous compilation, the tool commands for the host and the offload compilations will be displayed.

NOTE
This option only applies to host compilation. When offloading is enabled, it does not impact device-specific compilation.

IDE Equivalent

None

Alternate Options

watch cmd	Linux: -v Windows: None

See Also

v compiler option

Deprecated and Removed Compiler Options

This topic lists deprecated and removed compiler options and suggests replacement options, if any are available.

Deprecated and removed options for SYCL and C++ are listed in separate tables. There are currently no removed options for SYCL.

For more information on compiler options, see the detailed descriptions of the individual option descriptions in this section.

Deprecated Options for SYCL

Occasionally, compiler options are marked as deprecated. Deprecated options are still supported in the current release, but they may be unsupported in future releases.

The following table lists options that are currently deprecated.

Note that deprecated options are not limited to this list.

Deprecated Linux and Windows Options	Suggested Replacement
foffload-static-lib	None
fsycl-add-targets	None
fsycl-explicit-simd	None
fsycl-link-targets	None

Deprecated Options for C++

Occasionally, compiler options are marked as "deprecated." Deprecated options are still supported in the current release, but they may be unsupported in future releases.

The following two tables list options that are currently deprecated.

Note that deprecated options are not limited to these lists.

Deprecated Linux Options	Suggested Replacement
daal	qdaal
device-math-lib	None
fopenmp	None
ipp	qipp
Kc++	x c++

Deprecated Linux Options	Suggested Replacement
m32	None
march=pentiumii	None
march=pentiumiii	march=pentium3
mkl	qmkl
msse	Linux only: None
tbb	qtbb
xH	xSSE4.2

Deprecated Windows Options	Suggested Replacement
device-math-lib	None
GX	EHsc
GZ	RTC1
H	None
Oy	None
Qm32	None
Qsfalign	None
Quse-asm	None
QxH	QxSSE4.2
Ze	None
Zg	None

Removed Options for C++

Some compiler options are no longer supported and have been removed. If you use one of these options, the compiler issues a warning, ignores the option, and then proceeds with compilation.

The following two tables list options that are no longer supported.

Note that removed options are not limited to these lists.

Removed Linux Options	Suggested Replacement
A-	undef
0f_check	None
c99	std=c99
check-uninit	check=uninit
export	None

Removed Linux Options	Suggested Replacement
export-dir	None
F	P
falign-stack=mode	None
fdiv_check	None
fp	fno-omit-frame-pointer
fvisibility=internal	fvisibility=hidden
fwritable-strings	None
gcc-name and gxx-name	No exact replacement; use gcc-toolchain
guide-profile	None
i-dynamic	shared-intel
i-static	static-intel
inline-debug-info	debug inline-debug-info
ipo-obj (and -ipo_obj)	None
ipp-link=static-thread	None
Knopic, KNOPIC	fpic
Kpic, KPIC	fpic
mp	fp-model
no-alias-args	fargument-noalias
no-c99	std=c89
openmp	qopenmp
openmp-lib	qopenmp-lib
openmp-lib legacy	None
openmp-link and qopenmp-link	None
openmpP	qopenmp
openmp-profile	None
openmp-report	qopt-report-phase=openmp
openmpS	qopenmp-stubs
openmp-stubs	qopenmp-stubs
openmp-task	qopenmp-task
opt-gather-scatter-unroll	None

Removed Linux Options	Suggested Replacement
`opt-report`	`qopt-report`
`opt-streaming-cache-evict`	None
`prefetch`	`qopt-prefetch`
`print-sysroot`	None
`prof-format-32`	None
`prof-genx`	`prof-gen=srcpos`
`profile-functions`	None
`profile-loops`	None
`profile-loops-report`	None
`qopenmp-report`	`qopt-report-phase=openmp`
`qopenmp-task`	None
`qp`	`p`
`rct`	None
`shared-libcxa`	`shared-libgcc`
`ssp`	None
`static-libcxa`	`static-libgcc`
`std=c9x`	`std=c99`
`syntax`	`fsyntax-only`
`tcheck`	None
`tpp1`	None
`tpp2`	None
`tpp5`	None
`tpp6`	None
`tpp7`	None
`tprofile`	None
`Wpragma-once`	None

Removed Windows Options	Suggested Replacement
`debug:parallel`	None
`G5`	None

Removed Windows Options	Suggested Replacement
G6 (or GB)	None
G7	None
Gf	GF
ML[d]	Upgrade to MT[d]
Og	O1, O2, or O3
Op	fp:precise
QA-	u
Qc99	Qstd=c99
Qguide-profile	None
Qgpu-arch:ivybridge	None
QI0f	None
QIfdiv	None
Qinline-debug-info	debug:inline-debug-info
Qipo-obj (and Qipo_obj)	None
Qipp-link:static-thread	None
Qmspp	None
Qopenmp-lib:legacy	None
Qopenmp-link	None
Qopenmp-profile	None
Qopenmp-report	Qopt-report-phase:openmp
Qopenmp-task	None
Qopt-report-level	Qopt-report
Qprefetch	Qopt-prefetch
Qprof-format-32	None
Qprofile-functions	None
Qprofile-loops	None
Qprofile-loops-report	None
Qrct	None
Qssp	None
Qtprofile	None

Removed Windows Options	Suggested Replacement
Qtcheck	None
Qvc11 Qvc10 Qvc9 and earlier	None
YX	None
Zd	debug:minimal

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Display Option Information

To display a list of all available compiler options, specify option `help` on the command line.

To display functional groupings of compiler options, specify a functional category for option `help`. For example, to display a list of options that affect diagnostic messages, enter one of the following commands:

Linux

```
-help diagnostics
```

Windows

```
/help diagnostics
```

For details on other categories you can specify, see help.

Alternate Compiler Options

These options are not valid for SYCL applications.

This topic lists alternate names for compiler options and show the primary option name. Some of the alternate option names are deprecated and may be removed in future releases.

For more information on compiler options, see the detailed descriptions of the individual, primary options.

Some of these options are deprecated. For more information, see Deprecated and Removed Options.

Linux

Alternate Linux* Options	Primary Option Name
Code Generation:	
-fp	-fomit-frame-pointer
Advanced Optimizations:	
-funroll-loops	-unroll
OpenMP* and Parallel Processing Options:	

Alternate Linux* Options	Primary Option Name
-fopenmp	-qopenmp
Linking or Linker:	
-i-dynamic	-shared-intel
-i-static	-static-intel

Windows

Alternate Windows* Options	Primary Option Name
OpenMP* and Parallel Processing Options:	
/openmp	/Qopenmp

Portability and GCC-compatible Warning Options

This section discusses portability options and GCC-compatible warning options.

This content does not apply for SYCL.

Portability Options

A challenge in porting applications from one compiler to another is making sure that there is support for the compiler options you use to build your application. The Intel® compiler supports many of the options that are valid on other compilers you may be using.

The first table lists compiler options that are supported by the Intel® compiler and the GCC Compiler. Following this table, you will see information about GCC-compatible warning options.

The second table lists compiler options that are supported by the Intel® compiler and the Microsoft C++ Compiler .

Options that are unique to either compiler are not listed in this topic.

Linux

This table lists compiler options that are supported by both the Intel® compiler and the GCC Compiler.

-ansi
-B
-C
-c
-D
-dD
-dM
-E
-fargument-noalias
-fargument-noalias-global
-fcf-protection

```
-fdata-sections

-ffunction-sections

-f[no-]builtin

-f[no-]common

-f[no-]freestanding

-f[no-]gnu-keywords

-f[no-]inline

-f[no-]inline-functions

-f[no-]math-errno

-f[no-]operator-names

-f[no-]stack-protector

-f[no-]unsigned-bitfields

-fpack-struct

-fpermissive

-fPIC

-fpic

-freg-struct-return

-fshort-enums

-fsyntax-only

-funroll-loops

-funsigned-char

-fverbose-asm

-H

-help

-I

-idirafter

-imacros

-iprefix

-iwithprefix

-iwithprefixbefore

-l

-L

-M
```

-malign-double

-march

-mcpu

-MD

-MF

-MG

-MM

-MMD

-m[no-]ieee-fp

-MP

-MQ

-msse

-msse2

-msse3

-MT

-nodefaultlibs

-nostartfiles

-nostdinc

-nostdinc++

-nostdlib

-o

-O

-O0

-O1

-O2

-O3

-Os

-p

-P

-S

-shared

-static

-std

```
-trigraphs

-U

-u

-v

-V

-Wall

-Werror

-W[no-]cast-qual

-W[no-]comment

-W[no-]comments

-W[no-]deprecated

-W[no-]fatal-errors

-W[no-]format-security

-W[no-]main

-W[no-]missing-declarations

-W[no-]missing-prototypes

-W[no-]overflow

-W[no-]overloaded-virtual

-W[no-]pointer-arith

-W[no-]return-type

-W[no-]strict-prototypes

-W[no-]trigraphs

-W[no-]uninitialized

-W[no-]unknown-pragmas

-W[no-]unused-function

-W[no-]unused-variable

-X

-x assembler-with-cpp

-x c

-x c++

-Xlinker
```

The Intel® compiler recognizes many GCC-compatible warning options, but many are not documented.

In general, if a GCC-compatible option is accepted by the compiler, but not documented, the implementation of the option is the same as described in the GCC documentation.

To find the GCC documentation about GCC warning options, you can do any of the following:

- Enter the command:

```
man gcc
```

- Check the GCC website.
- Search the web for "gcc warning options".

Windows

This table lists compiler options that are supported by both the Intel® compiler and the Microsoft C++ Compiler.

For complete details about these options, such as the possible values for <n> when it appears below, see the Microsoft Visual Studio C++ documentation.

/C
/c
/D<name>{=|#}<text>
/E
/EH{a|s|c|r}
/EP
/F<n>
/Fa[file]
/FA[{c|s|cs}]
/FC
/Fe<file>
/FI<file>
/Fm[<file>]
/Fo<file>
/fp:<model>
/Fp<file>
/FR[<file>]
/GA
/Gd
/GF
/Gr
/GR[-]
/GS[-]
/Gs[<n>]

/Gy[-]

/Gz

/GZ

/H<n>

/help

/I<dir>

/J

/LD

/LDd

/link

/MD

/MDd

/MT

/MTd

/nologo

/O1

/O2

/Od

/Oi[-]

/Os

/Ot

/Ox

/Oy[-]

/P

/QIfist[-]

/RTC{1|c|s|u}

/showIncludes

/TC

/Tc<source file>

/TP

/Tp<source file>

/u

/U<name>

```
/vd<n>

/vmg

/vmv

/W<n>

/Wall

/WX

/X

/Y-

/Yc[<file>]

/Yu[<file>]

/Z7

/Zc:<arg1>[, <arg2>]

/Zg

/Zi

/ZI

/Zl

/Zp[<n>]

/Zs
```

Floating-Point Operations

This section contains information about floating-point operations, including IEEE floating-point operations, and it provides guidelines that can help you improve the performance of floating-point applications.

Programming Tradeoffs in Floating-Point Applications

In general, the programming objectives for floating-point applications fall into the following categories:

- **Accuracy:** The application produces results that are close to the correct result.
- **Reproducibility and portability:** The application produces consistent results across different runs, different sets of build options, different compilers, different platforms, and different architectures.
- **Performance:** The application produces fast, efficient code.

Based on the goal of an application, you will need to make tradeoffs among these objectives. For example, if you are developing a 3D graphics engine, performance may be the most important factor to consider, with reproducibility and accuracy as secondary concerns.

The compiler provides several options that allow you to tune your applications based on specific objectives. Broadly speaking, there are the floating-point specific options, such as the `-fp-model` (Linux*) or `/fp` (Windows*) option, and the fast-but-low-accuracy options, such as the `[Q]imf-max-error` option. The compiler optimizes and generates code differently when you specify these different compiler options. Select appropriate compiler options by carefully balancing your programming objectives and making tradeoffs among these objectives. Some of these options may influence the choice of math routines that are invoked.

Many routines in the *libirc*, *libm*, and *svml* library are more highly optimized for Intel microprocessors than for non-Intel microprocessors.

Use Floating-Point Options

Take the following code as an example:

```
float t0, t1, t2;
 ...
t0=t1+t2+4.0f+0.1f;
```

If you specify the `-fp-model fast` (Linux) or `/fp:fast` (Windows) option in favor of performance, the compiler generates the following assembly code:

```
movss     xmm0, DWORD PTR _Cnst4.1
addss     xmm0, DWORD PTR _t1
addss     xmm0, DWORD PTR _t2
movss     DWORD PTR _t0, xmm0
```

This code maximizes performance using Intel® Streaming SIMD Extensions (Intel® SSE) instructions and pre-computing `4.0f + 0.1f`. It is not as accurate as the first implementation, due to the greater intermediate rounding error. It does not provide reproducible results like the second implementation, because it must reorder the addition to pre-compute `4.0f + 0.1f`. All compilers, on all platforms, at all optimization levels do not reorder the addition in the same way.

For many other applications, the considerations may be more complicated.

Use Fast-But-Low-Accuracy Options

The fast-but-low-accuracy options provide an easy way to control the accuracy of mathematical functions and utilize performance/accuracy tradeoffs offered by the Intel® oneAPI Math Kernel Library (oneMKL). You can specify accuracy, via a command line interface, for all math functions or a selected set of math functions at the level more precise than low, medium, or high.

You specify the accuracy requirements as a set of function attributes that the compiler uses for selecting an appropriate function implementation in the math libraries. Examples using the attribute, `max-error`, are presented here. For example, use the following option to specify the relative error of two ULPs for all single, double, long double, and quad precision functions:

```
-fimf-max-error=2
```

To specify twelve bits of accuracy for a `sin` function, use:

```
-fimf-accuracy-bits=12:sin
```

To specify relative error of ten ULPs for a `sin` function, and four ULPs for other math functions called in the source file you are compiling, use:

```
-fimf-max-error=10:sin-fimf-max-error=4
```

On Windows systems, the compiler defines the default value for the `max-error` attribute depending on the `/fp` option settings. In `/fp:fast` mode the compiler sets a `max-error=`*4.0* for the call. Otherwise, it sets a `max-error=`*0.6*.

Dispatching of Math Routines

The compiler optimizes calls to routines from the *libm* and *svml* libraries into direct CPU-specific calls, when the compilation configuration specifies the target CPU where the code is tuned, and if the set of instructions available for the code compilation is not narrower than the set of instructions available in the tuning target CPU.

The dispatching optimization applies to the *exp()* routine, and to the other math routines with CPU specific implementations in the libraries. The dispatching optimization can be disabled using the `-fimf-force-dynamic-target` (or `Qimf-force-dynamic-target`) option. This option specifies a list of math routines that are improved with a dynamic dispatcher.

See Also

Using -fp-model(/fp) Options
fimf-max-error, Qimf-max-error compiler option

Use the -fp-model, /fp Option

The `-fp-model` (Linux) or `/fp` (Windows) option allows you to control the optimizations on floating-point data. You can use this option to tune the performance, level of accuracy, or result consistency for floating-point applications across platforms and optimization levels.

You can use keywords to specify the semantics to be used. The keywords specified for this option may influence the choice of math routines that are invoked. Many routines in the *libirc*, *libm*, and *libsvml* libraries are more highly optimized for Intel microprocessors than for non-Intel microprocessors. Possible values of the keywords are as follows:

Keyword	Description
`precise`	Enables value-safe optimizations on floating-point data.
`fast`	Enables more aggressive optimizations on floating-point data.
`strict`	Enables `precise` , disables contractions, and enables pragma `stdc fenv_access`.

NOTE
Using the default option keyword `-fp-model fast` or `/fp:fast`, you may get significant differences in your result depending on whether the compiler uses x87 or Intel® Streaming SIMD Extensions (Intel® SSE)/Intel® Advanced Vector Extensions (Intel® AVX) instructions to implement floating-point operations. Results are more consistent when the other option keywords are used.

See Also

`fp-model, fp` compiler option

Denormal Numbers

A normalized number is a number for which both the exponent (including bias) and the most significant bit of the mantissa are non-zero. For such numbers, all the bits of the mantissa contribute to the precision of the representation.

The smallest normalized single-precision floating-point number greater than zero is about 1.1754943^{-38}. Smaller numbers are possible, but those numbers must be represented with a zero exponent and a mantissa whose leading bit(s) are zero, which leads to a loss of precision. These numbers are called denormalized numbers or denormals(newer specifications refer to these as subnormal numbers).

Denormal computations use hardware and/or operating system resources to handle denormals; these can cost hundreds of clock cycles. Denormal computations take much longer to calculate than normal computations.

There are several ways to avoid denormals and increase the performance of your application:

- Scale the values into the normalized range.
- Use a higher precision data type with a larger range.
- Flush denormals to zero.

See Also

Reducing Impact of Denormal Exceptions

Intel® 64 and IA-32 (for C++ only) Architectures Software Developer's Manual, Volume 1: Basic Architecture

Institute of Electrical and Electronics Engineers, Inc*. (IEEE) web site for information about the current floating-point standards and recommendations

Set the FTZ and DAZ Flags

In Intel® processors, the flush-to-zero (`FTZ`) and denormals-are-zero (`DAZ`) flags in the `MXCSR` register are used to control floating-point calculations. Intel® Streaming SIMD Extensions (Intel® SSE) and Intel® Advanced Vector Extensions (Intel® AVX) instructions, including scalar and vector instructions, benefit from enabling the `FTZ` and `DAZ` flags. Floating-point computations using the Intel® SSE and Intel® AVX instructions are accelerated when the `FTZ` and `DAZ` flags are enabled. This improves the application's performance.

Manually set the `FTZ` flags with the following macros:

```
_MM_SET_FLUSH_ZERO_MODE(_MM_FLUSH_ZERO_ON)
```

Manually set the `DAZ` flags with the following macros:

```
_MM_SET_DENORMALS_ZERO_MODE(_MM_DENORMALS_ZERO_ON)
```

The prototypes for these macros are in `xmmintrin.h` (`FTZ`) and `pmmintrin.h` (`DAZ`).

Tuning Performance

This section describes several programming guidelines that can help you improve the performance of floating-point applications, including:

- Handling Floating-point Array Operations in a Loop Body
- Reducing the Impact of Denormal Exceptions
- Avoiding Mixed Data Type Arithmetic Expressions
- Using Efficient Data Types

Handling Floating-point Array Operations in a Loop Body

Following the guidelines below will help auto-vectorization of the loop.

- Statements within the loop body may contain float or double operations (typically on arrays). The following arithmetic operations are supported: addition, subtraction, multiplication, division, negation, square root, MAX, MIN, and mathematical functions such as SIN and COS.
- Writing to a single-precision scalar/array and a double scalar/array within the same loop decreases the chance of auto-vectorization due to the differences in the vector length (that is, the number of elements in the vector register) between float and double types. If auto-vectorization fails, try to avoid using mixed data types.

> **NOTE**
> The special `__m64`, `__m128`, and `__m256` datatypes are not vectorizable. The loop body cannot contain any function calls. Use of the Intel® Streaming SIMD Extensions (Intel® SSE) and Intel® Advanced Vector Extensions (Intel® AVX) intrinsics (for example, `mm_add_ps`) is not allowed.

Reducing the Impact of Denormal Exceptions

Denormalized floating-point values are those that are too small to be represented in the normal manner; that is, the mantissa cannot be left-justified. Denormal values require hardware or operating system interventions to handle the computation, so floating-point computations that result in denormal values may have an adverse impact on performance.

There are several ways to handle denormals to increase the performance of your application:

- Scale the values into the normalized range
- Use a higher precision data type with a larger range
- Flush denormals to zero

For example, you can translate them to normalized numbers by multiplying them using a large scalar number, doing the remaining computations in the normal space, then scaling back down to the denormal range. Consider using this method when the small denormal values benefit the program design.

Consider using a higher precision data type with a larger range; for example, by converting variables declared as `float` to be declared as `double`. Understand that making the change can potentially slow down your program. Storage requirements will increase, which will increase the amount of time for loading and storing data from memory. Higher precision data types can also decrease the potential throughput of Intel® Streaming SIMD Extensions (Intel® SSE) and Intel® Advanced Vector Extensions (Intel® AVX) operations.

If you change the type declaration of a variable, you might also need to change associated library calls, unless these are generic; ; for example, `cos()` instead of `cosf()`.. You should verify that the gain in performance from eliminating denormals is greater than the overhead of using a data type with higher precision and greater dynamic range.

In many cases, denormal numbers can be treated safely as zero without adverse effects on program results. Depending on the target architecture, use flush-to-zero (`FTZ`) options.

Avoiding Mixed Data Type Arithmetic Expressions

Avoid mixing integer and floating-point (float, double, or long double) data in the same computation. Expressing all numbers in a floating-point arithmetic expression (assignment statement) as floating-point values eliminates the need to convert data between fixed and floating-point formats. Expressing all numbers in an integer arithmetic expression as integer values also achieves this. This improves run-time performance.

For example, assuming that `I` and `J` are both int variables, expressing a constant number (2.0) as an integer value (2) eliminates the need to convert the data. The following examples demonstrate inefficient and efficient code.

Inefficient code:

```
int I, J;
  I = J / 2.0
;
```

Efficient code:

```
int I, J;
  I = J / 2;
```

Using Efficient Data Types

In cases where more than one data type can be used for a variable, consider selecting the data types based on the following hierarchy, listed from most to least efficient:

- char
- short
- int
- long

- long long
- float
- double
- long double

NOTE
In an arithmetic expression, you should avoid mixing integer and floating-point data.

You can use integer data types (*int*, *int long*, etc.) in loops to improve floating point performance. Convert the data type to integer data types, process the data, then convert the data to the old type.

See Also

Programming Guidelines for Vectorization
Setting the FTZ and DAZ Flags

Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture

IEEE Floating-point Operations

Understanding the IEEE Standard for Floating-point Arithmetic, IEEE 754-2008

This version of the compiler uses a close approximation to the IEEE Standard for Floating-point Arithmetic, version IEEE 754-2008, unless otherwise stated. This standard is common to many microcomputer-based systems due to the availability of fast processors that implement the required characteristics.

This section outlines the characteristics of the IEEE 754-2008 standard and its implementation in the compiler. Except as noted, the description refers to both the IEEE 754-2008 standard and the compiler implementation.

Special Values

The following list provides a brief description of the special values that the Intel® oneAPI DPC++/C++ Compiler supports.

- **Signed Zero:** The sign of zero is the same as the sign of a nonzero number. Comparisons consider +0 to be equal to -0. A signed zero is useful in certain numerical analysis algorithms, but in most applications the sign of zero is invisible.
- **Denormalized Numbers:** Denormalized numbers (denormals) fill the gap between the smallest positive and the smallest negative normalized number, otherwise only (+/-) 0 occurs in the interval. Denormalized numbers extend the range of computable results by allowing for gradual underflow.

  Systems based on the IA-32 architecture support a Denormal Operand status flag. When this is set, at least one of the input operands to a Floating-point operation is a denormal. The Underflow status flag is set when a number loses precision and becomes a denormal.
- **Signed Infinity:** Infinities are the result of arithmetic in the limiting case of operands with arbitrarily large magnitude. They provide a way to continue when an overflow occurs. The sign of an infinity is simply the sign you obtain for a finite number in the same operation as the finite number approaches an infinite value.

  By retrieving the status flags, you can differentiate between an infinity that results from an overflow and one that results from division by zero. The compiler treats infinity as signed by default. The output value of infinity is +Infinity or -Infinity.
- **Not a Number:** Not a Number (NaN) may result from an invalid operation. For example, `0/0` and `SQRT(-1)` result in NaN. In general, an operation involving a NaN produces another NaN. Because the fraction of a NaN is unspecified, there are many possible NaNs

The compiler treats all NaNs identically, but there are two classes of NaNs:

- Signaling NaNs: Have an initial mantissa bit of 0. They usually raise an invalid exception when used in an operation.
- Quiet NaNs: Have an initial mantissa bit of 1.

The floating-point hardware usually converts a signaling NaN into a quiet NaN during computational operations. An invalid exception is raised and the resulting Floating-point value is a quiet NaN.

Attributes

Attributes are a way to provide additional information about a declaration to the compiler. The C+11 attribute syntax is consistent with the C2x standard.

Use Attributes

The compiler supports three ways to add attributes to your program:

- **Gnu Syntax**

```
__attribute__((attribute_name(arguments)))
```

- **Microsoft Syntax**

```
__declspec(attribute_name(argument))
```

- **C++11 Standardized Attribute Syntax** (part of the C++11 language standard)

```
[[attribute_name(arguments)]]
```

```
[[attribute-namespace :: attribute_name(arguments)]]
```

Some attributes are available for both Intel® microprocessors and non-Intel microprocessors but they may perform additional optimizations for Intel® microprocessors than they perform for non-Intel microprocessors. Refer to the individual attribute name for a detailed description.

align

Directs the compiler to align the variable to a specified boundary and a specified offset.

Syntax

Windows* OS:

`__declspec(align(`*n*`))`

Linux* OS:

`__attribute__((aligned(`*n*`)))`

For portability on Linux OS, you should use the syntax form `__attribute__((aligned(n)))`. This form is compatible with the GNU compiler.

Arguments

n	Specifies the alignment. The compiler will align the variable to an *n*-byte boundary.

Description

This keyword directs the compiler to align the variable to an *n*-byte boundary.

NOTE
If you require 8-byte alignment, we recommend you specify 16 for *n*, instead of 8. When 8 is used, the compiler interprets the value as a suggestion and you may not get the requested 8-byte alignment, depending on various heuristics.

align_value

Provides the ability to add a pointer alignment value to a pointer typedef declaration.

Syntax

Windows* OS:

`__declspec(align_value(alignment))`

Linux* OS:

`__attribute__((align_value(alignment)))`

Arguments

alignment	Specifies the alignment (8, 16, 32, 64, 128, 256,...) for what the pointer points to.

Description

This keyword can be added to a pointer typedef declaration to specify the alignment value of pointers declared for that pointer type.

It tells the compiler that the data referenced by the designated pointer is aligned by the indicated value, and the compiler can generate code based on that assumption. If this attribute is used incorrectly, and the data is not aligned to the designated value, the behavior is undefined.

allow_cpu_features

Provides the ability for a function to use intrinsic functions and architecture specific functionality.

Syntax

Windows* OS:

`__declspec(allow_cpu_features(featp1[,featp2]))`

Linux* OS:

`__attribute__((allow_cpu_features(featp1[,featp2])))`

Arguments

featp1

Specifies features to allow for the function. Values are integral constant expressions that evaluate to the page one bitmask of permissible features from the libirc CPUID information. The evaluated type is an unsigned 64-bit integer which permits use of template-dependent code. Possible values are:

- `_FEATURE_GENERIC_IA32`
- `_FEATURE_FPU`
- `_FEATURE_CMOV`

- _FEATURE_MMX
- _FEATURE_FXSAVE
- _FEATURE_SSE
- _FEATURE_SSE2
- _FEATURE_SSE3
- _FEATURE_SSSE3
- _FEATURE_SSE4_1
- _FEATURE_SSE4_2
- _FEATURE_MOVBE
- _FEATURE_POPCNT
- _FEATURE_PCLMULQDQ
- _FEATURE_AES
- _FEATURE_F16C
- _FEATURE_AVX
- _FEATURE_RDRND
- _FEATURE_FMA
- _FEATURE_BMI
- _FEATURE_LZCNT
- _FEATURE_HLE
- _FEATURE_RTM
- _FEATURE_AVX2
- _FEATURE_AVX512DQ
- _FEATURE_PTWRITE
- _FEATURE_AVX512F
- _FEATURE_ADX
- _FEATURE_RDSEED
- _FEATURE_AVX512IFMA52
- _FEATURE_AVX512ER
- _FEATURE_AVX512PF
- _FEATURE_AVX512CD
- _FEATURE_SHA
- _FEATURE_MPX
- _FEATURE_AVX512BW
- _FEATURE_AVX512VL
- _FEATURE_AVX512VBMI
- _FEATURE_AVX512_4FMAPS
- _FEATURE_AVX512_4VNNIW
- _FEATURE_AVX512_VPOPCNTDQ
- _FEATURE_AVX512_BITALG
- _FEATURE_AVX512_VBMI2
- _FEATURE_GFNI
- _FEATURE_VAES
- _FEATURE_VPCLMULQDQ
- _FEATURE_AVX512_VNNI
- _FEATURE_CLWB
- _FEATURE_RDPID
- _FEATURE_IBT
- _FEATURE_SHSTK
- _FEATURE_SGX

- `_FEATURE_WBNOINVD`
- `_FEATURE_PCONFIG`
- `_FEATURE_AXV512_VP2INTERSECT`

featp2

Optional. Specifies features to allow for the function. Values are integral constant expressions that evaluate to the page two bitmask of permissible features from the libirc CPUID information. The evaluated type is an unsigned 64-bit integer which permits use of template-dependent code. If only features from page two are desired, specify 0 for *featp1*. Possible values are:

- `_FEATURE_CLDEMOTE`
- `_FEATURE_MOVDIRI`
- `_FEATURE_MOVDIR64B`
- `_FEATURE_WAITPKG`
- `_FEATURE_AVX512_Bf16`
- `_FEATURE_ENQCMD`
- `_FEATURE_AVX_VNNI`
- `_FEATURE_AMX_TILE`
- `_FEATURE_AMX_INT8`
- `_FEATURE_AMX_BF16`
- `_FEATURE_KL`
- `_FEATURE_WIDE_KL`

Description

This keyword can be added to a function to specify intrinsic functions and architecture specific functionality that the function is allowed to use. The function is generated as if the specified features are available.

concurrency_safe

Guides the compiler to parallelize more loops and straight-line code.

Syntax

Windows* OS:

`__declspec(concurrency_safe(`*clause*`))`

Linux* OS:

`__attribute__((concurrency_safe(`*clause*`)))`

Arguments

clause

Is one of the following:

cost(cycles): Specifies the execution cycles of the annotated function for the compiler to perform parallelization profitability analysis while compiling its enclosing loops or blocks. The value of `cycles` is a 2-byte unsigned integer (unsigned short); its maximal value is 2^16-1. If the cycle count is greater than 2^16-1, you should use *profitable*.

profitable: Specifies that the loops or blocks that contain calls to the annotated function are profitable to parallelize.

Description

This keyword specifies that there are no incorrect side-effects and no illegal (or improperly synchronized) memory access interferences among multiple invocations of the annotated function or between an invocation of this annotated function and other statements in the program, if they are executed concurrently.

For every function that is marked with this keyword, you must ensure that its side effects (if any) are acceptable (or expected), and the memory access interferences are properly synchronized.

const

Indicates that a function has no effect other than returning a value and that it uses only its arguments to generate that return value.

Syntax

Windows* OS:

`__declspec(const)`

Linux* OS:

`__attribute__((const))`

Arguments

None

Description

This keyword is equivalent to the gcc* attribute `const` and applies to function declarations.

cpu_dispatch, cpu_specific

Provides the ability to write one or more versions of a function that execute only on a list of targeted processors (cpu_dispatch). Provides the ability to declare that a version of a function is targeted at particular types of processors (cpu_specific).

Syntax

Windows* OS:

`__declspec(cpu_dispatch(`*`cpuid, cpuid, ...`*`))`

`__declspec(cpu_specific(`*`cpuid`*`))`

Linux* OS:

`__attribute__((cpu_dispatch(`*`cpuid, cpuid, ...`*`)))`

`__attribute__((cpu_specific(`*`cpuid`*`)))`

Arguments

cpuid	Possible values are: `atom`: Intel® Atom™ processors with Intel® Supplemental Streaming SIMD Extensions 3 (Intel® SSSE3) `atom_sse4_2`: Intel® Atom™ processors with Intel® Streaming SIMD Extensions 4.2 (Intel® SSE4.2)

`atom_sse4_2_movbe`: Intel® Atom™ processors with Intel® Streaming SIMD Extensions 4.2 (Intel® SSE4.2) with MOVBE instructions enabled

`broadwell`: This is a synonym for `core_5th_gen_avx`

`core_2nd_gen_avx`: 2nd generation Intel® Core™ processor family with support for Intel® Advanced Vector Extensions (Intel® AVX)

`core_3rd_gen_avx`: 3rd generation Intel® Core™ processor family with support for Intel® Advanced Vector Extensions (Intel® AVX) including the RDRND instruction

`core_4th_gen_avx`: 4th generation Intel® Core™ processor family with support for Intel® Advanced Vector Extensions 2 (Intel® AVX2) including the RDRND instruction

`core_4th_gen_avx_tsx`: 4th generation Intel® Core™ processor family with support for Intel® Advanced Vector Extensions 2 (Intel® AVX2) including the RDRND instruction, and support for Intel® Transactional Synchronization Extensions (Intel® TSX)

`core_5th_gen_avx`: 5th generation Intel® Core™ processor family with support for Intel® Advanced Vector Extensions 2 (Intel® AVX2) including the RDSEED and Multi-Precision Add-Carry Instruction Extensions (ADX) instructions

`core_5th_gen_avx_tsx`: 5th generation Intel® Core™ processor family with support for Intel® Advanced Vector Extensions 2 (Intel® AVX2) including the RDSEED and Multi-Precision Add-Carry Instruction Extensions (ADX) instructions, and support for Intel® Transactional Synchronization Extensions (Intel® TSX)

`core_aes_pclmulqdq`: Intel® Core™ processors with support for Advanced Encryption Standard (AES) instructions and carry-less multiplication instruction

`core_i7_sse4_2`: Intel® Core™ i7 processors with Intel® Streaming SIMD Extensions 4.2 (Intel® SSE4.2) instructions

`generic`: Other Intel processors for IA-32 (for C++ only) or Intel® 64 architecture or compatible processors not provided by Intel Corporation

`haswell`: This is a synonym for `core_4th_gen_avx`

`pentium`: Intel® Pentium® processor

`pentium_4`: Intel® Pentium® 4 processors

`pentium_4_sse3`: Intel® Pentium® 4 processor with Intel® Streaming SIMD Extensions 3 (Intel® SSE3) instructions, Intel® Core™ Duo processors, Intel® Core™ Solo processors

`pentium_ii`: Intel® Pentium® II processors

`pentium_iii`: Intel® Pentium® III processors

`pentium_iii_no_xmm_regs`: Intel® Pentium® III processors with no XMM registers

`pentium_m`: Intel® Pentium® M processors

`pentium_mmx`: Intel® Pentium® processors with MMX™ technology `pentium_pro`: Intel® Pentium® Pro processors

Description

Use the `cpu_dispatch` keyword to provide a list of targeted processors, along with an empty function body/ function stub.

Use the `cpu_specific` keyword to declare each function version targeted at particular type of processor.

These features are available only for Intel processors based on IA-32 (for C++ only) or Intel® 64 architecture. They are not available for non-Intel processors. Applications built using the manual processor dispatch feature may be more highly optimized for Intel processors than for non-Intel processors.

See Also

mpx

Directs the compiler to pass Intel® Memory Protection Extensions (Intel® MPX) bounds information along with any pointer-typed parameters.

Syntax

Windows* OS:

`__declspec(mpx)`

Linux* OS:

`__attribute__((mpx))`

Arguments

None

Description

When a function declared with this keyword is called, any pointer-typed parameters passed to the function will also have Intel® MPX bounds information passed. If the called function returns a pointer-typed object, the compiler will expect the function to return Intel® MPX bounds information along with the pointer object. Similarly, if this keyword is applied to a function definition, the function will expect the caller to pass Intel® MPX bounds information along with any pointer-type parameters. If the function returns a pointer-typed object, Intel® MPX bounds information will be returned with the object.

NOTE
The usage of this attribute is intended for Windows code that contains hand-written Intel® MPX enhancements based on Intel® MPX inline assembly or calls to Intel® MPX intrinsics, and where the user does not wish to enable automatic Intel® MPX code generation.

target

Specifies a target for called functions or variables.

Syntax

Windows* OS:

`__declspec(target(`*`target-name`*`))`

Linux* OS:

`__attribute__((target(*target-name*)))`

Arguments

target-name	Specifies the target name. Possible values are: • `arch=skylake-avx512` • `arch=corei7` • `arch=core2` • `arch=atom` • `mmx` • `sse` • `sse2` • `sse3` • `ssse3` • `sse4.1` • `sse4.2` • `popcnt` • `aes` • `pclmul` • `avx` • `avx2` • `avx512f`

Description

This keyword specifies that the called function or variable is also available on the target. Only functions or variables marked with this attribute are available on the target, and only these functions can be called on the target.

Intrinsics

A detailed introduction and information about Intel intrinsics is provided in the Intel® C++ Compiler Classic Developer Guide and Reference. The Intel® Intrinsics Guide provides detailed information and a lookup tool for viewing the available Intel intrinsics.

The following is some general information:

- Intrinsics are assembly-coded functions that let you use C++ function calls and variables in place of assembly instructions.
- Intrinsics can be used only on the host.
- Intrinsics are expanded inline eliminating function call overhead. Providing the same benefit as using inline assembly, intrinsics improve code readability, assist instruction scheduling, and help reduce debugging.
- Intrinsics provide access to instructions that cannot be generated using the standard constructs of the C and C++ languages.

NOTE
To use intrinsic-based code with the Intel® oneAPI DPC++/C++ Compiler, do the following:

- Specify compiler option `march` so that the compiler recognizes the processor-specific or architecture-specific intrinsic.
- Include the immintrin.h header file that comes with the intrinsic declarations.

Availability of Intrinsics on Intel Processors

Not all Intel® processors support all intrinsics. For information on which intrinsics are supported on Intel® processors, visit the Product Specification, Processors page. The Processor Spec Finder tool links directly to all processor documentation and the datasheets list the features, including intrinsics, supported by each processor.

Libraries

The Intel® oneAPI DPC++/C++ Compiler lets you use all the standard run-time libraries that are part of Microsoft* Visual C++*. The options described in this section can help you determine which libraries your application uses.

To create libraries, use the `lib.exe` tool or `xilib.exe` tool.

Create Libraries

Libraries are simply an indexed collection of object files that are included as needed in a linked program. Combining object files into a library makes it easy to distribute your code without disclosing the source. It also reduces the number of command-line entries needed to compile your project.

Static Libraries

Executables generated using static libraries are no different than executables generated from individual source or object files. Static libraries are not required at runtime, so you do not need to include them when you distribute your executable. At compile time, linking to a static library is generally faster than linking to individual source files.

These steps show how to build a static library on Linux using the `icpx` driver. You can alternately use the `dpcpp` driver. See Invoke the Compiler for information about all available compilers and drivers.

1. Use the `c` option to generate object files from the source files:

```
icpx -c my_source1.cpp my_source2.cpp my_source3.cpp
```

2. Use the GNU* tool `ar` to create the library file from the object files:

```
ar rc my_lib.a my_source1.o my_source2.o my_source3.o
```

3. Compile and link your project with your new library:

```
icpx main.cpp my_lib.a
```

If your library file and source files are in different directories, use the `L`*`dir`* option to indicate where your library is located:

```
icpx -L/cpp/libs main.cpp my_lib.a
```

If your library file and source files are in different directories, use the `L`*`dir`**dir* option to indicate where your library is located:

```
icpx -L/cpp/libs main.cpp my_lib.a
```

If you are using Interprocedural Optimization, see the topic Create a Library from IPO Objects, which discusses using `xiar`.

Shared Libraries

Shared libraries, also referred to as dynamic libraries or Dynamic Shared Objects (DSO), are linked differently than static libraries. At compile time, the linker insures that all the necessary symbols are either linked into the executable, or can be linked at runtime from the shared library. Executables compiled from shared libraries are smaller, but the shared libraries must be included with the executable to function correctly. When multiple programs use the same shared library, only one copy of the library is required in memory.

Linux

These steps show how to build a shared library on Linux using the `icpx` driver. You can alternately use the `dpcpp` driver. See Invoke the Compiler for information about all available compilers and drivers.

1. Use options `fPIC` and `c` to generate object files from the source files:

```
icpx -fPIC -c my_source1.cpp my_source2.cpp my_source3.cpp
```

2. Use the `shared` option to create the library file from the object files:

```
icpx -shared -o my_lib.so my_source1.o my_source2.o my_source3.o
```

3. Compile and link your project with your new library:

```
icpx main.cpp my_lib.so
```

Windows

Use the following options to create libraries on Windows:

Option	Description
`/LD`, `/LDd`	Produces a DLL. `d` indicates debug version.
`/MD`, `/MDd`	Compiles and links with the dynamic, multi-thread C run time library. `d` indicates debug version.
`/MT`, `/MTd`	Compiles and links with the static, multi-thread C run time library. `d` indicates debug version.
`/Zl`	Disables embedding default libraries in object files.

See Also

Use Intel Shared Libraries

Create a Library from IPO Objects

See Also

`/LD` compiler option
`/MD` compiler option
`/MT` compiler option

Use Intel Shared Libraries

This topic applies to Linux.

This content does not apply for SYCL.

By default, the Intel® oneAPI DPC++/C++ Compiler links Intel® C++ libraries dynamically. The GNU/Linux system libraries are also linked dynamically.

Options for Shared Libraries (Linux)

Option	Description
`-shared-intel`	Use the `shared-intel` option to link Intel®-provided libraries dynamically. This has the advantage of reducing the size of the application binary, but it also requires the libraries to be on the application's target system.
`-shared`	The `shared` option instructs the compiler to build a Dynamic Shared Object (DSO) instead of an executable. For more details, refer to the `ld` man page documentation.
`-fpic`	Use the `fpic` option when building shared libraries. It is required for the compilation of each object file included in the shared library.

Manage Libraries

Manage Libraries on Linux

During compilation, the compiler reads the `LIBRARY_PATH` environment variable for static libraries it needs to link when building the executable. At runtime, the executable will link against dynamic libraries referenced in the `LD_LIBRARY_PATH` environment variable. Add the location of your static libraries to the `LIBRARY_PATH` environment variable so that they are available for linking during compilation.

For example, to compile `file.cpp` and link it with the library `lib.a`, located in the `/libs` directory, using the `icpx` driver:

1. Add the directory `/libs` to `LIBRARY_PATH` from the command line with the export command:

```
export LIBRARY_PATH=/libs:$LIBRARY_PATH
```

   Alternately, add the directory to `LIBRARY_PATH` by addiing the export command to your startup file.
2. Compile `file.cpp` and link it with `lib.a`:

```
icpx file.cpp lib.a
```

To link your library during compilation without modifying the `LIBRARY_PATH` environment variable use the `-L` option. For example:

```
icpx file.cpp -L /libs lib.a
```

During compilation, the compiler passes object files to the linker in the following order:

1. Object files, from files specified on the command line, in the order they are specified (left to right)
2. Objects or libraries specified in default configuration files
3. Default Intel and system libraries

For example, the command

```
icpx lib1.a file.cpp lib2.a
```

would have the following link order:

1. `lib1.a`
2. `file.o`
3. `lib2.a`
4. Objects or libraries specified in default configuration files
5. Default Inel and system libraries

Compile with SYCL and Link Other Compilers

When you use the Intel® DPC++ Compiler and source its entire environment, then linking works correctly with other compilers if the correct path to the compiler libraries is set. This allows programs to be compiled with SYCL and then linked with other compilers (example: `gcc`). If you try to do this without sourcing the compiler environment, the linking fails with undefined references in `libsycl.so` and other internal libraries.

To resolve this, add the following paths to `LD_LIBRARY_PATH`:

```
<install_dir>/compiler/latest/linux/compiler/lib/intel64

<install_dir>/compiler/latest/linux/lib

<install_dir>/compiler/latest/linux/lib/x64

<install_dir>/tbb/latest/lib/intel64/gcc4.8
```

Manage Libraries on Windows

The `LIB` environment variable contains a semicolon-separated list of directories in which the Microsoft linker will search for library (`.lib`) files. The compiler does not specify library names to the linker but includes directives in the object file to specify the libraries to be linked with each object.

For more information on adding library names to the response file and the configuration file, see Using Response Files and Using Configuration Files.

To specify a library name on the command line, you must first add the library's path to the `LIB` environment variable. Then you can specify the library name on the command line. For example, to compile `file.cpp` and link it with the library `mylib.lib`with the Intel® C++ Compiler, enter the command:

```
icx file.cpp mylib.lib
```

Other Considerations

The Intel Compiler Math Libraries contain performance-optimized implementations for various Intel platforms. By default, the best implementation for the underlying hardware is selected at runtime. The library dispatch of multi-threaded code may lead to apparent data races, which may be detected by certain software analysis tools. However, as long as the threads are running on cores with the same CPUID, these data races are harmless and are not a cause for concern.

Redistribute Libraries When Deploying Applications

When you deploy your application to systems that do not have a compiler installed, you need to redistribute certain Intel® libraries where your application is linked. You can do so in one of the following ways:

- Statically link your application.

  An application built with statically-linked libraries eliminates the need to distribute runtime libraries with the application executable. By linking the application to the static libraries, you are not dependent on the Intel® Fortran or Intel® C/C++ dynamic shared libraries.
- Dynamically link your application.

  If you must build your application with dynamically linked (or shared) compiler libraries, you should address the following concerns:

  - You must build your application with shared or dynamic libraries that are redistributable.
  - Pay careful attention to the directory where the redistributables are installed and how the OS finds them.
  - You should determine which shared or dynamic libraries your application needs.

The information here is only introductory. The redistributable library installation packages are available at the following locations:

- Intel® oneAPI versions
- Older Intel® Parallel Studio XE versions

Resolve References to Shared Libraries

If you are relying on shared libraries distributed with Intel® oneAPI tools, you must make sure that your users have these shared libraries on their systems.

If you are building an application that will be deployed to your user community and you are relying on shared libraries (`.so` shared objects on Linux, `.dll` dynamic libraries on Windows) distributed with Intel® oneAPI tools, you must make sure that your users have these shared libraries on their systems. To determine what shared libraries you depend on, use one of the following commands for each of your programs and components:

Linux

```
ldconfig
```

Windows

```
dumpbin /DEPENDENTS programOrComponentName
```

Once you have done this, you must choose how your users will receive these libraries.

Shared Library Deployment

Once you have built, run, and debugged your application, you must deploy it to your users. That deployment includes any shared libraries, including libraries that are components of the Intel® oneAPI toolkits.

Deployment Models

You have two options for deploying the shared libraries from the Intel oneAPI toolkit that your application depends on:

Private Model	Copy the shared libraries from the Intel oneAPI toolkit into your application environment, and then package and deploy them with your application. Review the license and third-party files associated with the Intel oneAPI toolkits and/or components you have installed to determine the files that you can redistribute. The advantage to this model is that you have control over your library and version choice, so you only package and deploy the libraries that you have tested. The disadvantage is that the end users may see multiple libraries installed on their system, if multiple installed applications all use the private model. You are also responsible for updating these libraries whenever updates are required.
Public Model	You direct your users to runtime packages provided by Intel. Your users install these packages on their system when they install your application. The runtime packages install onto a fixed location, so all applications built with Intel oneAPI tools can be used. The advantage is that one copy of each library is shared by all applications, which results in improved performance. You can rely on updates to the runtime packages to resolve issues with libraries independently from when you update your application. The disadvantage is that the footprint of the runtime package is larger than a package from the private model. Another disadvantage is that your tested versions of the runtime libraries may not be the same as your end user's versions.

Select the model that best fits your environment, your needs, and the needs of your users.

NOTE Intel ensures that newer compiler-support libraries work with older versions of generated compiler objects, but newer versioned objects require newer versioned compiler-support libraries. If an incompatibility is introduced that causes newer compiler-support libraries not to work with older compilers, you will have sufficient warning and the library will be versioned so that deployed applications continue to work.

Additional Steps

Under either model, you must manually configure certain environment variables that are normally handled by the `setvars`/`vars` scripts or module files.

For example, with the Intel® MPI Library, you must set the following environment variables during installation:

Linux

```
I_MPI_ROOT=installPath FI_PROVIDER_PATH=installPath/intel64/libfabric:/usr/lib64/libfabric
```

Windows

```
I_MPI_ROOT=installPath
```

Compatibility in the Minor Releases of the Intel oneAPI Products

For Intel oneAPI products, each minor version of the product is compatible with the other minor version from the same release (for example, 2021). When there are breaking changes in API or ABI, the major version is increased. For example, if you tested your application with an Intel oneAPI product with a 2021.1 version, it will work with all 2021.x versions. It is not guaranteed that it will work with 2022.x or 19.x versions.

Intel's Memory Allocator Library

Intel's `libqkmalloc` library for fast memory allocation provides a C-level interface for memory allocation that is optimized for performance.

You can link the `libqkmalloc` library as a shared library only on Linux platforms for Intel® 64 architecture. This library provides optimized implementation of standard allocation routines `malloc`, `calloc`, `realloc`, and `free`, and is C99 standard compliant.

NOTE This library is limited to work only on Intel® processors and will redirect to standard C routines at runtime if used on non-Intel® processors.

Use Intel's Custom Memory Allocator Library

You can use the `libqkmalloc` library by linking directly to it or by using the `LD_PRELOAD` environment variable.

To ensure that the application overrides the standard library allocation routines with `libqkmalloc`, set the environment variable `LD_PRELOAD` in the command line before the application execution. This environment variable allows you to set a library path that loads before any other library (including the C runtime library). The application uses symbols from the specified library instead of symbols from the standard library.

Restrictions

This library does not support threaded code such as OpenMP* and is not thread-safe. It should not be used simultaneously from multiple threads. For the best results, this library should be used with large throughput workloads.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.
Notice revision #20201201

SIMD Data Layout Templates

SIMD Data Layout Templates (SDLT) is a C++11 template library providing containers that represent arrays of "Plain Old Data" objects (a struct whose data members do not have any pointers/references and no virtual functions) using layouts that enable generation of efficient SIMD (single instruction multiple data) vector code. SDLT uses standard ISO C++11 code. It does not require a special language or compiler to be functional, but takes advantage of performance features (such as OpenMP* SIMD extensions and `pragma ivdep`) that may not be available to all compilers. It is designed to promote scalable SIMD vector programming. To use the library, specify SIMD loops and data layouts using explicit vector programming model and SDLT containers, and let the compiler generate efficient SIMD code in an efficient manner.

Many of the library interfaces employ generic programming, in which interfaces are defined by requirements on types and not specific types. The C++ Standard Template Library (STL) is an example of generic programming. Generic programming enables SDLT to be flexible yet efficient. The generic interfaces enable you to customize components to your specific needs.

The net result is that SDLT enables you to specify a preferred SIMD data layout far more conveniently than re-structuring your code completely with a new data structure for effective vectorization, and at the same time can improve performance.

Motivation

C++ programs often represent an algorithm in terms of high level objects. For many algorithms there is a set of data that the algorithm will need to process. It is common for the data set to be represented as array of "plain old data" objects. It is also common for developers to represent that array with a container from the C ++ Standard Template Library, like std::vector. For example:

```
struct Point3s
{
    float x;
    float y;
    float z;
       // helper methods
};

std::vector<Point3s> inputDataSet(count);
std::vector<Point3s> outputDataSet(count);

for(int i=0; i < count; ++i) {
  Point3s inputElement = inputDataSet[i];
  Point3s result = // transformation of inputElement that is independent of other iterations
                 // can keep algorithm high level using object helper methods
  outputDataSet[i] = result;
}
```

When possible a compiler may attempt to vectorize the loop above, however the overhead of loading the "Array of Structures" data set into vector registers may overcome any performance gain of vectorizing. Programs exhibiting the scenario above could be good candidates to use a SDLT container with a SIMD-friendly internal memory layout. SDLT containers provide *accessor* objects to import and export Primitives between the underlying memory layout and the objects original representation. For example:

```
SDLT_PRIMITIVE(Point3s, x, y, z)

sdlt::soa1d_container<Point3s> inputDataSet(count);
sdlt::soa1d_container<Point3s> outputDataSet(count);

auto inputData = inputDataSet.const_access();
auto outputData = outputDataSet.access();

#pragma forceinline recursive
#pragma omp simd
for(int i=0; i < count; ++i) {
  Point3s inputElement = inputData[i];
  Point3s result = // transformation of inputElement that is independent of other iterations
                  // can keep algorithm high level using object helper methods
  outputData[i] = result;
}
```

When a local variable inside the loop is imported from or exported to using that loop's index, the compiler's vectorizor can now access the underlying SIMD friendly data format and when possible perform unit stride loads. If the compiler can prove nothing outside the loop can access the loop's local object, then it can optimize its private representation of the loop object be "Structure of Arrays" (SOA). In our example, the container's underlying memory layout is also SOA and unit stride loads can be generated. The Container also allocates aligned memory and its accessor objects provide the compiler with the correct alignment information for it to optimize code generation accordingly.

Version Information

This documentation is for SDLT version 2, which extends version 1 by introducing support for n-dimensional containers.

Backwards Compatibility

Public interfaces of version 2 are fully backward compatible with interfaces of version 1.

The backwards compatibility includes:

- Existing source code compatibility.
  - Any source code using the SDLT v1 public API (non-internal interfaces) can be recompiled against SDLT v2 headers with no changes.
- Binary compatibility.
  - Because SDLT v2 API's exist in a new name space, sdlt::v2, all ABI linkage should not collide with any existing SDLT v1 ABI's that exist only in sdlt namespace.
  - A binary, dynamically-linked library that uses SDLT v1 internally, can be linked into a program using SDLT v2, and vice versa.
- Passing SDLT containers or accessors as part of a libraries public API (ABI). When SDLT is used as part of an ABI, that library and the calling code must use the same version of SDLT. They cannot be mixed or matched.

This compatibility doesn't cover internal implementation. Internal implementation for SDLT v1 was updated and unified with parts introduced in v2, so for codes dependent on internal interfaces backwards compatibility is not guaranteed.

Deprecated

These interfaces do not apply for SYCL.

The interfaces below are deprecated; use the replacements provided in the table.

Deprecated Interface	Deprecated in Version	Replaced By
`sdlt::fixed_offset<>`	v2	`sdlt::fixed<>`
`sdlt::aligned_offset<>`	v2	`sdlt::aligned<>`

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Function Calls and Containers

Function Calls

Function calls are a commonly used programming construct. Follow these simple guidelines when using SDLT containers:

- If an SDLT Primitive is passed to a function by value, by pointer, or by reference, be sure to inline them
- Any Non-inlined functions should be SIMD enabled (for example, denote them with `#pragma omp declare simd`).

If a loop variable is passed to a non-inlined function, the current C++ Application Binary Interface (ABI) requires the memory layout match object's original which could cause additional data transformations or inhibit vectorization. For that reason, the SDLT approach works best when all the methods or functions called are inlined or use `#pragma omp declare simd`. Marking a function "inline" explicitly or implicitly is only a hint. Compilers have several limits and heuristics that could cause a function to not be inlined. To avoid this issue, we recommend utilizing the `#pragma forceinline recursive` which instructs the compiler to ignore its limits and heuristics: causing all functions in the following code block that could be inlined to actually be inlined together with any functions called, and functions they call, and so on. Please also note that this can cause the loop body and/or the function body to become too big to optimize. Under such circumstances, carefully examine and restructure the function call boundaries and consider applying non-inlined, SIMD-enabled function calls.

1-Dimensional Containers Overview

What if that `std::vector<typename>` could store data SIMD-friendly format internally while exposing an AOS view to the programmer?

The 1-dimensional containers in SDLT aim to achieve that goal. They can abstract the in-memory data layout of an array of objects to:

1. AOS (Array of Structures)

2. SOA (Structure of Arrays) which is SIMD friendly

Import/Export Only

As the memory layout is abstracted and may not match the original structure's layout, containers cannot provide memory references to the underlying data. Only import or export of the object to and from a particular element in the container. In use, an algorithm might require some minor code changes to follow import/export paradigm, however algorithm itself should read/flow the same.

The 1D containers in SDLT are dynamically resizable with an interface similar to `std::vector<T>`. To avoid accidental misuse of copying containers into C++11 lambda functions we chose to delete the container's copy constructor and instead provide explicit "clone" method instead.

Containers provide SDLT concepts of an accessor and const_accessor for use with SIMD loops, interfaces for `std::vector` compatibility are intended for ease of integration, not high performance.

Just like `std::vector`, the containers own the array data and its scope controls the life of that data.

n-Dimensional Containers Overview

Multi-dimensional containers generalize ideas from 1-dimensional containers; they separate multi-dimensional access semantics from storage logic in an abstract way. A multi-dimensional SDLT container is a generic container that handles an arbitrary number of dimensions, and at the same time internally represents data as needed. Unlike 1-dimensional containers, multi-dimensional containers are not resizable and don't have interfaces like that of `std::vector`. While 1-dimensional containers are like `std::vectors` with decoupled storage, multi-dimensional containers are more akin to arrays (statically sized or variable length).

Below is an example of an n-dimensional container parameterized by three concerns: the data item (primitive) type, the storage layout in memory, and the observed shape of the container.

```
n_container<PrimitiveT, LayoutT, ExtentsT>
```

Template Arguments	Description
`typename PrimitiveT`	The type of primitive that will be contained.
`typename LayoutT`	The type of data layout.
`typename ExtentsT`	Specifies the dimensions of the container

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Construct an n_container

Description

An N-dimensional (multi-dimensional) container must be constructed before it can be used. The data type to be contained must first be declared as a `SDLT_PRIMITIVE`, then a data layout is chosen, and finally the shape of the container is determined describing the extents of each dimension.

Specify Data Layout

Rather than defining different containers for different data layouts, the data layout to use is specified as a template parameter to the container.

Available layouts are summarized in table below. Full details can found on the table in the topic n_container.

Layout	Description
`layout::soa<>`	Structure of Arrays (SOA). Each data member of the Primitive will have its own N-dimensional array.
`layout::soa_per_row<>`	Structure of Arrays Per Row. Each data member of the Primitive will have its own 1-dimensional array per row. Layout repeats for remaining N-1 dimensions.

Layout	Description
`layout::aos_by_struct`	Array of Structures (AOS) Accessed by Struct. Native AOS layout and data access.
`layout::aos_by_stride`	Array of Structures Accessed by Stride. Native SOA data access through pointers to the built in types of members using a stride to account for the size of the Primitive.

Numbers and Constants

In order to define shape, integer values can be provided in three different forms, each successively providing less information to compiler. It is advised to use as precise specification as possible. The compiler may optimize better with more information.

Integer Value Specification	Description
`fixed<int NumberT>`	Known at compile time. `foo(fixed<1080>(), fixed<1920>());` The suffix `_fixed` will declare an equivalent literal. For example, `(1080_fixed` is equivalent to `fixed<1080>`. `foo(1080_fixed, 1920_fixed); )`
`aligned<int AlignmentT>(number)`	Programmer guarantees the number is a multiple of the `AlignmentT`. `foo(aligned<8>(height), aligned<128>(width));`
`“int”`	Arbitrary integer value. `foo(width, height);`

Specify Container Shape

`n_extent_t<…>` is a variadic template that accepts any number of arguments defining dimensions. Because construction using this type may look unclear, a generator object, `n_extent` , is provided to construct extents for all dimensions using a familiar array-definition-like syntax. Extent values may be specified using the most precise representation possible, as described above, to allow the compiler to better prove any potential data alignments.

```
n_extent[height][width];                  // OK
n_extent[height][aligned<128>(width)];  // Better
n_extent[1080_fixed][1920_fixed];       // Best
```

Define an n_container

Using a previously declared primitive (same as SDLT v1),

```
struct RGBAs { float red, green, blue, alpha; };
SDLT_PRIMITIVE(RGBAs, red, green, blue, alpha)
```

A two-dimensional container of RGBAs with HD image size 1920x0180 can be declared and instantiated as in the below example.

```
typedef n_container<RGBAs, layout::soa,
                    n_extent_t<fixed<1080>, fixed<1920>>> HdImage;
HdImage image1;
```

If sizes are not known, a container may be defined with extents unknown to the compiler but known at run-time when an instance of the container is created.

```
typedef n_container<RGBAs, layout::soa, n_extent_t<int, int>> Image;
Image image2(n_extent[height][width]);
```

Additionally, the templated factory function make_n_container<PrimitiveT, LayoutT> may be used to create containers.

```
auto image1 = make_n_container<RGBAs,
                          layout::soa>(n_extent[1080_fixed][1920_fixed]);
auto image2 = make_n_container<RGBAs,
                           layout::soa>(n_extent[height][width]);
```

Access Cells

Containers own data. To get to the data inside, use an "accessor."

```
auto ca = image1.const_access();
auto a = image2.access();
```

Specify the index for each dimension with a series of calls to the array subscript operator [], similar to a multi-dimensional array in C.

```
RGBAs pixel = ca[y][x];
float greyscale = (pixel.red + pixel.green + pixel.blue)/3;
a[y][x] = RGBAs(greyscale, greyscale, greyscale);
```

Discover Extents

Accessors know their extents.

Use template function `extent_d<int DimensionT>(object).`

```
for (int y = 0; y < extent_d<0>(ca); ++y)
    for (int x = 0; x < extent_d<1>(ca); ++x) {
          RGBAs pixel = ca[y][x];
         // …
    }
```

For convenience, non-template methods are also provided.

```
for (int y = 0; y < ca.extent_d0(); ++y)
    for (int x = 0; x < ca.extent_d1(); ++x) {
          RGBAs pixel = ca[y][x];
         // …
    }
```

Lower Dimensions

The result of not specifying all the dimensions required by an accessor is a new accessor with a lower rank that can then be accessed.

```
auto cay = ca[y];
RGBAs pixel = cay[x];
```

Bounds

Description

`bounds_t<LowerT, UpperT>` holds the lower and upper bounds of a half-open interval. It is templated to allow the different integer representations for the lower and upper bounds. The intent is to model a valid iteration space over a single dimension.

Bounds can be used to iterate over an entire extent or to restrict iteration space within an extent

Creating Bounds

Bounds can be created using full bounds_t type, but this may be tedious.

```
bounds_t<int, int>(start, finish)
bounds_t<int, aligned<16>>(start, aligned<16>(finish))
bounds_t<fixed<0>, fixed<1920>>()
```

It is simpler and clearer to use factory function bounds to build a bounds_t<>.

```
bounds(start,finish);
bounds(start, aligned<16>(finish));
bounds(0_fixed, 1920_fixed)
```

Discovering Bounds

Accessors know their valid iteraton space. Initial bounds for an accessor are set to set the lower bound to be fixed<0> and the upper bound set to the value and type of the dimension's extent as specified during construction of the n_container(fixed<>,aligned<>, or int).

To query bounds for given dimension of the accessor use template function `bounds_d<int DimensionT>(object)`.

```
auto b0 = bounds_d<0>(ca);
auto b1 = bounds_d<1>(ca);
for (int y = b0.lower(); y < b0.upper(); ++y)
    for (int x = b1.lower(); x < b1.upper(); ++x) {
          RGBAs pixel = ca[y][x];
         // …
    }
```

`bounds_t` can participate in C++11 range-based for loops.

```
for (auto y: bounds_d<0>(ca))
    for (auto x: bounds_d<1>(ca)) {
          RGBAs pixel = ca[y][x];
         // …
    }
```

```
for (auto y: ca.bounds_d0())
    for (auto x: ca.bounds_d1()) {
          RGBAs pixel = ca[y][x];
         // …
    }
```

N-Dimensional Indexes and Bounds

To model index and bounds values over multiple dimensions, respectively the following template classes are provided: `n_index_t<…>` and `n_bounds_t<…>` . These are both variadic templates, accepting any number of arguments.

`n_index` is a generator to simplify creating instances of `n_index_t`.

```
n_index[540][960]
```

`n_bounds` is a generator to simplify creating instances of `n_bounds_t`.

```
n_bounds[bounds(540,1080)][bounds(960,1920)]
```

Alternatively, `n_bounds_t` can be defined in terms of a `n_index_t` and `n_extent_t`.

```
n_bounds(n_index[540][960], n_extent[540][960]);
```

Accessing Subsections

From a container's accessors, a new accessor can be created over a subsection defined by a `n_bounds_t`.

```
auto ca = c.const_access();
auto subsect = ca.section(n_bounds[bounds(540, 1080)][bounds(960,1920)]);
```

The effect is to restrict the results of `bounds_d<int Dimension>` on the subsection accessor.

You can create a new accessor translated to a different index space.

```
auto offsetnewSpace = ca.translated_to(n_index[1000][2000]);
auto zeroSpace = ca.translated_to_zero();
```

Accesses will have a translation applied that maps the `n_index` back to the lower bounds of the accessor that created it. This allows a smaller container to be reused in a larger index space that is being walked over by blocks, or to move a subsection index space back to the origin.

User-Level Interface

This section describes the user-level interface for the SIMD Data Layout Templates (SDLT). This API is defined in `sdlt.h` and its associated header files.

SDLT Primitives

Primitives represent the data we want to work over in SIMD. They can be more than just data structures. As a C++ object, it can have its own methods that modify its data.

Rules

- Must be Plain Old Data (POD)
  - Has trivial copy constructor
  - Has trivial move constructor
  - Has trivial destructor
  - No virtual functions or virtual bases
- No reference data members
- No unions
- No bit fields
- No bool types
  - Comparison semantics not efficient in SIMD
  - Use 32-bit integer and compare against known values like 0 or 1 explicitly
- Data members need to be public or declare `SDLT_PRIMITIVE_FRIEND` in the object's definition

Current Limitations

- No pointer data members
- No C++11 strongly typed enums—use integers instead.
- No array based data members.
- copy constructor and assignment operator (=) defined by individual member assignment—strongly encouraged to facilitate better code generation

They may seem like large restrictions, but often code can easily be re-factored to meet this requirement. For example:

```
class Point3d {
    // methods...
protected:
    double v[3];
};
```

can be re-factored to have a public data member for each element in the array and update methods to use the `x`, `y`, and `z` data members rather than the array `v`.

```
class Point3d {
public:
    // methods...
    double x;
    double y;
    double z;
};
```

For better code generation, explicitly define a copy constructor and assignment operator (=) by individual member assignment.

SDLT_PRIMITIVE Macro

Once an object meets the criteria above, we can consider it a Primitive type in SDLT. In order for Container's to import and export the Primitive, it has to understand its data layout. Unfortunately C++11 lacks compile time reflection, so the user must provide SDLT with a description of your structure's data layout. This is easily done with the SDLT_PRIMITIVE helper macro that accepts a struct type followed by a comma separated list of its data members.

```
SDLT_PRIMITIVE(STRUCT_NAME, DATA_MEMBER_1, ...)
```

Example Usage:

```
struct UserObject
{
    float x;
    float y;
    double acceleration;
    int behavior;
};

SDLT_PRIMITIVE(UserObject, x, y, acceleration, behavior)
```

An object must be declared as a Primitive before it can be used in a Container. However, built-in types like float, double, int, etc. do not need to be declared as a Primitive before use with a Container. Built-in's are automatically considered Primitives by SDLT.

Nested Primitives are supported, but the nested Primitive must be declared before the outer Primitive is. Example: Axis Aligned Bounding Box made up of two 3d points

```
struct Point3s
{
    float x;
    float y;
    float z;
};

struct AABB
{
    Point3s topLeft;
    Point3s bottomRight;
};

SDLT_PRIMITIVE(Point3s, x, y, z)
SDLT_PRIMITIVE(AABB, topLeft, bottomRight)
```

Notice the `struct` definitions themselves do not derive from SDLT or use any of its nomenclature. This independence allows classes to be used in code not using SDLT and only code that does use SDLT Containers needs to see the Primitive declarations.

soa1d_container

Template class for "Structure of Arrays" memory layout of a one-dimensional container of Primitives.
#include <sdlt/soa1d_container.h>

Syntax

```
template<typename PrimitiveT,
         int AlignD1OnIndexT = 0,
         class AllocatorT = allocator::default_alloc>
class soa1d_container;
```

Arguments

`typename PrimitiveT`	The type that each element in the array will store
`int AlignD1OnIndexT = 0`	[Optional] The index on which the data access will be aligned (useful for stencils)
`class AllocatorT = allocator::default_alloc`	[Optional] Specify type of allocator to be used. `allocator::default_alloc` is currently the only allocator supported.

Description

Dynamically sized container of Primitive elements with memory layout as a Structure of Arrays internally providing:

- Dynamic resizing with interface similar to `std::vector`
- Accessor objects suitable for efficient data access inside SIMD loops

Member	Description
`typedef size_t size_type;`	Type to use when specifying sizes to methods of the container.
`template <typename OffsetT = no_offset> using accessor;`	Template alias to an accessor for this container
`template <typename OffsetT = no_offset > using const_accessor;`	Template alias to an const_accessor for this container

Member Type	Description
`soa1d_container( size_type size_d1 = 0u, buffer_offset_in_cachelines buffer_offset = buffer_offset_in_cachelines(0), const allocator_type & an_allocator = allocator_type());`	Constructs an uninitialized container of size_d1 elements, using optionally specified allocator instance, using optionally specified number of cache lines to offset the start of the buffer in memory to allow management of 4k cache aliasing.

Member Type	Description
soa1d_container(size_type size_d1, const PrimitiveT &a_value, buffer_offset_in_cachelines buffer_offset = buffer_offset_in_cachelines(0), const allocator_type & an_allocator = allocator_type());	Constructs a container of size_d1 elements initializing each with a_value, using optionally specified allocator instance, using optionally specified number of cache lines to offset the start of the buffer in memory to allow management of 4k cache aliasing.
template<typename StlAllocatorT> soa1d_container(const std::vector<PrimitiveT, StlAllocatorT> &other, buffer_offset_in_cachelines buffer_offset = buffer_offset_in_cachelines(0), const allocator_type & an_allocator = allocator_type());	Constructs a container with a copy of each of the elements in other, in the same order, using optionally specified allocator instance, using optionally specified number of cache lines to offset the start of the buffer in memory to allow management of 4k cache aliasing.
soa1d_container(const PrimitiveT *other_array, size_type number_of_elements, buffer_offset_in_cachelines buffer_offset = buffer_offset_in_cachelines(0), const allocator_type & an_allocator = allocator_type());	Constructs a container with a copy of number_of_elements elements from the array other_array, in the same order, using optionally specified allocator instance, using optionally specified number of cache lines to offset the start of the buffer in memory to allow management of 4k cache aliasing.
template< typename IteratorT > soa1d_container(IteratorT a_begin, IteratorT an_end, buffer_offset_in_cachelines buffer_offset = buffer_offset_in_cachelines(0), const allocator_type & an_allocator = allocator_type());	Constructs a container with as many elements as the range [a_begin - an_end), each with a copy of the value from its corresponding element in that range, in the same order, using optionally specified allocator instance, using optionally specified number of cache lines to offset the start of the buffer in memory to allow management of 4k cache aliasing.
soa1d_container clone() const;	Returns: a new soa1d_container instance with its own copy of the elements
void resize(size_type new_size_d1);	Resize the container so that it contains *new_size_d1* elements. If the new size is greater than the current container size, the new elements are unitialized.
accessor<> access();	Returns: accessor with no embedded index offset.
accessor<int> access(int offset);	Returns: accessor with an integer based embedded index offset.
template<int IndexAlignmentT> accessor<aligned_offset<IndexAlignmentT> > access(aligned_offset<IndexAlignmentT>);	Returns: accessor with an aligned_offset<IndexAlignmentT> based embedded index offset.
template<int OffsetT> accessor<fixed_offset<OffsetT> > access(fixed_offset<OffsetT>);	Returns: accessor with a fixed_offset<OffsetT> based embedded index offset.

Member Type	Description
`const_accessor<> const_access() const;`	Returns: const_accessor with no embedded index offset.
`const_accessor<int> const_access(int offset) const;`	Returns: const_accessor with an integer based embedded index offset.
`const_accessor<aligned_offset<IndexAlignmentT> > const_access(aligned_offset<IndexAlignmentT> offset) const;`	Returns: const_accessor with an aligned_offset<IndexAlignmentT> based embedded index offset.
`template<int OffsetT> const_accessor<fixed_offset<OffsetT> > const_access(fixed_offset<OffsetT>) const;`	Returns: const_accessor with a fixed_offset<OffsetT> based embedded index offset.

STL Compatibility

In addition to the performance oriented interface explained in the table above, `soa1d_container` implements a subset of the `std::vector` interface that is intended for ease of integration, not high performance. Due to the import/export only requirement we can’t return a reference to the object, instead `iterators` and `operator[]` return a Proxy object while other "const' methods return a "value_type const". Futhermore, iterators do not support the `->` operator. Despite that limitation the iterators can be passed to any STL algorithm. Also for performance reasons, resize does not initialize new elements. The following `std::vector` interface methods are implemented:

- size, max_size, capacity, empty, reserve, shrink_to_fit
- assign, push_back, pop_back, clear, insert, emplace, erase
- cbegin, cend, begin, end, begin, end, crbegin, crend, rbegin, rend, rbegin, rend
- operator[], front() const, back() const, at() const
- swap, ==, !=
- swap, soa1d_container(soa1d_container&& donor), soa1d_container & operator=(soa1d_container&& donor)

aos1d_container

Template class for "Array of Structures" memory layout of a one-dimensional container of Primitives.
`#include <sdlt/aos1d_container.h>`

Syntax

```
template<
    typename PrimitiveT,
    AccessBy AccessByT,
    class AllocatorT = allocator::default_alloc
>
class aos1d_container;
```

Arguments

`typename PrimitiveT`	The type that each element in the array will store
`access_by AccessByT`	Enum to control how the memory layout will be accessed. Recommend access_by_struct unless you are having issues vectorizing. See the documentation of `access_by` for more details

`class AllocatorT = allocator::default_alloc`	[Optional] Specify the type of allocator to be used. `allocator::default_alloc` is currently the only allocator supported.

Description

Provide compatible interface with `soa1d_container` while keeping the memory layout as an Array of Structures internally. User can easily switch between data layouts by changing the type of container they use. The rest of the code written against accessors and proxy elements and members can stay the same.

- Dynamic resizing with interface similar to `std::vector`
- Accessor objects suitable for efficient data access inside SIMD loops

Member	Description
`typedef size_t size_type;`	Type to use when specifying sizes to methods of the container.
`template <typename OffsetT = no_offset> using accessor;`	Template alias to an `accessor` for this container
`template <typename OffsetT = no_offset> using const_accessor;`	Template alias to a `const_accessor` for this container

Member Type	Description
`aos1d_container( size_type size_d1 = 0u, buffer_offset_in_cachelines buffer_offset = buffer_offset_in_cachelines(0), const allocator_type & an_allocator = allocator_type());`	Constructs an uninitialized container of size_d1 elements, using optionally specified allocator instance, using optionally specified number of cache lines to offset the start of the buffer in memory to allow management of 4k cache aliasing.
`aos1d_container ( size_type size_d1, const PrimitiveT &a_value, buffer_offset_in_cachelines buffer_offset = buffer_offset_in_cachelines(0), const allocator_type & an_allocator = allocator_type());`	Constructs a container of *size_d1* elements initializing each with *a_value*, using optionally specified allocator instance, using optionally specified number of cache lines to offset the start of the buffer in memory to allow management of 4k cache aliasing.
`template<typename StlAllocatorT> aos1d_container( const std::vector<PrimitiveT, StlAllocatorT> &other, buffer_offset_in_cachelines buffer_offset = buffer_offset_in_cachelines(0), const allocator_type & an_allocator = allocator_type());`	Constructs a container with a copy of each of the elements in *other*, in the same order, using optionally specified allocator instance, using optionally specified number of cache lines to offset the start of the buffer in memory to allow management of 4k cache aliasing.
`aos1d_container( const PrimitiveT *other_array, size_type number_of_elements, buffer_offset_in_cachelines buffer_offset`	Constructs a container with a copy of *number_of_elements* elements from the array *other_array*, in the same order, using optionally specified allocator instance, using optionally

Member Type	Description
`        = buffer_offset_in_cachelines(0),` `    const allocator_type & an_allocator` `        = allocator_type());`	specified number of cache lines to offset the start of the buffer in memory to allow management of 4k cache aliasing.
`template< typename IteratorT >` `aos1d_container(` `    IteratorT a_begin,` `    IteratorT an_end,` `    buffer_offset_in_cachelines buffer_offset` `        = buffer_offset_in_cachelines(0),` `    const allocator_type & an_allocator` `        = allocator_type());`	Constructs a container with as many elements as the range [*a_begin*-*an_end*), each with a copy of the value from its corresponding element in that range, in the same order, using optionally specified allocator instance, using optionally specified number of cache lines to offset the start of the buffer in memory to allow management of 4k cache aliasing.
`aos1d_container clone() const;`	Returns: a new *aos1d_container* instance with its own copy of the elements
`void resize(size_type new_size_d1);`	Resize the container so that it contains *new_size_d1* elements. If the new size is greater than the current container size, the new elements are unitialized
`accessor<> access();`	Returns: *accessor* with no embedded index *offset*.
`accessor<int> access(int offset);`	Returns:*accessor* with an integer based embedded index *offset*.
`template<int IndexAlignmentT>` `accessor<aligned_offset<IndexAlignmentT> >` `    access(aligned_offset<IndexAlignmentT>);`	Returns: *accessor* with an aligned_offset<IndexAlignmentT> based embedded index *offset.*
`template<int OffsetT>` `accessor<fixed_offset<OffsetT> >` `    access(fixed_offset<OffsetT>);`	Returns: *accessor* with a fixed_offset<OffsetT> based embedded index *offset*.
`const_accessor<> const_access() const;`	Returns: *const_accessor* with no embedded index offset.
`const_accessor<int>` `    const_access(int offset) const;`	Returns: *const_accessor* with an integer based embedded index *offset.*
`const_accessor<aligned_offset<IndexAlignmentT> >` `const_access(aligned_offset<IndexAlignmentT> offset) const;`	Returns:*const_accessor* with an aligned_offset<IndexAlignmentT> based embedded index *offset*.
`template<int OffsetT>` `const_accessor<fixed_offset<OffsetT> >` `    const_access(fixed_offset<OffsetT>) const;`	Returns:*const_accessor* with a fixed_offset<OffsetT> based embedded index *offset*.

STL Compatibility

In addition to the performance oriented interface explained in the table above, `aos1d_container` implements a subset of the `std::vector` interface that is intended for ease of integration, not high performance. Due to the import/export only requirement we can't return a reference to the object, instead `iterators` and `operator[]` return a Proxy object while other "const' methods return a "value_type const". Furthermore, iterators do not support the `->` operator. Despite that limitation the iterators can be passed to any STL algorithm. Also for performance reasons, resize does not initialize new elements. The following `std::vector` interface methods are implemented:

- size, max_size, capacity, empty, reserve, shrink_to_fit
- assign, push_back, pop_back, clear, insert, emplace, erase
- cbegin, cend, begin, end, crbegin, crend, rbegin, rend, rbegin, rend
- operator[], front() const, back() const, at() const
- swap, ==, !=
- swap, aos1d_container(aos1d_container&& donor), aos1d_container & operator=(aos1d_container&& donor)

access_by

Enum to control how the memory layout will be accessed. `#include <sdlt/access_by.h>`

Syntax

```
enum access_by
{
    access_by_struct,
    access_by_stride
};
```

Description

The `access_by_struct` causes data access via structure member access. Nested structures will drill down through the structure members in a nested manner. For example an Axis Aligned Bounding Box (AABB) containing two Point3d objects (with x,y,z data members) will logically expand to something like:

```
AABB local;
local = accessor.mData[i];
```

`access_by_stride` will cause data access through pointers to built in types with a stride to account for the size of the primitive. For an Axis Aligned Bounding Box (AABB) containing two Point3d objects (with x,y,z data members) will logically expand to something like:

```
AABB local;
local.topLeft.x = *(accessor.mData + offsetof(AABB,topLeft) + offset(Point3d,x) +
(sizeof(AABB)*i));
local.topLeft.y = *(accessor.mData + offsetof(AABB,topLeft) + offset(Point3d,y) +
(sizeof(AABB)*i));
local.topLeft.z = *(accessor.mData + offsetof(AABB,topLeft) + offset(Point3d,z) +
(sizeof(AABB)*i));
local.topRight.x = *(accessor.mData + offsetof(AABB,topRight) + offset(Point3d,x) +
(sizeof(AABB)*i));
local.topRight.y = *(accessor.mData + offsetof(AABB,topRight) + offset(Point3d,y) +
(sizeof(AABB)*i));
local.topRight.z = *(accessor.mData + offsetof(AABB,topRight) + offset(Point3d,z) +
(sizeof(AABB)*i));
```

When vectorizing, `access_by_struct` can sometimes generate better code as the compiler could perform wide loads and use shuffle/insert instructions to move data into SIMD registers. However, depending on the complexity of the primitive, it can also fail to vectorize, especially when the primitive contains nested structures.

On the other hand `access_by_stride` has always vectorized successfully, because the data access is simplified to an array pointer with a stride. The compiler is able to handle any complexity of primitive, because it never sees the complexity and instead just sees the simple array pointer with strided access.

`access_by_struct` is probably the best choice as it offers a chance of better code generation especially when used outside of a SIMD loop. However if you run into issues when vectorizing, try `access_by_stride` to see if that alleviates the problem.

We leave this choice up to the developer and require they explicitly make a choice, so this is not hidden behavior.

n_container

Template class for N-dimensional container. The contained primitive type, exact memory layout and container shape are defined via template arguments.

Syntax

```
template <typename PrimitiveT,
          typename LayoutT,
          typename ExtentsT,
          typename AllocatorT >
class n_container;
```

Description

N-dimensional container of PrimitiveT elements with predefined memory layout and shape. Provides accessor interface suitable for flexible and efficient data access inside SIMD loops

The following table provides information on the template arguments for `n_container`

Template Argument	Description
`typename PrimitiveT`	The type that each cell in the multi-dimensional container will store. Requirements: PrimitiveT must be previously declared with the SDLT_PRIMITIVE macro.
`typename LayoutT`	The in-memory data layout of cells in the container. Requirements: LayoutT must be a class from *layout* namespace.
`typename ExtentsT`	The shape of the container. Requirements: ExtentsT must be a concrete type of *n_extent_t* variadic template.
`class AllocatorT = allocator::default_alloc`	[Optional] Specify type of *allocator* to be used. allocator::default_alloc is currently the only allocator supported.

The following table provides information on the types defined as members of `n_container`

Member Type	Description
`typedef PrimitiveT primitive_type;`	Type inside each cell of the container.
`typedef PrimitiveT allocator_type;`	Type of allocator used by the container.
`typedef implementation-defined accessor`	Type of an *accessor* that can write or read cells to and from this container.

Member Type	Description
`typedef implementation-defined const_accessor;`	Type of a *const_accessor* that can read cells from this container.

The following table provides information on the methods of `n_container`

Member	Description
`n_container ( const ExtentsT &a_extents, buffer_offset_in_cachelines buffer_offset =buffer_offset_in_cachelines(0), const AllocatorT &an_allocator=AllocatorT())`	Constructs an uninitialized container of the shape defined as *a_extents*, using optionally specified number of cache lines to offset the start of the buffer in memory to allow management of 4k cache aliasing, using optionally specified allocator instance.
`n_container (buffer_offset_in_cachelines buffer_offset = buffer_offset_in_cachelines(0), const AllocatorT &an_allocator=AllocatorT())`	Constructs an uninitialized container of the shape, defined via template parameter ExtentsT using optionally specified number of cache lines to offset the start of the buffer in memory to allow management of 4k cache aliasing, using optionally specified allocator instance. ExtentsT must be default constructible. Only true when ExtentsT is made up enitrely of fixed<NumberT> types.
`n_container(n_container&& donor)`	Transfers ownership of the donor's currently owned buffers and organization, if any. Any outstanding *accessors* on the donor are no longer valid.
`n_container & operator = (n_container&& donor)`	Frees any existing buffers, then transfers ownership of the donor's currently owned buffers and organization, if any. Any outstanding *accessors* on the donor are no longer valid. **Returns:** Reference to this instance.
`const ExtentsT& n_extent () const`	Provides the shape of the container. Alternatively, the free template function *extent_d<int DimenstionT>(const n_container &)* could be used. **Returns:** Constant reference to ExtentsT instance describing the shape of the container.
`const_accessor const_access();`	Constructs an *const_accessor* with knowledge of the underlying data organization to read cells inside the container. **Returns:***const_accessor* for the container
`accessor access();`	Constructs an *accessor* with knowledge of the underlying data organization to write or read cells inside the container. **Returns:***accessor* for the container

The following table provides information about the friend functions of n_container.

Friend Function	Description
`std::ostream& operator << (std::ostream& output_stream, const n_container & a_container)`	Append string representation of a_container's extents values to a_output_stream. **Returns:** Reference to a_output_stream for chained calls.

Layouts

sdlt::layout namespace

Rather than having different container types for different data layouts, the library uses the types from the layout namespace as a template parameter to the n_container.

Available layouts are defined in the namespace layout and summarized in table below.

Layout	Description
`template <typename AlignOnColumnIndexT=0> layout::soa`	Structure of Arrays: Each data member of the Primitive will have its own N-dimensional array. The arrays are placed back-to-back inside a contiguous buffer. Template parameter AlignOnColumnIndexT identifies which column of the row dimension should be cache line aligned. The AlignOnColumnIndexT of each row is cache line aligned.
`template <typename AlignOnColumnIndexT> layout::soa_per_row`	Structure of Arrays Per Row: Each data member of the Primitive will have its own 1-dimensional array for the row dimension (Soa1d) placed back to back. The AlignOnColumnIndexT of each row is cache line aligned. Multiple of these Soa1d's are laid out sequentially to model the remaining dimensions, effectively becoming an Array of Structures of Arrays where the SOA where the size of the array is the row's extent. This can be particularly efficient when the extent of the row can be fixed<NumberT>. **Note**: If the size of the row isn't known at compile time, consider adding an additional dimension that is fixed<Number> and dividing the row up by that fixed<NumberT>.
`layout::aos_by_struct`	Array of Structures Accessed by Struct: Primitives are laid out in native format back to back in memory and access happens via structure or member access. Nested structures will drill down through the structure members in a nested manner.
`layout::aos_by_stride`	Array of Structures Accessed by Stride: Primitives are laid out in native format back to back in memory and accessed through pointers to built in types with a stride to account for the size of the Primitive. Can be useful if aos_by_struct doesn't vectorize.

Description

The classes are empty and only for specialization of containers for denoted layouts.

Shape

Variadic template class n_extent_t describes the shape of the n_dimensional container. Specifically, the number of dimensions the size of each.

Syntax

```
template<typename... TypeListT>
class n_extent_t
```

Description

`n_extent_t` represents the shape of a container as a sequence of sizes for each dimension. The size of each dimension can be represented by different types. This flexibility allows the same interface to be used to declare `n_extents_t` whose dimensions are fully known at compile time with `fixed<int NumberT>`, or to be only known at runtime with `int`, or only known at runtime but with a guarantee will be a multiple of an alignment with `aligned<int Alignment>`. For details, see the Number representation section.

The following table provides information on the template arguments for `n_extent_t`.

Template Argument	Description
`typename... TypeListT`	Comma separated list of types, where the number of types provided controls how many dimensions there are. Each type in the list identifies how the size of the corresponding dimension is to be represented. The order of the dimensions is the same order as C++ subscripts declaring a multi-dimensional array, from leftmost to rightmost. Type must be `int`, `fixed<NumberT>`, or `aligned<AlignmentT>` for each value describing corresponding dimensions size (extent) in regular order of C++ subscripts - from outer to inner.

The following table provides information on the members of `n_extent_t`

Member	Description
`static constexpr int rank;`	Number of dimensions.
`static constexpr int row_dimension = rank-1;`	Index of last dimension, *row.*
`n_extent_t()`	**Requirements:** Every type in TypeListT is default constructible. **Effects:** Construct n_extent_t, uses default values of each type in TypeListT for the dimesnion sizes. In general, only correctly initialized when every type is a fixed<NumberT>
`n_extent_t(const n_extent_t &a_other)`	**Effects:** Construct n_exent_t, copying size of each dimension from *a_other*.

Member	Description
`explicit n_extent_t(const TypeListT & … a_values)`	**Effects:** Construct n_exent_t, initializing each dimension with the corresponding value from the list of *a_values* passed as an argument. In use, *a_values* is a comma separate list of values whose length and types are defined by TypeListT.
`template<int DimensionT> auto get() const`	**Requirements:** DimenstionT >=0 and DimensiontT < rank. **Effects:** Determine the exent of *DimensionT*. **Returns:** In the type declared by the *DimensionT* position of 0-based TypeListT, the extent of the specified *DimensionT*
`template<int DimensionT> auto rightmost_dimensions() const`	**Requirements:** DimenstionT >=0 and DimensiontT <= rank. **Effects:** Construct a n_extent_t with a lower rank by copying the righmost DimensionT values from this instance. **Returns:** n_exent[get<rank - DimensionT>()] [get<rank + 1 - DimensionT>()] [get<…>()] [get<row_dimension>()]
`template<class... OtherTypeListT> bool operator == (const n_extent_t<OtherTypeListT...> a_other) const`	**Requirements:** rank of a_other is the same as this instance's. **Effects:** Compare size of each dimension for equality. Only compares numeric values, not the types of each dimension. **Returns:***true* if all dimensions are numerically equal, *false* otherwise.
`template<class... OtherTypeListT> bool operator != (const n_extent_t<OtherTypeListT...> a_other) const`	**Requirements:** rank of a_other is the same as this instance's. **Effects:** Compare size of each dimension for inequality. Only compares numeric values, not the types of each dimension. **Returns:***true* if any dimensions are numerically different, *false* otherwise.
`size_t size() const`	**Returns:** Number of elements specified by extent **Effects:** Calculates the number of cells represented by the current extent values of each dimension by multiplying them all together. **Returns:** get<0>()*get<1>()*get<…>()*get<rank-1>()

The following table provides information on the friend functions of `n_extent_t`.

Friend function	Description
`std::ostream& operator << (std::ostream& output_stream, const n_extent_t & a_extents)`	**Effects:** Append string representation of a_extents' values to a_output_stream **Returns:** Reference to a_output_stream for chained calls.

n_extent_generator
To facilitate simpler and clearer creation of `n_extent_t` *objects.*

Syntax

```
template<typename... TypeListT>
class n_extent_generator;

namespace {
    // Instance of generator object
    n_extent_generator<> n_extent;
}
```

Description

The generator object provides recursively constructing operators [] for `fixed<>`, `aligned<>`, and `integer` values allowing building of an `n_extent_t` <...> instance, one dimension at a time. The main purpose is to allow a usage syntax that is similar to C multi-dimensional array definition:

Compare the following examples, instantiating three `n_extent_t` instances. and using the generator object to instantiate equivalent instances.

```
n_extent_t<int, int> ext1(height, width);
n_extent_t<int, aligned<128>> ext2(height, width);
n_extent_t<fixed<1080>, fixed<1920>> ext3(1080_fixed, 1920_fixed);
```

```
auto ext1 = n_extent[height][width];
auto ext2 = n_extent[height][aligned<128>(width)];
auto ext3 = n_extent[1080_fixed][1920_fixed];
```

Class Hierarchy

It is expected that `n_extent_generator` < … > not be directly used as a data member or parameter, instead only `n_extent_t` <...> from which it is derived. The generator object `n_extent` can be automatically downcast any place expecting an `n_extent_t<…>` .

The following table provides the template arguments for `n_extent_generator`

Template Argument	Description
`typename... TypeListT`	Comma separated list of types, where the number of types provided controls how many dimensions the generator currently represent. Each type in the list identifies how the size of the corresponding dimension is to be represented. The order of the dimensions is the same order as C++ subscripts declaring a multi-dimensional array – from leftmost to rightmost. **Requirements**: Type is `int`, `fixed<NumberT>`, or `aligned<AlignmentT>`.

The following table provides information on the types defined as members of `n_extent_generator` in addition to those inherited from `n_extent_t`.

Member Type	Description
`typedef n_extent_t<TypeListT...> value_type`	Type value that the any chained [] operator calls have produced.

The following table provides information on the members of `n_extent_generator` in addition to those inherited from `n_extent_t`

Member	Description
`n_extent_generator ()`	**Requirements:** TypeListT is empty **Effects:** Construct generator with no extents specified
`n_extent_generator (const n_extent_generator &a_other)`	**Effects:** Construct generator copying any extent values from a_other
`n_extent_generator<TypeListT..., int> operator [] (int a_size) const`	**Requirements:** a_size >= 0 **Returns:** n_extent_generator<...> with additional rightmost integer based extent.
`n_extent_generator<TypeListT..., fixed<NumberT>> operator [] (fixed<NumberT> a_size) const`	**Requirements:** a_size >= 0 **Returns:** n_extent_generator<...> with additional rightmost fixed<NumberT> extent.
`n_extent_generator<TypeListT..., aligned<AlignmentT>> operator [] (aligned<AlignmentT> a_size)`	**Requirements:** a_size >= 0 **Returns:** n_extent_generator<...> with additional rightmost aligned<AlignmentT> based extent.
`value_type value() const`	**Returns:** n_extent_t<...> with the correct types and values of the multi-dimensional extents aggregated by the generator.

make_ n_container template function

Factory function to construct an instance of a properly-typed n_container<...> based on n_extent_t passed to it.

Syntax

```
template<
    typename PrimitiveT,
    typename LayoutT,
    typename AllocatorT = allocator::default_alloc,
    typename ExtentsT
>
auto make_n_container(const ExtentsT &_extents)
->n_container<PrimitiveT, LayoutT, ExtentsT, AllocatorT>
```

Description

Use `make_n_container` to more easily create an n-dimensional container using template argument deduction, and avoid specifying the type of extents.

An example of the instantiation of a High Definition image object is below.

```
typedef n_container<RGBAs, layout::soa,
                    n_extent_t<int, int>> HdImage;
HdImage image1(n_extent[1080][1920]);
```

Alternatively, it is possible to use factory function with the C++11 keyword `auto`, as shown below.

```
auto image1 = make_n_container<RGBAs,
                              layout::soa>(n_extent[1080][1920]);
```

extent_d template function

Syntax

```
template<int DimensionT, typename ObjT>
auto extent_d(const ObjT &a_obj)
```

Description

The template function offers a consistent way to determine the extent of a dimension for a multi-dimensional object. It can avoid extracting an entire n_extent_t<…> when only the extent of a single dimension is needed.

Template Argument	Description
`int DimensionT`	0 based index starting at the leftmost dimension indicating which n-dimensions to query the extent of. **Requirements:** DimensionT >=0 and DimensionT < ObjT::rank
`typename ObjT`	The type of n-dimensional object from which to retrieve the extent. **Requirements:** ObtT is one of: n_container<…> n_extent_t<…> n_extent_generator<…>

Returns

The correctly typed extent corresponding to the requested DimensionT of a_obj.

Example

```
template <typename VolumeT>
void foo(const VolumeT & a_volume)
{
    int extent_z = extent_d<0>(volume);
    int extent_y = extent_d<1>(volume);
    int extent_x = extent_d<2>(volume);
    /…
}
```

Bounds

This section provides information related to bounds for the SIMD Data Layout Templates (SDLT).

bounds_t

Class represents a half-open interval with lower and upper bounds. `#include <sdlt/bounds.h>`

Syntax

```
template<typename LowerT = int, typename UpperT = int>
struct bounds_t
```

Description

bounds_t holds the lower and upper bounds of a half open interval. It is templated to allow the different representations for the lower and upper bounds. Supported types include fixed<NumberT>, aligned<AlignmentT> and integer values. bounds_t models a valid iteration space over a single dimension.

bounds_t can be used to represent an iteration space over the entire extent of a dimension or to restrict iteration space within the extent. n_bounds_t aggregates a number of bounds_t objects to allow construction of multi-demensional subsections restricting multiple extents.

The class interface is compatible with C++ range-based loops to simplify iteration.

Template Argument	Description
`typename LowerT = int`	Type of lower bound. Requirements: type is int, or fixed<NumberT>, or aligned<AlignmentT>
`typename UpperT = int`	Type of upper bound. Requirements: type is int, or fixed<NumberT>, or aligned<AlignmentT>

Member Types	Description
`typedef LowerT lower_type`	Type of the lower bound
`typedef UpperT upper_type`	Type of the upper bound
`typedef implementation-defined iterator`	Iterator type for C++ range-based loops support.

Member	Description
`bounds_t()`	Effects: Constructs bounds_t with uninitialized lower and upper bounds.
`bounds_t(lower_type l, upper_type u)`	Requirements: (u >= l) Effects: Constructs bounds_t representing the half-open interval [l, u)
`bounds_t(const bounds_t & a_other)`	Effects: Constructs bounds_t with lower and upper bounds initialized from those of a_other.
`template<typename OtherLowerT, typename OtherUpperT> bounds_t(const bounds_t<OtherLowerT, OtherUpperT> & a_other)`	Requirements: OtherLowerT and OtherUpperT can legally be converted to lower_type and upper_type. For example it would be illegal to convert an int to fixed<8>(). Effects: Constructs bounds_t with lower and upper bounds initialized from those of a_other.
`void set(lower_type l, upper_type u)`	Effects: Set index of the inclusive lower bound and the index of the exclusive upper bound.
`void set_lower(lower_type a_lower)`	Effects: Set index of the inclusive lower bound
`void set_upper(upper_type a_upper)`	Effects: Set index of the exclusive upper bound

Member	Description
lower_type lower() const	Returns: index of the inclusive lower bound
upper_type upper() const	Returns: index of the exclusive upper bound
iterator begin() const	Returns: index iterator for the inclusive lower bound. NOTE: C++11 range-based loops require begin() & end()
iterator end() const	Returns: index iterator for the exclusive upper bound. NOTE: C++11 range-based loops require begin() & end()
auto width() const	Effects: Determine width of iteration space inside the half open interval between lower() and upper() bounds. Returns: upper() – lower() NOTE: the return type depends on resulting type of a subtraction between the types of upper() and lower().
template<typename OtherLowerT, typename OtherUpperT> bool contains(const bounds_t<OtherLowerT, OtherUpperT> &a_other) const	Effects: Determine if interval of a_other is entirely contained inside this object's bounds Returns: (a_other.lower() >= lower() && a_other.upper() <= upper())
template<typename T> auto operator + (const T &offset) const	Effects: create a new bounds_t instance with offset added to both lower and upper bounds. Returns: bounds(lower() + offset, upper()+offset) NOTE: The lower_type and upper_type of the returned bound_t maybe different as result of addition of the offset.
template<typename T> auto operator - (const T & offset) const	Effects: create a new bounds_t instance with offset subtracted from both lower and upper bounds. Returns: bounds(lower() - offset, upper()-offset) NOTE: The lower_type and upper_type of the returned object maybe different as result of subtraction of T.
bool operator == (const bounds_t &a_other) const	Effects: Equality comparison with same-typed bounds_t object Returns: (lower() == a_other.lower() && upper() == a_other.upper())
template<typename OtherLowerT, typename OtherUpperT> bool operator == (const bounds_t<OtherLowerT, OtherUpperT> &a_other) const	Effects: Equality comparison with bounds_t object of different lower_type or upper_type. Returns: (lower() == a_other.lower() && upper() == a_other.upper())

Member	Description
`bool operator != (const bounds_t &) const`	Effects: Inequality comparison with same-typed bounds_t object Returns: (lower() != a_other.lower() || upper() != a_other.upper())
`template<typename OtherLowerT, typename OtherUpperT> bool operator != ( const bounds_t<OtherLowerT, OtherUpperT> &a_other) const`	Effects: Inequality comparison with with bounds_t object of different lower_type or upper_type Returns: (lower() != a_other.lower() || upper() != a_other.upper())

Friend Function	Description
`std::ostream& operator << (std::ostream& a_output_stream, const bounds_t &a_bounds)`	Effects: append string representation of bounds_t lower and upper values to a_output_stream Returns: reference to a_output_stream for chained calls

Range-based loops support

The bounds_t provides begin() and end() methods returning iterators to enable C++11 range-based loops. The may save quite some typing and improve code clarity when iterating over bounds of a multidimensional container.

Compare:

```
auto ca = image_container.const_access();
auto b0 = bounds_d<0>(ca);
auto b1 = bounds_d<1>(ca);
for (auto y = b0.lower(); y < b0.upper(); ++y)
    for (auto x = b1.lower(); x < b1.upper(); ++x) {
          RGBAs pixel = ca[y][x];
            // …
    }
```

and

```
auto ca = image_container.const_access();
for (auto y: bounds_d<0>(ca))
    for (auto x: bounds_d<1>(ca)) {
          RGBAs pixel = ca[y][x];
            // …
    }
```

Note that iterator only gives an index value within the bounds, not an object value. It is expected to be used to index into accessors like in example above.

sdlt::bounds Template Function

Factory function provided for creation of bounds_t objects. `#include <sdlt/bounds.h>`

Syntax

```
template<typename LowerT, typename UpperT>
auto bounds(LowerT a_lower, UpperT a_upper)
```

Description

In order to make creation of objects of bounds_t cleaner the factory function bounds is provided. It basically enables LowerT and UpperT to be deduced from the arguments passed into it.

Template Argument	Description
`typename LowerT = int`	Type of lower bound. Requirements: type is int, or fixed<NumberT>, or aligned<AlignmentT>
`typename UpperT = int`	Type of upper bound. Requirements: type is int, or fixed<NumberT>, or aligned<AlignmentT>

Returns:

The correctly typed bounds_t<LowerT, UpperT> corresponding to types of a_lower and a_upper passed to the factory function.

Example:

Compare two ways of instantiating a bounds:

```
bounds_t<fixed<0>, aligned<16>> my_bounds1(0_fixed, aligned<16>(upper))
auto my_bounds2 = bounds_t<fixed<0>, aligned<16>>(0_fixed, aligned<16>(upper))
```

With the factory function:

```
auto my_bounds = bounds(0_fixed, aligned<16>(upper))
```

n_bounds_t

Variadic template class to describe the valid iteration space over an N-dimensional container. `#include <sdlt/n_bounds.h>`

Syntax

```
template<typename... TypeListT>
class n_bounds_t
```

Description

n_bound_t represents the valid iteration space over a n_container or its accessor as as a sequence of bounds_t for each dimension. The bounds_t of each dimension can be represented by different types. This flexibility allows the same interface to be used to declare n_bounds_t whose dimensions are fully known at compile time with fixed<int NumberT>, or to be only known at runtime with int, or only known at runtime but with a guarantee will be a multiple of an alignment with aligned<int Alignment>. For details see the Number Representation section).

When an n_container is created, its n_bounds_t always start at fixed<0> for the inclusive lower bounds of each dimension, and exclusive upper bounds match the extent of the dimension. Accessors can be translated to different index spaces as well as restrict their iteration space to subsections, which will change the n_bounds_t those accessors provide.

The following table provides information on the template arguments for `n_bounds_t`.

Template Argument	Description
`typename... TypeListT`	Comma separated list of types, where the number of types provided controls how many dimensions there are. Each type in the list identifies how the bounds of the corresponding dimension is to be represented. The order of the dimensions is the same order as C++ subscripts declaring a multi-dimensional array – from leftmost to rightmost. Requirements: types in the list be bounds_t<LowerT, UpperT>

The following table provides information on the member types of n_bounds_t

Member Types	Description
`typedef implementation-defined lower_type`	Type of n_index_t<…> returned by method lower()
`typedef implementation-defined upper_type`	Type of n_index_t<…> returned by method upper()

The following table provides information on the members of n_bounds_t.

Member	Description
`static constexpr int rank;`	Number of dimensions
`static constexpr int row_dimension = rank-1;`	Index of last dimension considered to be the row
`n_bounds_t()`	Requirements: Every bounds_t in TypeListT is default constructible. Effects: Construct n_bounds_t, uses default values of each bounds_t in TypeListT for the dimesnion sizes. In general only correctly initialized when every bounds_t has an LowerT and UpperT that is a fixed<NumberT>.
`n_bounds_t(const n_bounds_t &a_other)`	Effects: Construct n_bounds_t, copying bounds of each dimension from a_other.
`template<int DimensionT>` `auto get() const`	Requirements: DimenstionT >=0 and DimensiontT < rank. Effects: Determine the bounds of DimensionT. Returns: In the type declared by the DimensionT position of 0-based TypeListT, the bounds_t of the specified DimensionT
`lower_type lower()`	Effects: build n_index<…> representing the inclusive lower bounds for all dimensions Returns: n_index[get<0>().lower()] [get<1>().lower()] [get<…>().lower()] [get<row_dimension>().lower()]

Member	**Description**
`upper_type upper()`	Effects: build n_index<…> representing the exclusive upper bounds for all dimensions Returns: n_index[get<0>().upper()] [get<1>(). upper ()] [get<…>(). upper ()] [get<row_dimension>().upper()]
`template<typename... OtherTypeListT>` `bool contains(n_bounds_t<OtherTypeListT...> &a_other)` `const`	Requirements: rank of a_other is the same as this instance's. Effects: Determine whether each dimension of the passed n_bounds_t is fully contained within bounds of each dimenson of this object. Returns: get<0>().contains(a_other.get<0>()) && get<1>().contains(a_other.get<1>()) && get<…>().contains(a_other.get<…>()) && get<row_dimension>().contains(a_other.get<row_ dimension>())
`template<class... OtherTypeListT>` `bool operator == (const` `n_bounds_t<OtherTypeListT...> a_other) const`	Requirements: rank of a_other is the same as this instance's. Effects: Compare bounds each of dimension for equality. Only compares numeric values, not the types of each dimension. Returns: true if all dimensions are numerically equal, false otherwise.
`template<class... OtherTypeListT>` `bool operator != (const` `n_bounds_t<OtherTypeListT...> a_other) const`	Requirements: rank of a_other is the same as this instance's. Effects: Compare bounds of each dimension for inequality. Only compares numeric values, not the types of each dimension. Returns: true if any dimensions are numerically different, false otherwise.
`template<class ...OtherTypeListT>` `auto operator+ (const` `n_index_t<OtherTypeListT...> a_offset) const`	Requirements: rank of a_other is the same as this instance's. Effects: construct a n_bound_t whose types and bounds value for each dimension are determined by taking the bounds for each dimension and adding the an offset for that dimension from a_offset. Returns: n_bounds[get<0>() + a_offset.get<0>()] [get<1>() + a_offset.get<1>()] [get<…>() + a_offset.get<…>()] [get<row_dimension>() + a_offset.get< row_dimension >()]

Member	Description
`template<int DimensionT> auto rightmost_dimensions() const`	Requirements: DimenstionT >=0 and DimensiontT <= rank. Effects: Construct a n_bounds_t with a lower rank by copying the righmost DimensionT values from this instance. Returns: n_bounds[get<rank – DimensionT>()] [get<rank + 1 – DimensionT>()] [get<…>()] [get<row_dimension>()]
`template<class... OtherTypeListT> auto overlay_rightmost(const n_bounds_t<OtherTypeListT...> & a_other) const`	Requirements: rank of a_other is <= rank Effects: Construct copy of n_bounds_t where the rightmost dimensions' values are copied from a_other, effectively overlaying a_other ontop of rightmost dimensions of this instance. Returns: n_bounds[get<0>()] [get<1 >()] [get<…>()] [get<rank-a_other::rank>()] [a_other.get<0>()] [a_other.get<…>()] [a_other.get<a_other::row_dimension>()]

The following table provides information on the friend functions of n_bounds_t.

Friend Function	Description
`std::ostream& operator << (std::ostream& output_stream, const n_bounds_t & a_bounds_list)`	Effects: append string representation of a_bounds_list values to a_output_stream Returns: reference to a_output_stream for chained calls.

n_bounds_generator

Facilitates simple creation of n_bounds_t objects.

```
#include <sdlt/n_bounds.h>
```

Syntax

```
template<typename... TypeListT>
class n_bounds_generator;

namespace {
    // Instance of generator object
    n_bounds_generator<> n_bounds;
}
```

Description

The generator object provides recursively constructing operators [] for bounds_t<LowerT, UpperT> values allowing building of a n_bounds_t<…> instance one dimension at a time. Its main purpose is to allow a usage syntax that is similar to C multi-dimensional array definition:

Compare creating two n_bounds_t instances:

```
n_bounds_t<bounds_t<fixed<540>, fixed<1080>>,
           bounds_t<fixed<960>, fixed<1920>>> bounds1(bounds_t<540_fixed,1080_fixed>(),
        bounds_t<960_fixed, 1920_fixed>));

n_bounds_t<bounds_t<int, int>,
           bounds_t<int, int>> bounds2(bounds_t<int, int>(540,960),
        bounds_t<int, int>(960, 1920));
```

and the equivalent instances using the generator objects and factory functions

```
auto bounds1 = n_bounds[bounds(540_fixed, 1080_fixed)]
                       [bounds(960_fixed, 1920_fixed)];
auto bounds2 = n_bounds[bounds(540, 1080)]
                       [bounds(960, 1920)];
```

or alternatively using the operator() with n_index_t and n_extent_t generator objects

```
auto bounds1 = n_bounds(n_index[540_fixed][960_fixed],
                         n_extent[540_fixed][960_fixed]);

auto bounds2 = n_bounds(n_index[540][960],
                        n_extent[540][960]);
```

Class Hierarchy

It is expected that n_bounds_generator<…> not be directly used as a data member or parameter, instead only n_bounds_t<...> from which it is derived. The generator object n_bounds can be automatically downcast any place expecting a n_bounds_t<…>.

The following table provides information on the template arguments for n_bounds_generator

Template Argument	Description
`typename... TypeListT`	Comma separated list of types, where the number of types provided controls how many dimensions there are. Each type in the list identifies how the bounds of the corresponding dimension is to be represented. The order of the dimensions is the same order as C++ subscripts declaring a multi-dimensional array – from leftmost to rightmost. Requirements: types in the list be bounds_t<LowerT, UpperT>

The following table provides information on the types defined as members of n_bounds_generator in addition to those inherited from n_bounds_t

Member Types	Description
`typedef n_bounds_t<TypeListT...> value_type`	Type value that the any chained [] operator calls have produced.

The following table provides information on the members of n_bounds_generator in addition to those inherited from n_bounds_t

Member	Description
`n_bounds_generator()`	Requirements: TypeListT is empty Effects: Construct generator with no bounds specified
`n_bounds_generator(const n_bounds_generator` `&a_other)`	Effects: Construct generator copying any bounds values from a_other
`template<typename LowerT, typename UpperT>` `auto` `operator [] (const bounds_t<LowerT, UpperT> &` `a_bounds) const`	Effects: build a n_bounds_generator<…> with additional rightmost bounds_t<LowerT, UpperT> based dimension. Returns: n_bounds_generator<TypeListT..., bounds_t< LowerT, UpperT >>
`template<class... IndexTypeListT, class...` `ExtentTypeListT>` `auto operator () (` ` const n_index_t<IndexTypeListT...> &` `a_indices,` ` const n_extent_t<ExtentTypeListT...> &` `a_extents)  const`	Requirements: rank of a_indices is same as rank of a_extents and TypeListT be empty Effects: build a n_bounds_generator<…> where n-lower bounds are specified by a_indices, and n-upper bounds are calculated by adding a_extents to a_indices Returns: n_bounds[bounds(a_indices.get<0>(), a_indices.get<0>() + a_extents.get<0>())] [bounds(a_indices.get<1>(), a_indices.get<1>() + a_extents.get<1>())] [bounds(a_indices.get<…>(), a_indices.get<…>() + a_extents.get<…>())] [bounds(a_indices.get<row_dimension>(), a_indices.get< row_dimension >() + a_extents.get< row_dimension >())]
`value_type value() const`	Returns: n_bounds_t<…> with the correct types and values of the multi-dimensional bounds aggregated by the generator.

bounds_d Template Function

Provides a consistent way to determine the bounds of a dimension for a multi-dimensional object. `#include <sdlt/n_extent.h>`

Syntax

```
template<int DimensionT, typename ObjT>
auto bounds_d(const ObjT &a_obj)
```

Description

Consistent way to determine the bounds of a dimension for a multi-dimensional object. Can avoid extracting an entire n_bounds_t<…> when only the extent of a single dimension is needed.

Template Argument	Description
`int DimensionT`	0 based index starting at the leftmost dimension indicating which n-dimensions to query the bounds of. Requirements: DimensionT >=0 and DimensionT < ObjT::rank
`typename ObjT`	The type of n-dimensional object from which to retrieve the extent. Requirements: ObtT is one of: n_container<…> n_bounds_t<…> n_bounds_generator<…> n_container<…>::accessor n_container<…>::const_accessor or any sectioned or translated accessor.

Returns:

The correctly typed bounds_t<LowerT, UpperT> corresponding to the requested DimensionT of a_obj.

Example:

```
template <typename VolumeT>
void foo(const VolumeT & a_volume)
{
    auto bounds_z = bounds_d<0>(volume);
    auto bounds_y = bounds_d<1>(volume);
    auto bounds_x = bounds_d<2>(volume);
    for(auto z : bounds_z)
        for(auto y : bounds_y)
            for(auto x : bounds_x) {
                // …
            }
}
```

Accessors

This section provides information related to accessors for the SIMD Data Layout Templates (SDLT).

soa1d_container::accessor and aos1d_container::accessor

Lightweight object provides efficient array subscript [] access to the read or write elements from inside a soa1d_container or aos1d_container. `#include <sdlt/soa1d_container.h>` *and* `#include <sdlt/aos1d_container.h>`

Syntax

```
template <typename OffsetT> soa1d_container::accessor;
template <typename OffsetT> aos1d_container::accessor;
```

Arguments

`typename OffsetT`	The type offset that will be applied to each operator[] call determined by the type of offset passed into `soa1d_container::access(offset)`/ `aos1d_container::access(offset)` which constructs an accessor.

Description

`accessor` provides [] operator that returns a proxy object representing an Element inside the Container that can export or import the Primitive's data. Can re-access with an offset to create a new *accessor* that when accessed at [0] will really be accessing at index corresponding to the embedded offset. Lightweight and meant to be passed by value into functions or lambda closures. Use accessors in place of pointers to access the logical array data.

Member	Description
`accessor();`	Default Constructible
`accessor(const accessor &);`	Copy Constructible
`accessor & operator = (const accessor &);`	Copy Assignable
`const int & get_size_d1() const;`	Returns: Number of elements in the container.
`auto operator [] (int index_d1) const`	Returns: proxy Element representing element *at index_d1* in the container..
`template<typename IndexT_D1> auto operator [] (const IndexT_D1 index_d1);`	When: IndexT_D1 is one of the SDLT defined or generated Index types, Returns: proxy Element representing element at *index_d1* in the container.
`auto reaccess(const int offset) const;`	Returns: *accessor* with an integer-based embedded index *offset*.
`template<int IndexAlignmentT> auto reaccess(aligned_offset<IndexAlignmentT> offset) const;`	Returns: *accessor* with an `aligned_offset<IndexAlignmentT>` based embedded index *offset*.
`template<int fixed_offsetT> auto reaccess(fixed_offset<fixed_offsetT>) const;`	Returns: *accessor* with a `fixed_offset<OffsetT>` based embedded index *offset*.

soa1d_container::const_accessor and aos1d_container::const_accessor

Lightweight object provides efficient array subscript [] access to the read elements from inside a soa1d_container or aos1d_container. `#include <sdlt/soa1d_container.h>` *and* `#include <sdlt/aos1d_container.h>`

Syntax

```
template <typename OffsetT> soa1d_container::const_accessor;
template <typename OffsetT> aos1d_container::const_accessor;
```

Arguments

`typename OffsetT`	The type offset that embedded offset that will be applied to each operator[] call

Description

`const_accessor` provides [] operator that returns a proxy object representing a const Element inside the Container that can export the Primitive's data. Can re-access with an offset to create a new *const_accessor* that when accessed at [0] will really be accessing at index corresponding to the embedded offset. Lightweight and meant to be passed by value into functions or lambda closures. Use *const_accessors* in place of const pointers to access the logical array data.

Member	Description
`const_accessor();`	Default Constructible
`const_accessor(const const_accessor &);`	Copy Constructible
`const_accessor & operator = (const const_accessor &);`	Copy Assignable
`const int & get_size_d1() const;`	Returns: Number of elements in the container.
`auto operator [] (int index_d1) const`	Returns: proxy ConstElement representing element at*index_d1* in the container..
`template<typename IndexT_D1> auto operator [] (const IndexT_D1 index_d1);`	When: IndexT_D1 is one of the SDLT defined or generated Index types. Returns: proxy ConstElement representing element at *index_d1* in the container.
`auto reaccess(const int offset) const;`	Returns: *const_accessor* with an integer-based embedded index *offset*.
`template<int IndexAlignmentT> auto reaccess(aligned_offset<IndexAlignmentT> offset) const;`	Returns: *const_accessor* with an `aligned_offset<IndexAlignmentT>` based embedded index *offset*.
`template<int fixed_offsetT> auto reaccess(fixed_offset<fixed_offsetT>) const;`	Returns: *const_accessor* with a `fixed_offset<OffsetT>` based embedded index *offset*.

Accessor Concept

Accessor and const_accessor objects obtained via `n_container::access()` and `n_container::const_access()` provide access to read from or write to cells inside an `n_container`.

Syntax

The following methods return objects meeting the requirements of the accessor concept.

```
auto n_container::access();
auto n_container::const_access();
auto accessor_concept::section(n_bounds_t<…>);
auto accessor_concept::translated_to(n_index_t<…>);
auto accessor_concept::translated_to_zero();
```

Description

Accessor objects provide read/write access to individual cells of an n-dimensional container. Index values passed to a sequence of array subscript operator calls will produce a proxy concept that can import to or export the primitive data the corresponding cell inside the container.

```
auto image = make_n_container<MyStruct, layout::soa>(n_extent[128][256]);
auto acc = image.access();
MyStruct in_value(100.0f, 200.0f, 300.0f);

acc[64][128] = in_value;
MyStruct out_value = acc[64][128];

assert(out_value == in_value);
```

Accessors also know their valid iteration space, which can queried using the template function bound_d<int DimensionT>(accessor).

```
assert(bounds_d<0>(acc) == bounds(0_fixed,128));
assert(bounds_d<1>(acc) == bounds(0_fixed,256));
```

An accessor may have a non-zero index space if it has a translation embedded into it, `bounds_d` will reflect any such translation.

```
auto shifted_acc = acc.translated_to(n_index[1000][2000]);
assert(bounds_d<0>(shifted_acc) == bounds(1000,1128));
assert(bounds_d<1>(shifted_acc) == bounds(2000,2256));
```

This is useful to have a smaller sized container participate in a calculation over a portion of a larger index space, simplifying programming as the same index variable can be used, and the accessor takes care of applying the necessary translation. An accessor may represent a subsection over the original extents, bounds_d will identify the valid iteration space for that accessor.

```
auto subsection_acc = a.section(n_bounds[bounds(64,96)][bounds(128,160)]);
assert(bounds_d<0>(subsection_acc) == bounds(64, 96));
assert(bounds_d<1>(subsection_acc) == bounds(128, 160);
```

It can also be useful to have subsections be translated back to start their iteration space at 0. For efficiency, the translated_to_zero() method is provided to create an accessor shifted back to zero.

```
auto zb_sub_acc = a.section( n_bounds[bounds(64, 96)][bounds(128, 160)] ).translated_to_zero();
assert(bounds_d<0>(zb_sub_acc) == bounds(0, 32));
assert(bounds_d<1>(zb_sub_acc) == bounds(0, 32));
```

If fewer array subscript calls applied to an accessor than its rank, the result is another accessor of a lower rank. This can be useful to obtain accessors suitable to pass to code expecting lower rank accessors. Such as a obtaining a 3d accessor from a 4d container by specifying only a single index via array subscript. This has the effect of embedding the index value of the dimension inside accessor. When the final dimension is sliced, the result is a proxy object to the cell inside the container corresponding to the embedded index values inside the sliced accessors

```
auto image4d = make_n_container<MyStruct, layout::soa>(n_extent[10][20][128][256]);

MyStruct in_value(100.0f, 200.0f, 300.0f);
auto acc4d = image4d.access();
auto acc3d = acc4d[5];
auto acc2d = acc3d[10];
auto acc1d = acc2d[64];
acc1d[128] = in_value;
MyStruct out_value = acc4d[5][10][64][128];
assert(out_value == in_value);
```

The following table provides information on the requirements of the accessor concept.

Pseudo-Signature	Description
`typedef PrimitiveT primitive_type;`	Data type inside the cells of the container.
`static constexpr int rank;`	Number of free dimensions of accessor
`accessor_concept(const accessor_concept &a_other)`	Effects: constructs a copy of another accessor of the exact same type
`template<typename IndexT> element_concept operator[] (const IndexT a_index) const`	**Requirements:** rank == 1 and IndexT is one of: int, aligned<AlignmentT>, fixed<NumberT>, linear_index, or simd_index<LaneCountT> **Effects:** When only 1 free dimension is left, the operator[] will construct an element_concept which is the proxy to the cell inside the container. If this accessor was obtained with const_access(), then the proxy will provide read only interface to the cell's data. **Returns:** The proxy object to cell inside the container corresponding to the position identified by the a_index along with any embedded index values for other dimensions
`template<typename IndexT> accessor_concept operator[] (const IndexT a_index) const`	**Requirements:** rank > 1 and IndexT is one of: int, aligned<AlignmentT>, fixed<NumberT>, linear_index, or simd_index<LaneCountT> **Effects:** When 2 or more free dimensions are left, the operator[] will construct another accessor_concept of lower rank embeding a_index inside of it, effectively fixing that dimension's index value for any accesses made through the returned accessor_concept. **Returns:** The accessor_concept of lower rank (one less free dimension).
`template<int DimensionT> auto bounds_d() const`	**Requirements:** DimensionT >=0 and DimensionT < rank

Pseudo-Signature	Description
	Effects: Determine the bounds of a free dimension using DimensionT as a 0 based index starting at the leftmost dimension. **Returns:** bounds_t of the DimensionT
`auto bounds_dXX() const` `where XX is 0-19`	**Requirements:** XX >=0 and XX < rank and XX < 20 **Effects:** Non templated methods to determine the bounds of a free dimension using XX as a 0 based index starting at the leftmost dimension. **Returns:** bounds_t of the XX dimension
`template<int DimensionT>` `auto extent_d() const`	**Requirements:** DimensionT >=0 and DimensionT < rank **Effects:** Determine the extent of a free dimension using DimensionT as a 0 based index starting at the leftmost dimension. Returns: extent of the DimensionT
`auto extent_dXX() const` `where XX is 0-19`	**Requirements:** XX >=0 and XX < rank and XX < 20 **Effects:** Non templated methods to determine the extent of a free dimension using XX as a 0 based index starting at the leftmost dimension. Returns: extent of the XX dimension
`template<typename ...IndexListT>` `accessor_concept translated_to(` `    n_index_t<IndexListT...> a_n_index) const`	**Requirements:** a_n_index has same rank as the accessor **Effects:** construct an accessor_concept with an embedded translation such that accessing a_n_index will corresponds back to the current lower bounds. Easy way to think of it is that current iteration space is translated to a_n_index space. **Returns:** accessor_concept whose bounds have the same extents, but whose lower bounds start at the supplied a_n_index
`template<typename ...IndexListT>` `accessor_concept translated_to_zero() const`	**Effects:** construct an accessor_concept with an embedded translation such that accessing [0] index for all dimensions will corresponds back to the current lower bounds. Easy way to think of it is that current iteration space is translated to [0] for all free dimensions. **Returns:** accessor_concept whose bounds have the same extents, but whose lower bounds start [0]…[0]

Pseudo-Signature	Description
`template<typename ...BoundsTypeListT>` `    auto` `    section(const n_bounds_t<BoundsTypeListT...> &a_n_bounds) const`	**Requirements:** a_n_bounds has same rank as the accessor and a_n_bounds is contained by the accessors current bounds. **Effects:** construct an accessor_concept with using the supplied a_n_bounds to represent its valid iteration space. Because a_n_bounds must be contained within the existing bounds, we are effictively creating an accessor over a section of the container. Easy way to think of it is that current bounds are being restricted to a_n_bounds. Note: can be useful to chain a call `translated_to_zero()` on to the return value. **Returns:**`accessor_concept` whose bounds are set to the supplied a_n_bounds

Proxy Objects

*accessor*s can't return a reference to the Primitive because its memory layout is abstracted. Instead a Proxy object is returned. That Proxy supports importing or exporting data to and from the Container. The actual type of Proxy objects is an implementation detail, but they all support the same public interface which we will document.

Each *accessor* [index] operator returns a Proxy object.

Each *const_accessor* [index] operator returns a ConstProxy object.

The Proxy objects provide a Data Member Interface where for each data member of *value_type* they are representing, a member access method is defined which returns a new Proxy or ConstProxy representing just that data member. Users can drill down through a complex data structure to get a Proxy representing the exact data member they need versus importing and exporting the entire Primitive value.

Proxy objects also overload the following operators if the underlying *value_type* supports the operator:

==, !=, <, >, <=, >=, +, -, *, /, %, &&, ||, &, |, ^, ~, *, +, -, !, +=, -=, *=, /=, %=, >>=, <<=, &=, |=, ^=, ++, --

Proxy

Proxy object provides access to a specific Primitive, Primitive data member, or nested data member within a Primitive for an element in a container.

Description

accessor [index] or a Proxy object's Data Member Interfaces return Proxy objects. That Proxy object represents the Primitive, Primitive data member, or nested data member within a Primitive for an element in a container. The Proxy object has the following features:

- A value_type can be exported or imported from the Proxy.
  - Conversion operator is used to export the value_type
  - Alternatively the Proxy can be passed to the function `unproxy` to export a `value_type`
  - Assignment operator = is used to import value_type into the Proxy
- Overloads the following operators if the underlying value_type supports the operator
  - ==, !=, <, >, <=, >=, +, -, *, /, %, &&, ||, &, |, ^, ~, *, +, -, !, +=, -=, *=, /=, %=, >>=, <<=, &=, |=, ^=, ++, --
  - When an operator is called the following occurs:

    - value_type is exported
    - The operator applied to the exported value
    - If the operator was an assignment, the result is imported back into the Member and returns the proxy
    - Otherwise a result is returned.
- Data Member Interface.
  - For each data member of value_type
    - A member access method is defined which returns a Member proxy representing just that member.

Member Type	Description
`typedef implementation-defined value_type`	The type of the data the Proxy is representing

Member	Description
`operator value_type const () const;`	Returns: exports a copy of the Proxy's value. NOTE: constant return value prevents rvalue assignment for structs offering some protection against code that expected a modifiable reference.
`const value_type & operator = (const value_type &a_value);`	Imports a_value into container at the position the Proxy is representing. Returns: the same constant value_type it was passed. NOTE: This behavior is different from traditional assignment operators that return *this. Choice was to enable efficient chaining of assignment operators versus returning a Proxy which would have to export the value it had just imported.
`Proxy & operator = (const Proxy &other);`	Exports value from the other Proxy and imports it. Returns: A reference to this Proxy obect.
`auto name_of_values_data_member_1()const;`	Returns: Proxy instance representing the 1st data member of the value_type NOTE: actual method name is the name of the value_type's 1st data member
`auto name_of_values_data_member_2()const;`	Returns: Proxy instance representing the 2nd data member of the value_type. NOTE: actual method name is the name of the value_type's 2nd data member.
`auto name_of_values_data_member_...()const;`	Returns: Proxy instance representing the ...th data member of the value_type. NOTE: actual method name is the name of the value_type's …th data member.
`auto name_of_values_data_member_N()const;`	Returns: Proxy instance representing the Nth data member of the value_type. NOTE: actual method name is the name of the value_type's Nth data member

ConstProxy

ConstProxy object provides access to a specific constant primitive, primitive data member, or nested data member within a primitive for an element in a container.

Description

const_accessor [index] or a ConstProxy object's Data Member Interfaces return ConstProxy objects. That ConstProxy object represents the constant primitive, primitive data member, or nested data member within a primitive for an element in a container. The ConstProxy object has the following features:

- A value_type can be exported or imported from the ConstProxy.
  - Conversion operator is used to export the value_type
  - Alternatively the ConstProxy can be passed to the function `unproxy` to export a `value_type`
- Overloads the following operators if the underlying value_type supports the operator
  - ==, !=, <, >, <=, >=, +, -, *, /, %, &&, ||, &, |, ^, ~, *, +, -, !
  - When an operator is called the following occurs:
    - value_type is exported
    - The operator applied to the exported value
    - returns the result.
- Data Member Interface.
  - For each data member of value_type
    - A member access method is defined which returns a Member ConstProxy representing just that member.

Member Type	Description
`typedef implementation-defined value_type`	The type of the data the ConstProxy is representing

Member	Description
`operator value_type const () const;`	Returns: exports a copy of the ConstProxy's value. NOTE: constant return value prevents rvalue assignment for structs offering some protection against code that expected a modifiable reference.
`auto name_of_values_data_member_1()const;`	Returns: ConstProxy instance representing the 1st data member of the value_type NOTE: actual method name is the name of the value_type's 1st data member
`auto name_of_values_data_member_2()const;`	Returns: ConstProxy instance representing the 2nd data member of the value_type. NOTE: actual method name is the name of the value_type's 2nd data member.
`auto name_of_values_data_member_...()const;`	Returns: ConstProxy instance representing the ...th data member of the value_type. NOTE: actual method name is the name of the value_type's ...th data member.

Member	Description
`auto name_of_values_data_member_N()const;`	Returns: ConstProxy instance representing the Nth data member of the value_type. NOTE: actual method name is the name of the value_type's Nth data member

Number Representation

When specifying extents, positions inside of, or bounds of a container, numeric values can be represented three different ways: `fixed`, `aligned`, and `int`. `Fixed` is most precise and `int` is least precise. It is advised to use as precise specification as possible. The compiler may optimize better with more information.

Fixed

Represent a numerical constant whose value specified at compile time.

```
template <int NumberT> class fixed;
```

If offsets applied to index values inside a SIMD loop are known at compile time, then the compiler can use that information. For example, to maintain aligned access, if boundary is fixed and known to be aligned when accessing underlying data layout. When multiple accesses are happening near each other, the compiler will have the opportunity to detect which accesses occur in the same cache lines and potentially avoid prefetching the same cache line repeatedly. Additionally, if the start of an iteration space is known at compile time, if it's a multiple of the SIMD lane count, the compiler could skip generating a peel loop. Whenever possible, `fixed` values should be used over `aligned` or arbitrary integer values.

Although `std::integral_constant<int>` provides the same functionality, the library defines own type to provide overloaded operators and avoid collisions with any other code's interactions with `std::integral_constant<int>`.

The following table provides information about the template arguments for `fixed`.

Template Argument	Description
`int Number T`	The numerical value the fixed will represent.

The following table provides information about the members of `fixed`.

Member	Description
`static constexpr int value = NumberT`	The numerical value known at compile-time.
`constexpr operator value_type() const`	**Returns:** The numerical value
`constexpr value_type operator()() const;`	**Returns:** The numerical value

Constant expression arithmetic operators +,- (both unary and binary), * and / are defined for type `sdlt::fixed<>` and will be evaluated at compile-time.

The suffix `_fixed` is a C++11 user-defined equivalent literal. For example, `1080_fixed` is equivalent to `fixed<1080>`. Consider the readability of the two samples below.

```
foo3d(fixed<1080>(), fixed<1920>());
```

versus

```
foo3d(1080_fixed, 1920_fixed);
```

NOTEThis note does not apply to SYCL. The `sdlt::fixed<NumberT>` type supersedes the deprecated `sdlt::fixed_offset<OffsetT>` type found in SDLT v1. It is strongly advised to use `sdlt::fixed<NumberT>`. However, in this release, a template alias is provided mapping `sdlt::fixed_offset<OffsetT>` onto `sdlt::fixed<NumberT>`.

Aligned

Represent integer value known at compile time to be a multiple of an `IndexAlignment`.

```
template <int IndexAlignmentT> class aligned;
```

If you can tell the compiler that you know that an integer will be a multiple of known value, then, when combined with a loop index inside a SIMD loop, the compiler can use that information to maintain aligned access when accessing underlying data layout.

Internally, the integer value is converted to a block count, where:

```
block_count = value/IndexAlignmentT;
```

Overloaded math operations can then use that aligned block count as needed. The value() is represented by `AlignmentT*block_count` allowing the compiler to prove that the value() is a multiple of `AlignmentT`, which can utilize alignment optimizations.

The following table provides information about the template arguments for `aligned`.

Template Argument	Description
`int IndexAlignmentT`	The alignment the user is stating that the number is a multiple of. `IndexAlignmentT` must be a power of two.

The following table provides information about the types defined as members of `aligned`.

Member Type	Description
`typedef int value_type`	The type of the numerical value.
`typedef int block_type`	The type of the `block_count`.

The following table provides information about the members of `aligned`.

Member	Description
`static const int index_alignment`	The `IndexAlignmentT` value.
`aligned()`	Constructs empty (uninitialized) object
`explicit aligned(value_type)`	Constructs computing `block_count=a_value/IndexAlignmentT`.
`aligned(const aligned& a_other)`	Constructs copying `block_count` from `a_other`. `a_other` must have same `IndexAlignmentT`.
`template<int OtherAlignment> explicit aligned(const aligned& other)`	Constructs computing `block_count` optimized by avoiding computing other.value(). Must have `IndexAlignmentT` of a_other < `IndexAlignmentT` and other.value() be multiple of `IndexAlignmentT`.
`template<int OtherAlignment> aligned(const aligned& other)`	Constructs computing `block_count` with a multiply instead of divide. Must have `IndexAlignmentT` of a_other > `IndexAlignmentT`

Member	Description
static aligned from_block_count(block_type block_count)	Creates an instance of aligned avoiding any math by directly using supplied block_count
value_type value() const	Computes the value represented by the aligned. **Returns:** aligned_block_count()*IndexAlignmentT
operator value_type()	Conversion to int. **Returns:** value()
block_type aligned_block_count() const	Conversion to int. **Returns:** The block count

The following operations are supported for the aligned type.

Operation	Description
operator *(int), commutative	Scale value. Returns: aligned<IndexAlignmentT >
operator *(fixed<V>), commutative	Scales IndexAlignment by 2^M and value by K. Must have V=2^M*K (V is a multiple of a power of 2). **Returns:** aligned<IndexAlignmentT*(2^M)>
operator *(aligned<OtherAl>)	Scales IndexAlignment by OtherAl and block_count by argument. **Returns:** aligned<IndexAlignmentT*OtherAl>
int operator/(fixed<IndexAlignmentT>)	**Returns:** aligned_block_count()
int operator/(fixed<-IndexAlignmentT>)	**Returns:** -aligned_block_count();
int operator/(fixed<V>)	Must have abs(V)>IndexAlignmentT && IndexAlignmentT%V==0. **Returns:** aligned_block_count()/(V/IndexAlignmentT)
int operator/(fixed<V>)	Must have abs(V) < IndexAlignmentT && V %IndexAlignmentT==0 **Returns:** aligned_block_count()*(IndexAlignmentT/V)
aligned operator -()	**Returns:** Same type aligned for negated value.
aligned operator -(const aligned &) const	**Returns:** Same type aligned for value of difference.
template<int OtherAl> aligned<?> operator -(const aligned<OtherAl>&) const	Difference with other alignment. Behavior and returned alignment type depend on relation between alignments of operands.

Operation	Description
	Returns: Value for difference as lower of incoming alignments
`template<int V>` `aligned<?> operator -(const fixed<V> &) const`	Difference with fixed value. Behavior and returned alignment type depend on relation between alignments of aligned<> operand and the value of V. **Returns:** Adjusted aligned value of a difference
`aligned operator +(const aligned &)const`	**Returns:** Same type aligned for value of sum
`template<int OtherAl>` `aligned<?> operator +(const aligned<OtherAl>&) const`	Sum with other alignment. Behavior and returned alignment type depend on relation between alignments of operands. **Returns:** Value for sum as lower of incoming alignments
`template<int V>` `aligned<?> operator +(const fixed<V> &) const`	Sum with fixed value. Behavior and returned alignment type depend on relation between alignments of aligned<> operand and the value of V. **Returns:** Adjusted aligned value of a sum.
`template<int OtherAl>` `aligned operator +=(const aligned<OtherAl> &) const`	Increments value for the aligned object if `IndexAlignmentT` is compatible with `OtherAl` **Returns:** Aligned with incremented value.
`template<int OtherAl>` `aligned operator -=(const aligned<OtherAl> &) const`	Decrements value for the aligned object if `IndexAlignmentT` is compatible with `OtherAl` **Returns:** Same type aligned with decremented value.
`template<int OtherAl>` `aligned operator *=(const aligned<OtherAl> &) const`	Multiplies value for the aligned object if `IndexAlignmentT` is compatible with `OtherAl`. **Returns:** Same type aligned with multiplied value.
`template<int OtherAl>` `aligned operator /=(const aligned<OtherAl> &) const`	Divides value for the aligned object if `IndexAlignmentT` is compatible with `OtherAl` **Returns:** Same type aligned with divided value.

NOTEThis note does not apply to SYCL. The `sdlt::aligned<>` type supersedes the deprecated `sdlt::aligned_offset<>` type found in SDLT v1. It is strongly advised to use `sdlt::aligned<>`, however in this release a template alias is provided mapping `sdlt::aligned_offset<>` onto `sdlt::aligned<>`.

int

Represents an arbitrary integer value. In interfaces where fixed<> and aligned<> values supported you may also use plain old integer value. It provides least information among these three and so least facilitates compiler optimizations.

aligned_offset

Represent an integer based offset whose value is a multiple of an IndexAlignment specified at compile time. `#include <sdlt/aligned_offset.h>`

Syntax

```
template<int IndexAlignmentT>
class aligned_offset;
```

Arguments

`int IndexAlignmentT`	The index alignment the user is stating that the offset have.

Description

aligned_offset is a deprecated feature.

If we can tell the compiler that we know an offset will be a multiple of known value, then when combined with a loop index inside a SIMD loop, the compiler can use that information to maintain aligned access when accessing underlying data layout.

Internally, the offset value is converted to a block count.

```
Block Count = offsetValue/IndexAlignmentT;
```

Indices can then use that aligned block count as needed.

Member	Description
`static const int IndexAlignment = IndexAlignmentT;`	The alignment the offset is a multiple of
`explicit aligned_offset(const int offset)`	Construct instance based on offset
`static aligned_offset from_block_count(int aligned_block_count);`	Returns: Instance based on aligned_block_count, where the offset value = IndexAlignment*aligned_block_count
`int aligned_block_count() const;`	Returns: number of blocks of IndexAlignment it takes to represent the offset value.
`int value() const;`	Returns: offset value

fixed_offset

Represent an integer based offset whose value specified at compile time. `#include <sdlt/fixed_offset.h>`

Syntax

```
template <int OffsetT> fixed_offset;
```

Arguments

`int OffsetT`	The value the fixed_offset will represent

Description

fixed_offset is a deprecated feature.

If we can tell the compiler that we know an offset at compile time, then when combined with a loop index inside a SIMD loop, the compiler can use that information to maintain aligned access (should the offset be aligned) when accessing underlying data layout. When multiple accesses are happening near each other, the compiler will have the opportunity to detect which accesses occur in the same cache lines and potentially avoid prefetching the same cache line repeatedly. Whenever possible, a fixed_offset should be used over an aligned_offset or integer based offset.

Member	Description
`static constexpr int value = OffsetT`	The offset value known at compile

Indexes

soa1d_container's and aos1d_container's accessors [] operator can accept an integer based loop index. However if any modifications were applied to that loop index, the fact that it's a loop index may be lost by the compiler as it is handled before being passed to the [] operator.

To avoid this situation, SDLT provides classes to wrap loop indexes that capture multiple additions or subtractions of offsets (see the Offsets section). The resulting index can be passed to [] and preserve the original loop index and track any arithmetic with Offsets to be applied to underlying data layout.

It is common for stencil based algorithms to need to apply offsets during data access.

For a regular linear loop, use linear_index to wrap your loop index.

linear_index

Wraps an integer-based loop index that is iterating linearly through an iteration space. `#include <sdlt/linear_index.h>`

Syntax

```
class linear_index;
```

Description

Inside of a linear loop, wrap the loop index with a `linear_index` to allow addition or subtraction of offsets.

Member	Description
`explicit linear_index(int an_index);`	Construct instance from a loop index
`int value() const;`	Returns the original loop index

n_index_t

Variadic template class n_index_t describes a position inside of the N-dimensional container. Specifically, the number of dimensions and the of index value of each.

Syntax

```
template<typename... TypeListT>
class n_index_t
```

Description

`n_index_t` represents a position inside an n-dimensional space as a sequence of index value for each dimension. The index of each dimension can be represented by different types. This flexibility allows the same interface to be used to declare `n_index_t` with indices that are fully known at compile time with `fixed<int NumberT>`, or to be only known at runtime with `int`, or only known at runtime but with a guarantee will be a multiple of an alignment with `aligned<int Alignment>`. For more details, see the Number representation section.

Objects of this class may be used to identify a cell in a container, describe the inclusive lower bounds for `n_bounds()`, n-dimensional position for accessor's *translated_to()*.

The following table provides information about the template arguments for `n_index_t`.

Template Argument	Description
`typename... TypeListT`	Comma separated list of types, where the number of types provided controls how many dimensions there are. Each type in the list identifies how the index of the corresponding dimension is to be represented. The order of the dimensions is the same order as C++ subscripts declaring a multi-dimensional array, from leftmost to rightmost. **Requirements**: Type must be `int`, or `fixed<NumberT>`, or `aligned<AlignmentT>`.

The following table provides information about the members of `n_index_t`

Member	Description
`static constexpr int rank;`	Number of dimensions.
`static constexpr int row_dimension = rank-1;`	Index of last dimension, *row.*
`n_index_t()`	Default constructor. Uses default values for extent types. **Requirements:** Every type in TypeListT is default constructible. **Effects:** Construct n_index_t, uses default values of each type in TypeListT for the dimesnion sizes. In general only correctly initialized when every type is a fixed<NumberT>.
`n_index_t(const n_extent_t &a_other)`	Copy constructor. **Effects:** Construct n_index_t, copying index value of each dimension from a_other.
`explicit n_index_t(const TypeListT & … a_values)`	**Returns:** The last extent in its native type **Effects:** Construct n_index_t, initializing each dimension with the corresponding value from the list of *a_values* passed as an argument. In use, *a_values* is a comma separate list of values whose length and types are defined by TypeListT.

Member	Description
`template<int DimensionT>` `auto get() const`	**Requirements:** DimenstionT >=0 and DimensiontT < rank. **Effects:** Determine the index value of DimensionT. **Returns:** In the type declared by the DimensionT position of 0-based TypeListT, the index value of the specified *DimensionT*
`n_index_t operator +() const`	**Effects:** Determine the positive unary value of each dimension's index, effectively no operation is performed **Returns:** Copy of the current instance.
`auto operator -() const`	**Effects:** Determine the negative unary value of each dimension's index **Returns:** n_index[-get<0>()] [-get<1>()] [-get<…>()] [-get<row_dimension>()]
`template<class... OtherTypeListT>` `auto operator +(` `const n_index_t<OtherTypeListT...> &` `a_other) const`	**Requirements:** Rank of a_other is the same as this instance's. **Effects:** Build n_index_t whose values are the result of adding the index value for each dimension with those of a_other **Returns:** n_index[get<0>() + a_other.get<0>()] [get<1>() + a_other.get<1>()] [get<…>() + a_other.get<…>()] [get<row_dimension>() + a_other.get<row_dimension>()]
`template<class... OtherTypeListT>` `auto operator -(` `const n_index_t<OtherTypeListT...> &` `a_other) const`	**Requirements:** Rank of a_other is the same as this instance's. **Effects:** Build n_index_t whose values are the result of subtracting the index value for each dimension of a_other with this instance's. **Returns:** n_index[get<0>() - a_other.get<0>()] [get<1>() - a_other.get<1>()] [get<…>() - a_other.get<…>()] [get<row_dimension>() - a_other.get<row_dimension>()]
`template<class... OtherTypeListT>` `bool operator == (const` `n_index_t<OtherTypeListT...> a_other) const`	**Requirements:** Rank of a_other is the same as this instance's.

Member	Description
	Effects: Compare index of each dimension for equality. Only compares numeric values, not the types of each dimension. **Returns:***true* if all dimensions are numerically equal, *false* otherwise.
`template<class... OtherTypeListT> bool operator != (const n_index_t<OtherTypeListT...> a_other) const`	**Requirements:** Rank of a_other is the same as this instance's. **Effects:** Compare index of each dimension for inequality. Only compares numeric values, not the types of each dimension. **Returns:** *true* if any dimensions are numerically different, *false* otherwise.
`template<int DimensionT> auto rightmost_dimensions() const`	**Requirements:** DimenstionT >=0 and DimensiontT <= rank. **Effects:** Construct a n_index_t with a lower rank by copying the righmost DimensionT values from this instance. **Returns:** n_index[get<rank - DimensionT>()] [get<rank + 1 - DimensionT>()] [get<…>()] [get<row_dimension>()]
`template<class... OtherTypeListT> auto overlay_rightmost(const n_index_t<OtherTypeListT...> & a_other) const`	**Requirements:** rank of a_other is <= rank **Effects:** Construct copy of n_index_t where the rightmost dimensions' values are copied from a_other, effectively overlaying a_other ontop of rightmost dimensions of this instance. **Returns**: n_index[get<0>()] [get<1 >()] [get<…>()] [get<rank-a_other::rank>()] [a_other.get<0>()] [a_other.get<…>()] [a_other.get<a_other::row_dimension>()]

The following table provides information about the friend functions of `n_index_t`

Friend Function	Description
`std::ostream& operator << (std::ostream& output_stream, const n_index_t & a_indices)`	**Effects:** Append string representation of a_indices' values to a_output_stream. **Returns:** Reference to a_output_stream for chained calls.

n_index_generator

To facilitate simpler creation of `n_index_t` *objects, the generator object* `n_index` *is provided.*

Syntax

```
template<typename... TypeListT>
class n_index_generator;

namespace {
    // Instance of generator object
    n_index_generator<> n_index;
}
```

Description

The generator object provides recursively constructing operators [] for `fixed<>`, `aligned<>`, and `integer` values allowing building of a `n_index_t<…>` instance one dimension at a time. Its main purpose is to allow a usage syntax that is similar to C multi-dimensional array definition.

Compare the following examples, instantiating three `n_index_t` instances, and using the generator object to instantiate equivalent instances.

```
n_index_t<int, int> idx1(row, col);
n_index_t<int, aligned<16>> idx2(row, aligned<16>(col));
n_index_t<fixed<540>, fixed<960>> idx3(540_fixed, 960_fixed);
```

```
auto idx1 = n_index[row][col];
auto idx2 = n_index[row][aligned<16>(col)];
auto idx3 = n_index[540_fixed][960_fixed];
```

Class Hierarchy

It is expected that `n_index_generator` < … > not be directly used as a data member or parameter, instead only `n_index_t` <...> from which it is derived. The generator object `n_index` can be automatically downcast any place expecting an `n_index_t<…>` .

The following table provides the template arguments for `n_index_generator`

Template Argument	Description
`typename... TypeListT`	Comma separated list of types, where the number of types provided controls how many dimensions the generator currently represents. Each type in the list identifies how the size of the corresponding dimension is to be represented. The order of the dimensions is the same order as C++ subscripts declaring a multi-dimensional array – from leftmost to rightmost. **Requirements**: Type is `int`, `fixed<NumberT>`, or `aligned<AlignmentT>`.

The following table provides information on the types defined as members of `n_index_generator` in addition to those inherited from `n_index_t`.

Member Type	Description
`typedef n_index_t<TypeListT...> value_type`	Type value that the any chained [] operator calls have produced.

The following table provides information on the members of `n_index_generator` in addition to those inherited from `n_index_t`

Member	Description
`n_index_generator ()`	**Requirements:** TypeListT is empty. **Effects:** Construct generator with no indices specified.
`n_index_generator (const n_index_generator &a_other)`	**Effects:** Construct generator copying any index values from a_other
`n_index_generator<TypeListT..., int> operator [] (int a_index) const`	**Requirements:** a_size >= 0. **Returns:** n_index_generator<…> with additional rightmost integer based index.
`n_index_generator<TypeListT..., fixed<NumberT>> operator [] (fixed<NumberT> a_index) const`	**Requirements:** a_size >= 0. **Returns:** n_index_generator<…> with additional rightmost fixed<NumberT> index.
`n_index_generator<TypeListT..., aligned<AlignmentT>> operator [] (aligned<AlignmentT> a_index)`	**Requirements:** a_size >= 0 **Returns:** n_index_generator<…> with additional rightmost aligned<AlignmentT> based index.
`value_type value() const`	**Returns:** n_extent_t<…> with the correct types and values of the multi-dimensional extents aggregated by the generator.

index_d template function

Syntax

```
template<int DimensionT, typename ObjT>
auto index_d(const ObjT &a_obj)
```

Description

The template function offers a consistent way to determine the index of a dimension for a multi-dimensional object. It can avoid extracting an entire `n_index_t<…>` when only the extent of a single dimension is needed.

Template Argument	Description
`int DimensionT`	0 based index starting at the leftmost dimension indicating which n-dimensions to query the index of. **Requirements:** DimensionT >=0 and DimensionT < ObjT::rank
`typename ObjT`	The type of n-dimensional object from which to retrieve the extent. **Requirements:** ObtT is one of: n_index_t<…> n_index_generator<…>

Returns

The correctly typed index corresponding to the requested DimensionT of a_obj.

Example

```
template <typename IndicesT>
void foo(const IndicesT & a_pos)
{
    int z = index_d<0>(a_pos);
    int y = index_d<1>(a_pos);
    int x = index_d<2>(a_pos);
    /…
}
```

Convenience and Correctness

Users can include a single header file `sdlt.h` that includes all the supported public features, or users can include the individual headers of features they will be using (which might build faster). In other words,

```
#include <sdlt/sdlt.h>
```

instead of

```
#include <sdlt/primitive.h>
#include <sdlt/soa1d_container.h>
```

For convenience, SDLT provides a macro to encapsulate `#pragma forceinline recursive`.

```
SDLT_INLINE_BLOCK
```

SDLT reduces overhead by trusting the programmer to pass it valid values for template and function parameters. Adding conditional checks inside of a SIMD loop can cause unnecessary code generation and inhibit vectorization by creating multiple exit points in a loop. To assist in verifying that a program is indeed passing valid values to SDLT, the programmer can add a compilation flag to their build to define `SDLT_DEBUG=1`.

```
-DSDLT_DEBUG=1
```

If `_DEBUG` is defined and `SDLT_DEBUG` has not been defined to 0 or 1, then `SDLT_DEBUG` is automatically set to 1. When set to 1, every operator[] is bounds checked and all addresses are validated for correct alignment. It is very useful for tracking down any usage bugs.

The macro `__SDLT_VERSION` is predefined to be 2001. Programs could use it for conditional compilation if incompatibilities arise in future updates.

C++ implementations of `std::min` and `std::max` sometimes have a negative impact on performance. SDLT defines `min_val` and `max_val` that help avoid such performance penalties.

max_val

Return the right value if the right value is greater than left, otherwise returns the left value. `#include <sdlt/min_max_val.h>`

Syntax

```
template<typename T>
T max_val(const T left, const T right);
```

Arguments

`typename T`	The type of the left and right values

Description

C++ implementations of std::min and std::max create a conditional control flow that returns references to its parameters, which may cause inefficient vector code generation. max_val is a really simple template that returns by value instead of reference, allowing more efficient vector code to be generated. For most cases the algorithm didn't need a reference to the inputs and a copy by value should suffice. It should inline, adding no overhead. Inside of SIMD loops, we suggest using sdlt::max_val in place of std::max.

Requires < operator be defined for the type T.

min_val

Return the left value if the right value is greater than left, otherwise returns the right value. `#include <sdlt/min_max_val.h>`

Syntax

```
template<typename T>
T min_val(const T left, const T right);
```

Arguments

`typename T`	The type of the left and right values

Description

C++ implementations of std::min and std::max create a conditional control flow that returns references to its parameters, which may cause inefficient vector code generation. min_val is a really simple template that returns by value instead of reference, allowing more efficient vector code to be generated. For most cases the algorithm didn't need a reference to the inputs and a copy by value should suffice. It should inline, adding no overhead. Inside of SIMD loops, we suggest using sdlt::min_val in place of std::min.

Requires < operator be defined for the type T.

Examples

The example programs in this section demonstrate the following:

- The efficiency of using SDLT and its Structure of Arrays approach rather than a typical Array of Structures
- Construction of more complex SDLT primitives
- Performance improvement in case of a forward-dependency
- Use of offsets and calling methods on the SDLT primitive
- RGB to YUV conversion

Efficiency with Structure of Arrays Example

This example demonstrates the efficiency of using a Structure of Arrays (SoA) approach by comparing the assembly generated from a simple SIMD loop using an Array of Structures (AoS) approach with the assembly generated using the SoA approach of SDLT.

Array of Structures: Non-unit stride access version

Source:

```
#include <stdio.h>

#define N 1024

typedef struct RGBs {
    float r;
    float g;
```

```
    float b;
} RGBTy;


void main()
{
    RGBTy a[N];
    #pragma omp simd
    for (int k = 0; k<N; ++k) {
        a[k].r = k*1.5;  // non-unit stride access
        a[k].g = k*2.5;  // non-unit stride access
        a[k].b = k*3.5;  // non-unit stride access
    }
    std::cout << "k =" << 10 <<
        ", a[k].r =" << a[10].r <<
        ", a[k].g =" << a[10].g <<
        ", a[k].b =" << a[10].b << std::endl;
}
```

AVX2 assembly generated (69 instructions):

```
..TOP_OF_LOOP:
        vcvtdq2ps %ymm7, %ymm1
        lea       (%rax), %rcx
        vcvtdq2ps %ymm5, %ymm2
        vpaddd    %ymm3, %ymm7, %ymm7
        vpaddd    %ymm3, %ymm5, %ymm5
        vmulps    %ymm1, %ymm4, %ymm8
        vmulps    %ymm1, %ymm6, %ymm12
        vmulps    %ymm2, %ymm6, %ymm14
        vmulps    %ymm1, %ymm0, %ymm1
        vmulps    %ymm2, %ymm4, %ymm10
        addl      $16, %edx
        vextractf128 $1, %ymm8, %xmm9
        vmovss    %xmm8, (%rsp,%rcx)
        vmovss    %xmm9, 48(%rsp,%rcx)
        vextractps $1, %xmm8, 12(%rsp,%rcx)
        vextractps $2, %xmm8, 24(%rsp,%rcx)
        vextractps $3, %xmm8, 36(%rsp,%rcx)
        vmulps    %ymm2, %ymm0, %ymm8
        vextractps $1, %xmm9, 60(%rsp,%rcx)
        vextractps $2, %xmm9, 72(%rsp,%rcx)
        vextractps $3, %xmm9, 84(%rsp,%rcx)
        vextractf128 $1, %ymm12, %xmm13
        vextractf128 $1, %ymm14, %xmm15
        vextractf128 $1, %ymm1, %xmm2
        vextractf128 $1, %ymm8, %xmm9
        vmovss    %xmm12, 4(%rsp,%rax)
        vmovss    %xmm13, 52(%rsp,%rax)
        vextractps $1, %xmm12, 16(%rsp,%rax)
        vextractps $2, %xmm12, 28(%rsp,%rax)
        vextractps $3, %xmm12, 40(%rsp,%rax)
        vextractps $1, %xmm13, 64(%rsp,%rax)
        vextractps $2, %xmm13, 76(%rsp,%rax)
        vextractps $3, %xmm13, 88(%rsp,%rax)
        vmovss    %xmm14, 100(%rsp,%rax)
        vextractps $1, %xmm14, 112(%rsp,%rax)
        vextractps $2, %xmm14, 124(%rsp,%rax)
        vextractps $3, %xmm14, 136(%rsp,%rax)
```

```
        vmovss    %xmm15, 148(%rsp,%rax)
        vextractps $1, %xmm15, 160(%rsp,%rax)
        vextractps $2, %xmm15, 172(%rsp,%rax)
        vextractps $3, %xmm15, 184(%rsp,%rax)
        vmovss    %xmm1, 8(%rsp,%rax)
        vextractps $1, %xmm1, 20(%rsp,%rax)
        vextractps $2, %xmm1, 32(%rsp,%rax)
        vextractps $3, %xmm1, 44(%rsp,%rax)
        vmovss    %xmm2, 56(%rsp,%rax)
        vextractps $1, %xmm2, 68(%rsp,%rax)
        vextractps $2, %xmm2, 80(%rsp,%rax)
        vextractps $3, %xmm2, 92(%rsp,%rax)
        vmovss    %xmm8, 104(%rsp,%rax)
        vextractps $1, %xmm8, 116(%rsp,%rax)
        vextractps $2, %xmm8, 128(%rsp,%rax)
        vextractps $3, %xmm8, 140(%rsp,%rax)
        vmovss    %xmm9, 152(%rsp,%rax)
        vextractps $1, %xmm9, 164(%rsp,%rax)
        vextractps $2, %xmm9, 176(%rsp,%rax)
        vextractps $3, %xmm9, 188(%rsp,%rax)
        addq      $192, %rax
        vextractf128 $1, %ymm10, %xmm11
        vmovss    %xmm10, 96(%rsp,%rcx)
        vmovss    %xmm11, 144(%rsp,%rcx)
        vextractps $1, %xmm10, 108(%rsp,%rcx)
        vextractps $2, %xmm10, 120(%rsp,%rcx)
        vextractps $3, %xmm10, 132(%rsp,%rcx)
        vextractps $1, %xmm11, 156(%rsp,%rcx)
        vextractps $2, %xmm11, 168(%rsp,%rcx)
        vextractps $3, %xmm11, 180(%rsp,%rcx)
        cmpl      $1024, %edx
        jb        ..TOP_OF_LOOP
```

Structure of Arrays: Using SDLT for unit stride access

To introduce the use of SDLT, the code below will:

- declare a primitive,
- use an `soa1d_container` instead of an array
- use an accessor inside a SIMD loop to generate efficient code
- use a proxy object's data member interface to access individual data members of an element inside the container

Source:

```
#include <stdio.h>
#include <sdlt/sdlt.h>

#define N 1024

typedef struct RGBs {
    float r;
    float g;
    float b;
} RGBTy;

SDLT_PRIMITIVE(RGBTy, r, g, b)

void main()
```

```
{
    // Use SDLT to get SOA data layout
    sdlt::soa1d_container<RGBTy> aContainer(N);
    auto a = aContainer.access();

    // use SDLT Data Member Interface to access struct members r, g, and b.
    // achieve unit-stride access after vectorization
    #pragma omp simd
    for (int k = 0; k<N; k++) {
        a[k].r() = k*1.5;
        a[k].g() = k*2.5;
        a[k].b() = k*3.5;
    }
    std::cout << "k =" << 10 <<
        ", a[k].r =" << a[10].r() <<
        ", a[k].g =" << a[10].g() <<
        ", a[k].b =" << a[10].b() << std::endl;
}
```

AVX2 assemply generated (19 instructions):

```
..TOP_OF_LOOP:
        vpaddd    %ymm4, %ymm3, %ymm12
        vcvtdq2ps %ymm3, %ymm7
        vcvtdq2ps %ymm12, %ymm10
        vmulps    %ymm7, %ymm2, %ymm5
        vmulps    %ymm7, %ymm1, %ymm6
        vmulps    %ymm7, %ymm0, %ymm8
        vmulps    %ymm10, %ymm2, %ymm3
        vmulps    %ymm10, %ymm1, %ymm9
        vmulps    %ymm10, %ymm0, %ymm11
        vmovups   %ymm5, (%r13,%rax,4)
        vmovups   %ymm6, (%r15,%rax,4)
        vmovups   %ymm8, (%rbx,%rax,4)
        vmovups   %ymm3, 32(%r13,%rax,4)
        vmovups   %ymm9, 32(%r15,%rax,4)
        vmovups   %ymm11, 32(%rbx,%rax,4)
        vpaddd    %ymm4, %ymm12, %ymm3
        addq      $16, %rax
        cmpq      $1024, %rax
        jb        ..TOP_OF_LOOP
```

Both versions appear to have unrolled the loop twice. When examining the assembly generated for AVX2 instruction set, we can see a measurable reduction in the number of instructions (19 vs. 69) when we are able to perform unit stride access using SDLT. Also, at runtime, the `soa1d_container` aligned its data allocation and will gain any of the architectural advantages that come with using aligned instead of unaligned SIMD stores.

Complex SDLT Primitive Construction Example

This example demonstrates use of nested primitives and the use of an accessor inside a SIMD loop to generate efficient code.

```
#include <stdio.h>
#include <sdlt/sdlt.h>

#define N 1024

typedef struct XYZs {
    float x;
```

```
    float y;
    float z;
} XYZTy;

SDLT_PRIMITIVE(XYZTy, x, y, z)

typedef struct RGBs {
    float r;
    float g;
    float b;
    XYZTy w;
} RGBTy;

SDLT_PRIMITIVE(RGBs, r, g, b, w)

void main()
{
    sdlt::soa1d_container<RGBTy> aContainer(N);
    auto a = aContainer.access();

    #pragma omp simd
    for (int k = 0; k<N; k++) {
        RGBTy c;
        c.r = k*1.5f;
        c.g = k*2.5f;
        c.b = k*3.5f;
        c.w.x = k*4.5f;
        c.w.y = k*5.5f;
        c.w.z = k*6.5f;
        a[k] = c;
    }
    const RGBTy c = a[10];
    printf("k = %d, a[k].r = %f, a[k].g = %f, a[k].b = %f \n",
        10, c.r, c.g, c.b);

    printf("k = %d, a[k].w.x = %f, a[k].w.y = %f, a[k].w.z = %f \n",
        10, c.w.x, c.w.y, c.w.z);
```

Forward Dependency Example

This example demonstrates the declaration of a Structure of Arrays (SoA) interacting with a forward dependency.

```
#include <stdio.h>
#include <sdlt/primitive.h>
#include <sdlt/soa1d_container.h>

#define N 1024

typedef struct RGBs {
    float r;
    float g;
    float b;
} RGBTy;

SDLT_PRIMITIVE(RGBTy, r, g, b)

void main()
{
```

```
    // RGBTy a[N];  // AOS data layout

    sdlt::soa1d_container<RGBTy> aContainer(N);
    auto a = aContainer.access();    // SOA data layout

    // use SDLT access method to access struct members r, g, and b.
    // with unit-stride access  after vectorization
    #pragma omp simd
    for (int k = 0; k<N; k++) {
        a[k].r() = k*1.5;
        a[k].g() = k*2.5;
        a[k].b() = k*3.5;
    }

    // Test forward-dependency on SOA memory access
    #pragma omp simd
    for (int i = 0; i<N - 1; i++) {
        sdlt::linear_index k(i);
        a[k].r() = a[k + 1].r() + k*1.5;
        a[k].g() = a[k + 1].g() + k*2.5;
        a[k].b() = a[k + 1].b() + k*3.5;
    }
    std::cout << "k =" << 10 <<
        ", a[k].r =" << a[10].r() <<
        ", a[k].g =" << a[10].g() <<
        ", a[k].b =" << a[10].b() << std::endl;
}
```

Use of Offsets and Methods on a SDLT Primitive Example

This example demonstrates a linearized 2d stencil using embedded offsets and calling methods on the primitive.

```
#include <sdlt/sdlt.h>

// Typical C++ object to represent a pixel in an image
struct RGBs
{
    float red;
    float green;
    float blue;

    RGBs() {}
    RGBs(const RGBs &iOther)
        : red(iOther.red)
        , green(iOther.green)
        , blue(iOther.blue)
    {
    }

    RGBs & operator =(const RGBs &iOther)
    {
        red = iOther.red;
        green = iOther.green;
        blue = iOther.blue;
        return *this;
    }

    RGBs operator + (const RGBs &iOther) const
```

```
    {
        RGBs sum;
        sum.red = red + iOther.red;
        sum.green = green + iOther.green;
        sum.blue = blue + iOther.blue;
        return sum;
    }

    RGBs operator * (float iScalar) const
    {
        RGBs scaledColor;
        scaledColor.red = red * iScalar;
        scaledColor.green = green * iScalar;
        scaledColor.blue = blue * iScalar;
        return scaledColor;
    }
};

SDLT_PRIMITIVE(RGBs, red, green, blue)

const int StencilHaloSize = 1;
const int width = 1920;
const int height = 1080;

template<typename AccessorT> void loadImageStub(AccessorT) {}
template<typename AccessorT> void saveImageStub(AccessorT) {}


// performs average color filtering with neighbors left,right,above,below
void main(void)
{
    // We are padding +-1 so we can avoid boundary conditions
    const int paddedWidth = width + 2 * StencilHaloSize;
    const int paddedHeight = height + 2 * StencilHaloSize;
    int elementCount = paddedWidth*paddedHeight;
    sdlt::soa1d_container<RGBs> inputImage(elementCount);
    sdlt::soa1d_container<RGBs> outputImage(elementCount);

    loadImageStub(inputImage.access());

    SDLT_INLINE_BLOCK
    {
        const int endOfY = StencilHaloSize + height;
        const int endOfX = StencilHaloSize + width;
        for (int y = StencilHaloSize; y < endOfY; ++y)
        {
            // Embed offsets into Accessors to get the to correct row
            auto prevRow = inputImage.const_access((y - 1)*paddedWidth);
            auto curRow = inputImage.const_access(y*paddedWidth);
            auto nextRow = inputImage.const_access((y + 1)*paddedWidth);

            auto outputRow = outputImage.access(y*paddedWidth);

            #pragma omp simd
            for (int ix = StencilHaloSize; ix < endOfX; ++ix)
            {
                sdlt::linear_index x(ix);
```

```
                const RGBs color1 = curRow[x - 1];
                const RGBs color2 = curRow[x];
                const RGBs color3 = curRow[x + 1];
                const RGBs color4 = prevRow[x];
                const RGBs color5 = nextRow[x];
                // Despite looking like AOS code, compiler is able to create
                // privatized instances and call inlinable methods on the objects
                // keeping the algorithm at very high level
                const RGBs sumOfColors = color1 + color2 + color3 + color4 + color5;
                const RGBs averageColor = sumOfColors*(1.0f / 5.0f);
                outputRow[x] = averageColor;
            }
        }
    }
    saveImageStub(outputImage.access());
}
```

RGB to YUV Conversion Example

This example converts a 2D image from the RGB format to the YUV format. It demonstrates how storing both images in 2D SoA `n_container`s can improve performance.

```
#include <iostream>
#include <sdlt/sdlt.h>
using namespace sdlt;
#define WIDTH 1024
#define HEIGHT 1024

struct RGBs {
    float r;
    float g;
    float b;
};

struct YUVs {
    float y;
    float u;
    float v;

    YUVs(){ };

    YUVs& operator=(const RGBs &tmp){
        y = 0.229f * tmp.r + 0.587f * tmp.g + 0.114f * tmp.b;
        u = -0.147f * tmp.r - 0.289f * tmp.g + 0.436f * tmp.b;
        v = 0.615 * tmp.r - 0.515f * tmp.g - 0.100 * tmp.b;
        return *this;
    }
    YUVs(const RGBs &tmp){
        y = 0.229f * tmp.r + 0.587f * tmp.g + 0.114f * tmp.b;
        u = -0.147f * tmp.r - 0.289f * tmp.g + 0.436f * tmp.b;
        v = 0.615 * tmp.r - 0.515f * tmp.g - 0.100 * tmp.b;
   }
};

SDLT_PRIMITIVE(RGBs, r, g, b)
SDLT_PRIMITIVE(YUVs, y, u, v)

int main(){
    typedef layout::soa<> LayoutT;
```

```
    n_extent_t<int, int> extents(HEIGHT, WIDTH);

    /*  Creating a typedef for SoA N-dimensional container.
        RGBTy and YUVTy are user defined structures whose collection needs to be stored in SoA
format in memory.
        Layout in memory specified as layout::soa.
        In the below case N-dimensional SoA container is used in 2-D context
    */
    typedef sdlt::n_container< RGBs, LayoutT, decltype(extents) > ContainerRGB;
    typedef sdlt::n_container< YUVs, LayoutT, decltype(extents) > ContainerYUV;

    //Instantiate Input and Output Containers
    ContainerRGB inputRGB(extents);
    ContainerYUV outputYUV(extents);

    auto input = inputRGB.const_access();   //Get Constant Accessor object for inputRGB
    auto output = outputYUV.access();       //Get Accessor object for outputYUV

    //Select the iteration range in each dimension
    const auto iRGB1 = bounds_d<1>(input);  //bound_d<1>(input);
    const auto iRGB0 = bounds_d<0>(input);  //bound_d<0>(input);

    for(int y = iRGB0.lower(); y < iRGB0.upper(); y++)
    {
        #pragma simd
        for (int x = iRGB1.lower(); x < iRGB1.upper(); x++){
            const RGBs temp1 = input[y][x];
            YUVs temp2 = temp1;
            output[y][x] = temp2;
        }
    }
    return 0;
}
```

Intel® C++ Class Libraries

The Intel® C++ Class Libraries enable Single-Instruction, Multiple-Data (SIMD) operations. The principle of SIMD operations is to exploit microprocessor architecture through parallel processing. The effect of parallel processing is increased data throughput using fewer clock cycles. The objective is to improve application performance of complex and computation-intensive audio, video, and graphical data bit streams.

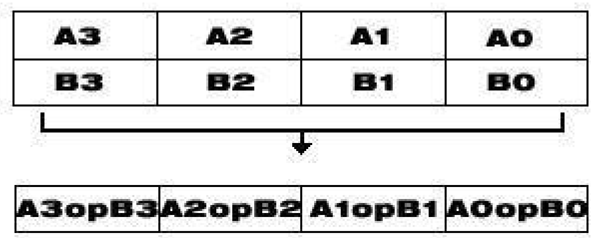
Hardware and Software Requirements

The Intel® C++ Class Libraries are functions abstracted from the instruction extensions available on Intel® processors.

Details About the Libraries

The Intel® C++ Class Libraries for SIMD Operations provide a convenient interface to access the underlying instructions for processors as specified above. These processor-instruction extensions enable parallel processing using the single instruction-multiple data (SIMD) technique as illustrated in the following figure.

SIMD Data Flow



Performing four operations with a single instruction improves efficiency by a factor of four for that particular instruction.

These new processor instructions can be implemented using assembly inlining, intrinsics, or the C++ SIMD classes. Compare the coding required to add four 32-bit floating-point values, using each of the available interfaces:

Comparison Between Inlining, Intrinsics, and Class Libraries

The table below shows an addition of four single-precision floating-point values using assembly inlining, intrinsics, and the libraries. You can see how much easier it is to code with the Intel C++ SIMD Class Libraries. Besides using fewer keystrokes and fewer lines of code, the notation is like the standard notation in C++, making it much easier to implement over other methods.

Assembly Inlining	Intrinsics	SIMD Class Libraries
`... __m128 a,b,c; __asm{ movaps xmm0,b movaps xmm1,c addps xmm0,xmm1 movaps a, xmm0 } ...`	`#include <xmmintrin.h> ... __m128 a,b,c; a = _mm_add_ps(b,c); ...`	`#include <fvec.h> ... F32vec4 a,b,c; a = b +c; ...`

C++ Classes and SIMD Operations

Use of C++ classes for SIMD operations allows for operating on arrays or vectors of data in a single operation. Consider the addition of two vectors, `A` and `B`, where each vector contains four elements. Using an integer vector class, the elements `A[i]` and `B[i]` from each array are summed in the typical method of adding elements using a loop example snippet below.

```
int a[4], b[4], c[4];
for (i=0; i<4; i++) /* needs four iterations */
c[i] = a[i] + b[i]; /* computes c[0], c[1], c[2], c[3] */
```

The following example shows the same results using one operation with an integer class, showing the SIMD method of adding elements using `Ivec` classes.

```
Is16vec4 ivecA, ivecB, ivec C; /*needs one iteration*/
ivecC = ivecA + ivecB; /*computes ivecC0, ivecC1, ivecC2, ivecC3 */
```

Available Classes

The C++ SIMD classes provide parallelism, which is not easily implemented using typical mechanisms of C++. The following table shows how the C++ classes use the SIMD classes and libraries.

SIMD Vector Classes

Instruction Set	Class	Signedness	Data Type	Size	Elements	Header File
MMX™ Technology	I64vec1	unspecified	__m64	64	1	ivec.h
	I32vec2	unspecified	int	32	2	ivec.h
	Is32vec2	signed	int	32	2	ivec.h
	Iu32vec2	unsigned	int	32	2	ivec.h
	I16vec4	unspecified	short	16	4	ivec.h
	Is16vec4	signed	short	16	4	ivec.h
	Iu16vec4	unsigned	short	16	4	ivec.h
	I8vec8	unspecified	char	8	8	ivec.h
	Is8vec8	signed	char	8	8	ivec.h
	Iu8vec8	unsigned	char	8	8	ivec.h
Intel® Streaming SIMD Extensions (Intel® SSE)	F32vec4	unspecified	float	32	4	fvec.h
	F32vec1	unspecified	float	32	1	fvec.h
Intel® Streaming SIMD Extensions 2 (Intel® SSE2)	F64vec2	unspecified	double	64	2	dvec.h
	I128vec1	unspecified	__m128i	128	1	dvec.h
	I64vec2	unspecified	long int	64	2	dvec.h
	I32vec4	unspecified	int	32	4	dvec.h
	Is32vec4	signed	int	32	4	dvec.h
	Iu32vec4	unsigned	int	32	4	dvec.h
	I16vec8	unspecified	int	16	8	dvec.h
	Is16vec8	signed	int	16	8	dvec.h
	Iu16vec8	unsigned	int	16	8	dvec.h
	I8vec16	unspecified	char	8	16	dvec.h
	Is8vec16	signed	char	8	16	dvec.h
	Iu8vec16	unsigned	char	8	16	dvec.h
Intel® Advanced Vector Extensions (Intel® AVX)	F32vec8	unspecified	float	32	8	dvec.h
	F64vec4	unspecified	double	64	4	dvec.h

Instruction Set	Class	Signedness	Data Type	Size	Elements	Header File
Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Foundation	F32vec16	unspecified	float	32	16	`dvec.h`
	F64vec8	unspecified	double	64	8	`dvec.h`
	M512vec	unspecified	__m512i	512	1	`dvec.h`
	I32vec16	unspecified	int	32	16	`dvec.h`
	Is32vec16	signed	int	32	16	`dvec.h`
	Iu32vec16	unsigned	int	32	16	`dvec.h`
	I64vec8	unspecified	long int	64	8	`dvec.h`
	Is64vec8	signed	long int	64	8	`dvec.h`
	Iu64vec8	unsigned	long int	64	8	`dvec.h`
Intel® AVX-512 Byte and Word Instructions (BWI)	I16vec32	unspecified	int	16	32	`dvec.h`
	`Is16vec32`	signed	int	16	32	`dvec.h`
	`Iu16vec32`	unsigned	int	16	32	`dvec.h`
	`I8vec64`	unspecified	int	8	64	`dvec.h`
	`Is8vec64`	signed	int	8	64	`dvec.h`
	`Iu8vec64`	unsigned	int	8	64	`dvec.h`

Most classes contain similar functionality for all data types and are represented by all available intrinsics. However, some capabilities do not translate from one data type to another without suffering from poor performance, and are therefore excluded from individual classes.

NOTE Intrinsics that take immediate values and cannot be expressed easily in classes are not implemented. For example:

- `_mm_shuffle_ps`
- `_mm_shuffle_pi16`
- `_mm_shuffle_ps`
- `_mm_extract_pi16`
- `_mm_insert_pi16`

Access to Classes Using Header Files

The required class header files are installed in the include directory with the Intel® oneAPI DPC++/C++ Compiler. To enable the classes, use the `#include` directive in your program file as shown in the table that follows.

Include Directives for Enabling Classes

Instruction Set Extension	Include Directive
MMX™ Technology	`#include <ivec.h>`

Instruction Set Extension	Include Directive
Intel® SSE	`#include <fvec.h>`
Intel® SSE2	`#include <dvec.h>`
Intel® Streaming SIMD Extensions 3 (Intel® SSE3)	`#include <dvec.h>`
Intel® Streaming SIMD Extensions 4 (Intel® SSE4)	`#include <dvec.h>`
Intel® AVX	`#include <dvec.h>`

Each succeeding file from the top down includes the preceding class. You only need to include `fvec.h` if you want to use both the `Ivec` and `Fvec` classes. Similarly, to use all the classes including those for Intel® SSE2, you only need to include the `dvec.h` file.

Usage Precautions

When using the C++ classes, you should follow some general guidelines. More detailed usage rules for each class are listed in Integer Vector Classes, and Floating-point Vector Classes.

Clear MMX™ Technology Registers

If you use both the `Ivec` and `Fvec` classes at the same time, your program could mix MMX™ Technology instructions, called by `Ivec` classes, with Intel® architecture floating-point instructions, called by `Fvec` classes. x87 floating-point instructions exist in the following `Fvec` functions:

- `fvec` constructors
- debug functions (`cout` and element access)
- `rsqrt_nr`

NOTE MMX™ Technology registers are aliased on the floating-point registers, so you should clear the MMX™ Technology state with the EMMS instruction intrinsic before issuing an x87 floating-point instruction.

Example	Usage
`ivecA = ivecA & ivecB;`	An `Ivec` logical operation that uses MMX™ Technology instructions.
`empty ();`	Creates a clear state.
`cout << f32vec4a;`	A `F32vec4` operation that uses x87 floating-point instructions.

Caution Failure to clear the MMX™ Technology registers can result in incorrect execution or poor performance due to an incorrect register state.

Capabilities of C++ SIMD Classes

The fundamental capabilities of each C++ SIMD class include:

- Computation

- Horizontal data support
- Branch compression/elimination
- Caching hints

Understanding each of these capabilities and how they interact is crucial to achieving desired results.

Computation

The SIMD C++ classes contain vertical operator support for most arithmetic operations, including shifting and saturation.

Computation operations include: `+`, `-`, `*`, `/`, reciprocal (`rcp` and `rcp_nr`), square root (`sqrt`), and reciprocal square root (`rsqrt` and `rsqrt_nr`).

Operations `rcp` and `rsqrt` are approximating instructions with very short latencies that produce results with at least 12 bits of accuracy. You may get a different answer if used on non-Intel processors. Operations `rcp_nr` and `rsqrt_nr` use software refining techniques to enhance the accuracy of the approximations, with a minimal impact on performance. (The `nr` stands for Newton-Raphson, a mathematical technique for improving performance using an approximate result.)

Horizontal Data Support

The C++ SIMD classes provide horizontal support for some arithmetic operations. The term *horizontal* indicates computation across the elements of one vector, as opposed to the vertical, element-by-element operations on two different vectors.

The `add_horizontal`, `unpack_low` and `pack_sat` functions are examples of horizontal data support. This support enables certain algorithms that cannot exploit the full potential of SIMD instructions.

Shuffle intrinsics are another example of horizontal data flow. Shuffle intrinsics are not expressed in the C++ classes due to their immediate arguments. However, the C++ class implementation enables you to mix shuffle intrinsics with the other C++ functions. For example:

```
F32vec4 fveca, fvecb, fvecd;
fveca += fvecb;
fvecd = _mm_shuffle_ps(fveca,fvecb,0);
```

Branch Compression/Elimination

Branching in SIMD architectures can be complicated and expensive. The SIMD C++ classes provide functions to eliminate branches, using logical operations, max and min functions, conditional selects, and compares. Consider the following example:

```
short a[4], b[4], c[4];
for (i=0; i<4; i++)
c[i] = a[i] > b[i] ? a[i] : b[i];
```

This operation is independent of the value of `i`. For each `i`, the result could be either `A` or `B` depending on the actual values. A simple way of removing the branch altogether is to use the `select_gt` function, as follows:

```
Is16vec4 a, b, c
c = select_gt(a, b, a, b)
```

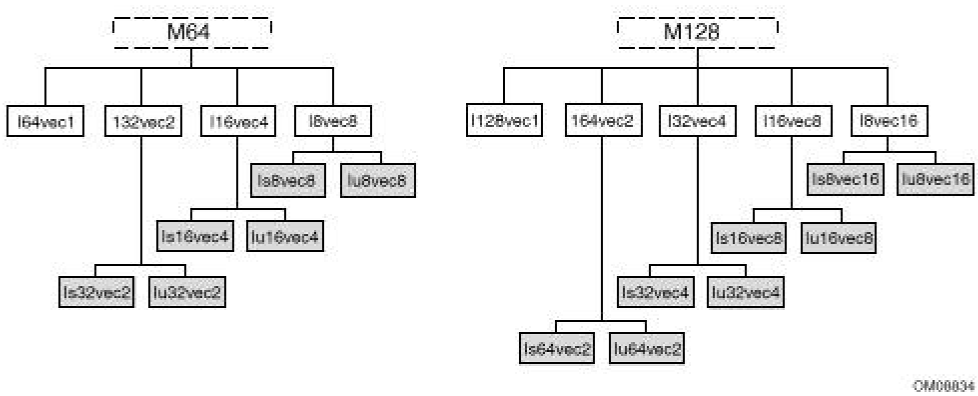
Caching Hints

Intel® Streaming SIMD Extensions provide prefetching and streaming hints. Prefetching data can minimize the effects of memory latency. Streaming hints allow you to indicate that certain data should not be cached.

Integer Vector Classes

The `Ivec` classes provide an interface to single instruction, multiple data (SIMD) processing using integer vectors of various sizes. The class hierarchy is represented in the following figure.

Ivec Class Hierarchy



The `M64` and `M128` classes define the `__m64` and `__m128i` data types from which the rest of the `Ivec` classes are derived. The first generation of child classes (the intermediate classes) are derived on element sizes of 128, 64, 32, 16, and 8 bits:

`I128vec1, I64vec1, I64vec2, I32vec2, I32vec4, I16vec4, I16vec8, I8vec8, I8vec16`

The second generation specify the signedness:

`Is64vec2, Iu64vec2, Is32vec2, Iu32vec2, Is32vec4, Iu32vec4, Is16vec4, Iu16vec4, Is16vec8, Iu16vec8, Is8vec8, Iu8vec8, Is8vec16, Iu8vec16`

> **Caution**
> Intermixing the `M64` and `M128` data types will result in unexpected behavior.

Terms and Syntax

The following are special terms and syntax used in this chapter to describe functionality of the classes with respect to their associated operations.

Ivec Class Syntax Conventions

The name of each class denotes the data type, signedness, bit size, and number of elements using the following generic format:

```
<type><signedness><bits>vec<elements>
```

```
{ F | I } { s | u } { 128 | 64 | 32 | 16 | 8 } vec { 16 | 8 | 4 | 2 | 1 }
```

where

- `type` indicates floating point (`F`) or integer (`I`).
- `signedness` indicates signed (`s`) or unsigned (`u`). For the Ivec class, leaving this field blank indicates an intermediate class. For the Fvec classes, this field is blank because there are no unsigned Fvec classes.
- `bits` specifies the number of bits per element.
- `elements` specifies the number of elements.

Special Terms and Conventions

The following terms are used to define the functionality and characteristics of the classes and operations defined in this manual.

- **Nearest Common Ancestor**: This is the intermediate or parent class of two classes of the same size. For example, the nearest common ancestor of `Iu8vec8` and `Is8vec8` is `I8vec8`, and the nearest common ancestor between `Iu8vec8` and `I16vec4` is `M64`.
- **Casting:** Changes the data type from one class to another. When an operation uses different data types as operands, the return value of the operation must be assigned to a single data type, and one or more of the data types must be converted to a required data type. This conversion is known as a typecast. While typecasting is occasionally automatic, in cases where it is not automatic you must use special syntax to explicitly typecast it yourself.
- **Operator Overloading:** This is the ability to use various operators on the user-defined data type of a given class. In the case of the Ivec and Fvec classes, once you declare a variable, you can add, subtract, multiply, and perform a range of operations. Each family of classes accepts a specified range of operators, and must comply by rules and restrictions regarding typecasting and operator overloading as defined in the header files.

Rules for Operators

To use operators with the `Ivec` classes you must use one of the following three syntax conventions, where

- [`operator`] represents an operator (for example, &, |, or ^)
- [`Ivec_Class`] represents an `Ivec` class
- `R`, `A`, `B` variables are declared using the pertinent `Ivec` classes

Convention One

Syntax:

```
[ Ivec_Class ] R = [ Ivec_Class ] A [ operator ][ Ivec_Class ] B
```

Example:

```
I64vec1 R = I64vec1 A & I64vec1 B;
```

Convention Two

Syntax:

```
[ Ivec_Class ] R =[ operator ] ([ Ivec_Class ] A,[ Ivec_Class ] B)
```

Example:

```
I64vec1 R = andnot(I64vec1 A, I64vec1 B);
```

Convention Three

Syntax:

```
[ Ivec_Class ] R [ operator ]= [ Ivec_Class ] A
```

Example:

```
I64vec1 R &= I64vec1 A;
```

Summary of Rules for Major Operators

The following table lists automatic and explicit sign and size typecasting. *Explicit* means that it is illegal to mix different types without an explicit typecasting. *Automatic* means that you can mix types freely and the compiler will do the typecasting for you.

Operators	Sign Typecasting	Size Typecasting	Other Typecasting Requirements
Assignment	N/A	N/A	N/A
Logical	Automatic	Automatic (to left)	Explicit typecasting is required for different types used in non-logical expressions on the right side of the assignment.
Addition and Subtraction	Automatic	Explicit	N/A
Multiplication	Automatic	Explicit	N/A
Shift	Automatic	Explicit	Casting Required to ensure arithmetic shift.
Compare	Automatic	Explicit	Explicit casting is required for signed classes for the less-than or greater-than operations.
Conditional Select	Automatic	Explicit	Explicit casting is required for signed classes for less-than or greater-than operations.

Data Declaration and Initialization

The following table lists literal examples of constructor declarations and data type initialization for all class sizes. All values are initialized with the most significant element on the left and the least significant to the right.

Operation	Class	Syntax
Declaration	`M128`	`I128vec1 A; Iu8vec16 A;`
Declaration	`M64`	`I64vec1 A; Iu8vec8 A;`
`__m128` Initialization	`M128`	`I128vec1 A(__m128 m); Iu16vec8(__m128 m);`
`__m64` Initialization	`M64`	`I64vec1 A(__m64 m);Iu8vec8 A(__m64 m);`
`__int64` Initialization	`M64`	`I64vec1 A = __int64 m; Iu8vec8 A =__int64 m;`
`int i` Initialization	`M64`	`I64vec1 A = int i; Iu8vec8 A = int i;`
`int` Initialization	`I32vec2`	`I32vec2 A(int A1, int A0); Is32vec2 A(signed int A1, signed int A0); Iu32vec2 A(unsigned int A1, unsigned int A0);`

Operation	Class	Syntax
`int` Initialization	`I32vec4`	`I32vec4 A(int A3, int A2, int A1, int A0);` `Is32vec4 A(signed int A3, ..., signed int A0);` `Iu32vec4 A(unsigned int A3, ..., unsigned int A0);`
`short int` Initialization	`I16vec4`	`I16vec4 A(short A3, short A2, short A1, short A0);` `Is16vec4 A(signed short A3, ..., signed short A0);` `Iu16vec4 A(unsigned short A3, ..., unsigned short A0);`
`short int` Initialization	`I16vec8`	`I16vec8 A(short A7, short A6, ..., short A1, short A0);` `Is16vec8 A(signed A7, ..., signed short A0);` `Iu16vec8 A(unsigned short A7, ..., unsigned short A0);`
`char` Initialization	`I8vec8`	`I8vec8 A(char A7, char A6, ..., char A1, char A0);` `Is8vec8 A(signed char A7, ..., signed char A0);` `Iu8vec8 A(unsigned char A7, ..., unsigned char A0);`
`char` Initialization	`I8vec16`	`I8vec16 A(char A15, ..., char A0);` `Is8vec16 A(signed char A15, ..., signed char A0);` `Iu8vec16 A(unsigned char A15, ..., unsigned char A0);`

Assignment Operator

Any `Ivec` object can be assigned to any other `Ivec` object; conversion on assignment from one `Ivec` object to another is automatic. For example:

```
Is16vec4 A;
Is8vec8 B;
I64vec1 C;

A = B; /* assign Is8vec8 to Is16vec4 */
B = C; /* assign I64vec1 to Is8vec8 */
B = A & C; /* assign M64 result of '&' to Is8vec8 */
```

Logical Operators

The logical operators use the symbols and intrinsics listed in the following table.

Bitwise Operation	Standard Operator Symbols	Operator Symbols with Assign	Standard Syntax Usage	Syntax Usage with Assign	Corresponding Intrinsic
AND	&	&=	R = A & B	R &= A	_mm_and_si64 _mm_and_si128
OR	|	|=	R = A | B	R |= A	_mm_and_si64 _mm_and_si128
XOR	^	^=	R = A^B	R ^= A	_mm_and_si64 _mm_and_si128
ANDNOT	andnot	N/A	R = A andnot B	N/A	_mm_and_si64 _mm_and_si128

Examples and Miscellaneous Exceptions

- A and B converted to M64. Result assigned to `Iu8vec8`:

```
I64vec1 A;
Is8vec8 B;
Iu8vec8 C;

C = A & B;
```

- Same size and signedness operators return the nearest common ancestor:

```
I32vec2 R = Is32vec2 A ^ Iu32vec2 B;
```

- A&B returns M64, which is cast to `Iu8vec8`:

```
C = Iu8vec8(A&B)+ C;
```

- When `A` and `B` are of the same class, they return the same type. When `A` and `B` are of different classes, the return value is the return type of the nearest common ancestor.

Ivec Logical Operator Overloading

The logical operator returns values for combinations of classes, listed in the following table, apply when `A` and `B` are of different classes.

Return Value	AND	OR	XOR	NAND	Operand A	Operand B
I64vec1 R	&	|	^	andnot	I[s|u]64vec2 A	I[s|u]64vec2 B
I64vec2 R	&	|	^	andnot	I[s|u]64vec2 A	I[s|u]64vec2 B
I32vec2 R	&	|	^	andnot	I[s|u]32vec2 A	I[s|u]32vec2 B
I32vec4 R	&	|	^	andnot	I[s|u]32vec4 A	I[s|u]32vec4 B
I16vec4 R	&	|	^	andnot	I[s|u]16vec4 A	I[s|u]16vec4 B

Return Value	AND	OR	XOR	NAND	Operand A	Operand B
I16vec8 R	&	|	^	andnot	I[s|u]16vec8 A	I[s|u]16vec8 B
I8vec8 R	&	|	^	andnot	I[s|u]8vec8 A	I[s|u]8vec8 B
I8vec16 R	&	|	^	andnot	I[s|u]8vec16 A	I[s|u]8vec16 B

Ivec Logical Operator Overloading with Assignment

For logical operators with assignment, the return value of `R` is always the same data type as the pre-declared value of R as listed in the following table:

Return Type	Left Side	AND	OR	XOR	Right Side (Any Ivec Type)
I128vec1	I128vec1 R	&=	|=	^=	I[s|u][N]vec[N] A;
I64vec1	I64vec1 R	&=	|=	^=	I[s|u][N]vec[N] A;
I64vec2	I64vec2 R	&=	|=	^=	I[s|u][N]vec[N] A;
I[x]32vec4	I[x]32vec4 R	&=	|=	^=	I[s|u][N]vec[N] A;
I[x]32vec2	I[x]32vec2 R	&=	|=	^=	I[s|u][N]vec[N] A;
I[x]16vec8	I[x]16vec8 R	&=	|=	^=	I[s|u][N]vec[N] A;
I[x]16vec4	I[x]16vec4 R	&=	|=	^=	I[s|u][N]vec[N] A;
I[x]8vec16	I[x]8vec16 R	&=	|=	^=	I[s|u][N]vec[N] A;
I[x]8vec8	I[x]8vec8 R	&=	|=	^=	I[s|u][N]vec[N] A;

Addition and Subtraction Operators

The addition and subtraction operators return the class of the nearest common ancestor when the right-side operands are of different signs. The following code snippets show examples of usage and miscellaneous exceptions.

- Return nearest common ancestor type, `I16vec4`:

```
Is16vec4 A;
Iu16vec4 B;
I16vec4 C;

C = A + B;
```

- Returns type left-hand operand type:

```
Is16vec4 A;
Iu16vec4 B;
```

```
A += B;
B -= A;
```

- Explicitly convert `B` to `Is16vec4`:

```
Is16vec4 A,C;
Iu32vec24 B;

C = A + C;
C = A + (Is16vec4)B;
```

The following table lists addition and subtraction operators with their corresponding intrinsics:

Operation	Symbols	Syntax	Corresponding Intrinsics
Addition	+ +=	R = A + B R += A	_mm_add_epi64 _mm_add_epi32 _mm_add_epi16 _mm_add_epi8 _mm_add_pi32 _mm_add_pi16 _mm_add_pi8
Subtraction	- -=	R = A - B R -= A	_mm_sub_epi64 _mm_sub_epi32 _mm_sub_epi16 _mm_sub_epi8 _mm_sub_pi32 _mm_sub_pi16 _mm_sub_pi8

Addition and Subtraction Operator Overloading

The following table lists addition and subtraction return values for combinations of classes when the right side operands are of different signedness. The two operands must be the same size, otherwise you must explicitly indicate the typecasting.

Return Value	Add	Sub	Right Side Operand (A)	Right Side Operand (B)
I64vec2 R	+	-	I[s|u]64vec2 A	I[s|u]64vec2 B
I32vec4 R	+	-	I[s|u]32vec4 A	I[s|u]32vec4 B
I32vec2 R	+	-	I[s|u]32vec2 A	I[s|u]32vec2 B
I16vec8 R	+	-	I[s|u]16vec8 A	I[s|u]16vec8 B
I16vec4 R	+	-	I[s|u]16vec4 A	I[s|u]16vec4 B
I8vec8 R	+	-	I[s|u]8vec8 A	I[s|u]8vec8 B
I8vec16 R	+	-	I[s|u]8vec2 A	I[s|u]8vec16 B

Addition and Subtraction with Assignment

The following table lists the return data type values for operands of the addition and subtraction operators with assignment. The left side operand determines the size and signedness of the return value. The right side operand must be the same size as the left operand; otherwise, you must use an explicit typecast.

Return Value	Left Side Operand	Add	Sub	Right Side Operand
`I[x]32vec4`	`I[x]32vec2 R`	`+=`	`-=`	`I[s|u]32vec4 A;`
`I[x]32vec2 R`	`I[x]32vec2 R`	`+=`	`-=`	`I[s|u]32vec2 A;`
`I[x]16vec8`	`I[x]16vec8`	`+=`	`-=`	`I[s|u]16vec8 A;`
`I[x]16vec4`	`I[x]16vec4`	`+=`	`-=`	`I[s|u]16vec4 A;`
`I[x]8vec16`	`I[x]8vec16`	`+=`	`-=`	`I[s|u]8vec16 A;`
`I[x]8vec8`	`I[x]8vec8`	`+=`	`-=`	`I[s|u]8vec8 A;`

Multiplication Operators

The multiplication operators can only accept and return data types from the `I[s|u]16vec4` or `I[s|u]16vec8` classes, as shown in the following examples:

- Explicitly convert B to `Is16vec4`:

```
Is16vec4 A,C;
Iu32vec2 B;

C = A * C;
C = A * (Is16vec4)B;
```

- Return nearest common ancestor type, `I16vec4`:

```
Is16vec4 A;
Iu16vec4 B;
I16vec4 C;

C = A + B;
```

- The `mul_high` and `mul_add` functions take `Is16vec4` data only:

```
Is16vec4 A,B,C,D;

C = mul_high(A,B);
D = mul_add(A,B);
```

Multiplication Operators with Corresponding Intrinsics

Symbols		Syntax Usage	Intrinsic
*	*=	R = A * B R *= A	`_mm_mullo_pi16` `_mm_mullo_epi16`
`mul_high`	N/A	`R = mul_high(A, B)`	`_mm_mulhi_pi16` `_mm_mulhi_epi16`
`mul_add`	N/A	`R = mul_high(A, B)`	`_mm_madd_pi16` `_mm_madd_epi16`

Multiplication Operator Overloading

The multiplication return operators always return the nearest common ancestor as listed in the following table. The two operands must be 16 bits in size, otherwise you must explicitly indicate typecasting.

R	Mul	Operand A	Operand B
`I16vec4 R`	`*`	`I[s|u]16vec4 A`	`I[s|u]16vec4 B`
`I16vec8 R`	`*`	`I[s|u]16vec8 A`	`I[s|u]16vec8 B`
`Is16vec4 R`	`mul_add`	`Is16vec4 A`	`Is16vec4 B`
`Is16vec8`	`mul_add`	`Is16vec8 A`	`Is16vec8 B`
`Is32vec2 R`	`mul_high`	`Is16vec4 A`	`Is16vec4 B`
`Is32vec4 R`	`mul_high`	`s16vec8 A`	`Is16vec8 B`

Multiplication with Assignment

The following table lists the return values and data type assignments for operands of the multiplication operators with assignment. All operands must be 16 bytes in size. If the operands are not the right size, you must use an explicit typecast.

Return Value	Left Side Operand	Mul	Right Side Operand
`I[x]16vec8`	`I[x]16vec8`	`*=`	`I[s|u]16vec8 A;`
`I[x]16vec4`	`I[x]16vec4`	`*=`	`I[s|u]16vec4 A;`

Shift Operators

The right shift argument can be any integer or Ivec value, and is implicitly converted to a M64 data type. The first or left operand of a `<<` can be of any type except `I[s|u]8vec[8|16]`. For example:

- Automatic size and sign conversion:

```
Is16vec4 A,C;
Iu32vec2 B;

C = A;
```

- A&B returns `I16vec4`, which must be cast to `Iu16vec4` to ensure logical shift, not arithmetic shift:

```
Is16vec4 A, C;
Iu16vec4 B, R;

R = (Iu16vec4)(A & B) C;
```

- A&B returns `I16vec4`, which must be cast to `Is16vec4` to ensure arithmetic shift, not logical shift:

```
R = (Is16vec4)(A & B) C;
```

Shift Operators with Corresponding Intrinsics

Operation	Symbols	Syntax Usage	Intrinsic
Shift Left	`<<` `&=`	`R = A << B` `R &= A`	`_mm_sll_si64` `_mm_slli_si64` `_mm_sll_pi32` `_mm_slli_pi32` `_mm_sll_pi16` `_mm_slli_pi16`
Shift Right	`>>`	`R = A >> B` `R >>= A`	`_mm_srl_si64` `_mm_srli_si64`

Operation	Symbols	Syntax Usage	Intrinsic
			_mm_srl_pi32 _mm_srli_pi32 _mm_srl_pi16 _mm_srli_pi16 _mm_sra_pi32 _mm_srai_pi32 _mm_sra_pi16 _mm_srai_pi16

Shift Operator Overloading

Right shift operations with signed data types use arithmetic shifts. All unsigned and intermediate classes correspond to logical shifts. The following table lists how the return type is determined by the first argument type:

Option	R	Right Shift		Left Shift		A	B
Logical	I64vec1	>>	>>=	<<	<<=	I64vec1 A;	I64vec1 B;
Logical	I32vec2	>>	>>=	<<	<<=	I32vec2 A	I32vec2 B;
Arithmetic	Is32vec2	>>	>>=	<<	<<=	Is32vec2 A	I[s|u][N]vec[N] B;
Logical	Iu32vec2	>>	>>=	<<	<<=	Iu32vec2 A	I[s|u][N]vec[N] B;
Logical	I16vec4	>>	>>=	<<	<<=	I16vec4 A	I16vec4 B
Arithmetic	Is16vec4	>>	>>=	<<	<<=	Is16vec4 A	I[s|u][N]vec[N] B;
Logical	Iu16vec4	>>	>>=	<<	<<=	Iu16vec4 A	I[s|u][N]vec[N] B;

Comparison Operators

The equality and inequality comparison operands can have mixed signedness, but they must be of the same size. The comparison operators for less-than and greater-than must be of the same sign and size. For example:

- The nearest common ancestor is returned for compare for equal/not-equal operations:

```
Iu8vec8 A;
Is8vec8 B;
I8vec8 C;

C = cmpneq(A,B);
```

- Type cast needed for different-sized elements for equal/not-equal comparisons:

```
Iu8vec8 A, C;
Is16vec4 B;

C = cmpeq(A,(Iu8vec8)B);
```

- Type cast needed for sign or size differences for less-than and greater-than comparisons:

```
Iu16vec4 A;
Is16vec4 B, C;

C = cmpge((Is16vec4)A,B);
C = cmpgt(B,C);
```

Inequality Comparison Symbols and Corresponding Intrinsics

Comparison	Operators	Syntax	Intrinsic
Equality	cmpeq	R = cmpeq(A, B)	_mm_cmpeq_pi32 _mm_cmpeq_pi16 _mm_cmpeq_pi8
Inequality	cmpneq	R = cmpneq(A, B)	_mm_cmpeq_pi32 _mm_cmpeq_pi16 _mm_cmpeq_pi8 _mm_andnot_si64
Greater Than	cmpgt	R = cmpgt(A, B)	_mm_cmpgt_pi32 _mm_cmpgt_pi16 _mm_cmpgt_pi8
Greater Than or Equal To	cmpge	R = cmpge(A, B)	_mm_cmpgt_pi32 _mm_cmpgt_pi16 _mm_cmpgt_pi8 _mm_andnot_si64
Less Than	cmplt	R = cmplt(A, B)	_mm_cmpgt_pi32 _mm_cmpgt_pi16 _mm_cmpgt_pi8
Less Than or Equal To	cmple	R = cmple(A, B)	_mm_cmpgt_pi32 _mm_cmpgt_pi16 _mm_cmpgt_pi8 _mm_andnot_si64

Compare Operator Overloading

Comparison operators have the restriction that the operands must be the size and sign as listed in the following table.

R	Comparison	Operand A	Operand B
I32vec2 R	cmpeq cmpne	I[s|u]32vec2 B	I[s|u]32vec2 B
I16vec4 R		I[s|u]16vec4 B	I[s|u]16vec4 B
I8vec8 R		I[s|u]8vec8 B	I[s|u]8vec8 B

R	Comparison	Operand A	Operand B
I32vec2 R	cmpgt cmpge cmplt cmple	Is32vec2 B	Is32vec2 B
I16vec4 R		Is16vec4 B	Is16vec4 B
I8vec8 R		Is8vec8 B	Is8vec8 B

Conditional Select Operators

For conditional select operands, the third and fourth operands determine the type returned. Third and fourth operands with same size, but different signedness, return the nearest common ancestor data type. For example:

- Return the nearest common ancestor data type if third and fourth operands are of the same size, but different signs:

```
I16vec4 R = select_neq(Is16vec4, Is16vec4, Is16vec4, Iu16vec4);
```

- Conditional select for equality:

```
R0 := (A0 == B0) ? C0 : D0;
R1 := (A1 == B1) ? C1 : D1;
R2 := (A2 == B2) ? C2 : D2;
R3 := (A3 == B3) ? C3 : D3;
```

- Conditional select for inequality:

```
R0 := (A0 != B0) ? C0 : D0;
R1 := (A1 != B1) ? C1 : D1;
R2 := (A2 != B2) ? C2 : D2;
R3 := (A3 != B3) ? C3 : D3;
```

Conditional Select Symbols and Corresponding Intrinsics

The following table lists the conditional select symbols and their corresponding intrinsics:

Conditional Select	Operators	Syntax	Corresponding Intrinsic	Additional Intrinsic (Applies to All)
Equality	select_eq	R = select_eq(A, B, C, D)	_mm_cmpeq_pi32 _mm_cmpeq_pi16 _mm_cmpeq_pi8	_mm_and_si64 _mm_or_si64 _mm_andnot_si64
Inequality	select_neq	R = select_neq(A, B, C, D)	_mm_cmpeq_pi32 _mm_cmpeq_pi16 _mm_cmpeq_pi8	
Greater Than	select_gt	R = select_gt(A, B, C, D)	_mm_cmpgt_pi32 _mm_cmpgt_pi16 _mm_cmpgt_pi8	
Greater Than or Equal To	select_ge	R = select_gt(A, B, C, D)	_mm_cmpge_pi32 _mm_cmpge_pi16 _mm_cmpge_pi8	

Conditional Select	Operators	Syntax	Corresponding Intrinsic	Additional Intrinsic (Applies to All)
Less Than	`select_lt`	`R = select_lt(A, B, C, D)`	`_mm_cmplt_pi32` `_mm_cmplt_pi16` `_mm_cmplt_pi8`	
Less Than or Equal To	`select_le`	`R = select_le(A, B, C, D)`	`_mm_cmple_pi32` `_mm_cmple_pi16` `_mm_cmple_pi8`	

Conditional Select Operator Overloading

All conditional select operands must be of the same size. The return data type is the nearest common ancestor of operands `C` and `D`. For conditional select operations using greater-than or less-than operations, the first and second operands must be signed as listed in the following table:

R	Comparison	A and B	C	D
`I32vec2 R`	`select_eq` `select_ne`	`I[s|u]32vec2`	`I[s|u]32vec2`	`I[s|u]32vec2`
`I16vec4 R`			`I[s|u]16vec4`	`I[s|u]16vec4`
`I8vec8 R`			`I[s|u]8vec8`	`I[s|u]8vec8`
`I32vec2 R`	`select_gt` `select_ge` `select_lt` `select_le`	`Is32vec2`	`Is32vec2`	`Is32vec2`
`I16vec4 R`			`Is16vec4`	`Is16vec4`
`I8vec8 R`			`Is8vec8`	`Is8vec8`

Conditional Select Operator Return Value Mapping

The following table lists the mapping of return values from `R0` to `R7` for any number of elements. The same return value mappings also apply when there are fewer than four return values.

Return Value	A Operands	Available Operators						B Operands	C and D Operands
R0:=	A0	==	!=	>	>=	<	<=	B0	C0 : D0;
R1:=	A0	==	!=	>	>=	<	<=	B0	C1 : D1;
R2:=	A0	==	!=	>	>=	<	<=	B0	C2 : D2;
R3:=	A0	==	!=	>	>=	<	<=	B0	C3 : D3;
R4:=	A0	==	!=	>	>=	<	<=	B0	C4 : D4;
R5:=	A0	==	!=	>	>=	<	<=	B0	C5 : D5;
R6:=	A0	==	!=	>	>=	<	<=	B0	C6 : D6;

Return Value	A Operands	Available Operators						B Operands	C and D Operands
R7:=	A0	==	!=	>	>=	<	<=	B0	C7 : D7;

Debug Operations

The debug operations do not map to any compiler intrinsics for MMX™ instructions. They are provided for debugging programs only. Use of these operations may result in loss of performance, so you should not use them outside of debugging.

Output Examples

- The four 32-bit values of `A` are placed in the output buffer and printed in the following format (default in decimal):

```
cout << Is32vec4 A;
cout << Iu32vec4 A;
cout << hex << Iu32vec4 A; /* print in hex format */
"[3]:A3 [2]:A2 [1]:A1 [0]:A0"
```

  Corresponding intrinsics: none
- The two 32-bit values of `A` are placed in the output buffer and printed in the following format (default in decimal):

```
cout << Is32vec2 A;
cout << Iu32vec2 A;
cout << hex << Iu32vec2 A; /* print in hex format */
"[1]:A1 [0]:A0"
```

  Corresponding intrinsics: none
- The eight 16-bit values of `A` are placed in the output buffer and printed in the following format (default in decimal):

```
cout << Is16vec8 A;
cout << Iu16vec8 A;
cout << hex << Iu16vec8 A; /* print in hex format */
"[7]:A7 [6]:A6 [5]:A5 [4]:A4 [3]:A3 [2]:A2 [1]:A1 [0]:A0"
```

  Corresponding intrinsics: none
- The four 16-bit values of `A` are placed in the output buffer and printed in the following format (default in decimal):

```
cout << Is16vec4 A;
cout << Iu16vec4 A;
cout << hex << Iu16vec4 A; /* print in hex format */
"[3]:A3 [2]:A2 [1]:A1 [0]:A0"
```

  Corresponding intrinsics: none

- The sixteen 8-bit values of A are placed in the output buffer and printed in the following format (default is decimal):

```
cout << Is8vec16 A; cout << Iu8vec16 A; cout << hex << Iu8vec8 A;
/* print in hex format instead of decimal*/
"[15]:A15 [14]:A14 [13]:A13 [12]:A12 [11]:A11 [10]:A10 [9]:A9 [8]:A8 [7]:A7 [6]:A6 [5]:A5 [4]:A4
[3]:A3 [2]:A2 [1]:A1 [0]:A0"
```

  Corresponding intrinsics: none
- The eight 8-bit values of A are placed in the output buffer and printed in the following format (default is decimal):

```
cout << Is8vec8 A; cout << Iu8vec8 A;cout << hex << Iu8vec8 A;
/* print in hex format instead of decimal*/
"[7]:A7 [6]:A6 [5]:A5 [4]:A4 [3]:A3 [2]:A2 [1]:A1 [0]:A0"
```

  Corresponding intrinsics: none

Element Access Operators

Access and read element i of A. If DEBUG is enabled and the user tries to access an element outside of A, a diagnostic message is printed and the program aborts.

Corresponding intrinsics: none

Examples:

```
int R = Is64vec2 A[i];
unsigned int R = Iu64vec2 A[i];
int R = Is32vec4 A[i];
unsigned int R = Iu32vec4 A[i];
int R = Is32vec2 A[i];
unsigned int R = Iu32vec2 A[i];
short R = Is16vec8 A[i];
unsigned short R = Iu16vec8 A[i];
short R = Is16vec4 A[i];
unsigned short R = Iu16vec4 A[i];
signed char R = Is8vec16 A[i];
unsigned char R = Iu8vec16 A[i];
signed char R = Is8vec8 A[i];
unsigned char R = Iu8vec8 A[i];
```

Element Assignment Operators

Assign R to element i of A. If DEBUG is enabled and the user tries to assign a value to an element outside of A, a diagnostic message is printed and the program aborts.

Corresponding intrinsics: none

Examples:

```
Is64vec2 A[i] = int R;
Is32vec4 A[i] = int R;
Iu32vec4 A[i] = unsigned int R;
Is32vec2 A[i] = int R;
Iu32vec2 A[i] = unsigned int R;
Is16vec8 A[i] = short R;
Iu16vec8 A[i] = unsigned short R;
```

```
Is16vec4 A[i] = short R;
Iu16vec4 A[i] = unsigned short R;
Is8vec16 A[i] = signed char R;
Iu8vec16 A[i] = unsigned char R;
Is8vec8 A[i] = signed char R;
Iu8vec8 A[i] = unsigned char R;
```

Unpack Operators

- Interleave the 64-bit value from the high half of `A` with the 64-bit value from the high half of `B`:

```
I64vec2 unpack_high(I64vec2 A, I64vec2 B);
Is64vec2 unpack_high(Is64vec2 A, Is64vec2 B);
Iu64vec2 unpack_high(Iu64vec2 A, Iu64vec2 B);
R0 = A1;
R1 = B1;
```

  Corresponding intrinsic: `_mm_unpackhi_epi64`
- Interleave the two 32-bit values from the high half of `A` with the two 32-bit values from the high half of `B`:

```
I32vec4 unpack_high(I32vec4 A, I32vec4 B);
Is32vec4 unpack_high(Is32vec4 A, Is32vec4 B);
Iu32vec4 unpack_high(Iu32vec4 A, Iu32vec4 B);
R0 = A1;
R1 = B1;
R2 = A2;
R3 = B2;
```

  Corresponding intrinsic: `_mm_unpackhi_epi32`
- Interleave the 32-bit value from the high half of `A` with the 32-bit value from the high half of `B`:

```
I32vec2 unpack_high(I32vec2 A, I32vec2 B);
Is32vec2 unpack_high(Is32vec2 A, Is32vec2 B);
Iu32vec2 unpack_high(Iu32vec2 A, Iu32vec2 B);
R0 = A1;
R1 = B1;
```

  Corresponding intrinsic: `_mm_unpackhi_pi32`
- Interleave the four 16-bit values from the high half of `A` with the two 16-bit values from the high half of `B`:

```
I16vec8 unpack_high(I16vec8 A, I16vec8 B);
Is16vec8 unpack_high(Is16vec8 A, Is16vec8 B);
Iu16vec8 unpack_high(Iu16vec8 A, Iu16vec8 B);
R0 = A2;
R1 = B2;
R2 = A3;
R3 = B3;
```

  Corresponding intrinsic: `_mm_unpackhi_epi16`
- Interleave the two 16-bit values from the high half of `A` with the two 16-bit values from the high half of `B`:

```
I16vec4 unpack_high(I16vec4 A, I16vec4 B);
Is16vec4 unpack_high(Is16vec4 A, Is16vec4 B);
Iu16vec4 unpack_high(Iu16vec4 A, Iu16vec4 B);
R0 = A2;R1 = B2;
R2 = A3;R3 = B3;
```

Corresponding intrinsic: `_mm_unpackhi_pi16`

- Interleave the four 8-bit values from the high half of `A` with the four 8-bit values from the high half of `B`:

```
I8vec8 unpack_high(I8vec8 A, I8vec8 B);
Is8vec8 unpack_high(Is8vec8 A, I8vec8 B);
Iu8vec8 unpack_high(Iu8vec8 A, I8vec8 B);
R0 = A4;
R1 = B4;
R2 = A5;
R3 = B5;
R4 = A6;
R5 = B6;
R6 = A7;
R7 = B7;
```

Corresponding intrinsic: `_mm_unpackhi_pi8`

- Interleave the sixteen 8-bit values from the high half of `A` with the four 8-bit values from the high half of `B`:

```
I8vec16 unpack_high(I8vec16 A, I8vec16 B);
Is8vec16 unpack_high(Is8vec16 A, I8vec16 B);
Iu8vec16 unpack_high(Iu8vec16 A, I8vec16 B);
R0 = A8;
R1 = B8;
R2 = A9;
R3 = B9;
R4 = A10;
R5 = B10;
R6 = A11;
R7 = B11;
R8 = A12;
R8 = B12;
R2 = A13;
R3 = B13;
R4 = A14;
R5 = B14;
R6 = A15;
R7 = B15;
```

Corresponding intrinsic: `_mm_unpackhi_epi16`

- Interleave the 32-bit value from the low half of `A` with the 32-bit value from the low half of `B`:

```
R0 = A0;
R1 = B0;
```

Corresponding intrinsic: `_mm_unpacklo_epi32`

- Interleave the 64-bit value from the low half of `A` with the 64-bit values from the low half of `B`:

```
I64vec2 unpack_low(I64vec2 A, I64vec2 B);
Is64vec2 unpack_low(Is64vec2 A, Is64vec2 B);
Iu64vec2 unpack_low(Iu64vec2 A, Iu64vec2 B);
R0 = A0;
R1 = B0;
R2 = A1;
R3 = B1;
```

Corresponding intrinsic: `_mm_unpacklo_epi32`

- Interleave the two 32-bit values from the low half of `A` with the two 32-bit values from the low half of `B`:

```
I32vec4 unpack_low(I32vec4 A, I32vec4 B);
Is32vec4 unpack_low(Is32vec4 A, Is32vec4 B);
Iu32vec4 unpack_low(Iu32vec4 A, Iu32vec4 B);
R0 = A0;
R1 = B0;
R2 = A1;
R3 = B1;
```

  Corresponding intrinsic: `_mm_unpacklo_epi32`
- Interleave the 32-bit value from the low half of `A` with the 32-bit value from the low half of `B`:

```
I32vec2 unpack_low(I32vec2 A, I32vec2 B);
Is32vec2 unpack_low(Is32vec2 A, Is32vec2 B);
Iu32vec2 unpack_low(Iu32vec2 A, Iu32vec2 B);
R0 = A0;
R1 = B0;
```

  Corresponding intrinsic: `_mm_unpacklo_pi32`
- Interleave the two 16-bit values from the low half of `A` with the two 16-bit values from the low half of `B`:

```
I16vec8 unpack_low(I16vec8 A, I16vec8 B);
Is16vec8 unpack_low(Is16vec8 A, Is16vec8 B);
Iu16vec8 unpack_low(Iu16vec8 A, Iu16vec8 B);
R0 = A0;
R1 = B0;
R2 = A1;
R3 = B1;
R4 = A2;
R5 = B2;
R6 = A3;
R7 = B3;
```

  Corresponding intrinsic: `_mm_unpacklo_epi16`
- Interleave the two 16-bit values from the low half of `A` with the two 16-bit values from the low half of `B`:

```
I16vec4 unpack_low(I16vec4 A, I16vec4 B);
Is16vec4 unpack_low(Is16vec4 A, Is16vec4 B);
Iu16vec4 unpack_low(Iu16vec4 A, Iu16vec4 B);
R0 = A0;
R1 = B0;
R2 = A1;
R3 = B1;
```

  Corresponding intrinsic: `_mm_unpacklo_pi16`
- Interleave the four 8-bit values from the high low of `A` with the four 8-bit values from the low half of `B`:

```
I8vec16 unpack_low(I8vec16 A, I8vec16 B);
Is8vec16 unpack_low(Is8vec16 A, Is8vec16 B);
Iu8vec16 unpack_low(Iu8vec16 A, Iu8vec16 B);
R0 = A0;
R1 = B0;
R2 = A1;
R3 = B1;
R4 = A2;
```

```
R5 = B2;
R6 = A3;
R7 = B3;
R8 = A4;
R9 = B4;
R10 = A5;
R11 = B5;
R12 = A6;
R13 = B6;
R14 = A7;
R15 = B7;
```

Corresponding intrinsic: `_mm_unpacklo_epi8`

- Interleave the four 8-bit values from the high low of `A` with the four 8-bit values from the low half of `B`:

```
I8vec8 unpack_low(I8vec8 A, I8vec8 B);
Is8vec8 unpack_low(Is8vec8 A, Is8vec8 B);
Iu8vec8 unpack_low(Iu8vec8 A, Iu8vec8 B);
R0 = A0;
R1 = B0;
R2 = A1;
R3 = B1;
R4 = A2;
R5 = B2;
R6 = A3;
R7 = B3;
```

Corresponding intrinsic: `_mm_unpacklo_pi8`

Pack Operators

- Pack the eight 32-bit values found in `A` and `B` into eight 16-bit values with signed saturation:

```
Is16vec8 pack_sat(Is32vec2 A,Is32vec2 B);
```

Corresponding intrinsic: `_mm_packs_epi32`

- Pack the four 32-bit values found in `A` and `B` into eight 16-bit values with signed saturation:

```
Is16vec4 pack_sat(Is32vec2 A,Is32vec2 B);
```

Corresponding intrinsic: `_mm_packs_pi32`

- Pack the sixteen 16-bit values found in `A` and `B` into sixteen 8-bit values with signed saturation:

```
Is8vec16 pack_sat(Is16vec4 A,Is16vec4 B);
```

Corresponding intrinsic: `_mm_packs_epi16`

- Pack the eight 16-bit values found in `A` and `B` into eight 8-bit values with signed saturation:

```
Is8vec8 pack_sat(Is16vec4 A,Is16vec4 B);
```

Corresponding intrinsic: `_mm_packs_pi16`

- Pack the sixteen 16-bit values found in `A` and `B` into sixteen 8-bit values with unsigned saturation:

```
Iu8vec16 packu_sat(Is16vec4 A,Is16vec4 B);
```

Corresponding intrinsic: `_mm_packus_epi16`

- Pack the eight 16-bit values found in `A` and `B` into eight 8-bit values with unsigned saturation:

```
Iu8vec8 packu_sat(Is16vec4 A,Is16vec4 B);
```

  Corresponding intrinsic: `_mm_packs_pu16`

Clear MMX™ State Operator

Empty the MMX™ registers and clear the MMX state. Read the guidelines for using the EMMS instruction intrinsic.

```
void empty(void);
```

Corresponding intrinsic: `_mm_empty`

Integer Functions for Intel® Streaming SIMD Extensions

This topic contains information about Intel® Streaming SIMD Extensions (Intel® SSE) integer functions.

> **NOTE** You must include the `fvec.h` header file.

- Compute the element-wise maximum of the respective signed integer words in `A` and `B`:

```
Is16vec4 simd_max(Is16vec4 A, Is16vec4 B);
```

  The corresponding intrinsic is: `_mm_max_pi16`
- Compute the element-wise minimum of the respective signed integer words in `A` and `B`:

```
Is16vec4 simd_min(Is16vec4 A, Is16vec4 B);
```

  The corresponding intrinsic is: `_mm_min_pi16`
- Compute the element-wise maximum of the respective unsigned bytes in `A` and `B`:

```
Iu8vec8 simd_max(Iu8vec8 A, Iu8vec8 B);
```

  The corresponding intrinsic is: `_mm_max_pu8`
- Compute the element-wise minimum of the respective unsigned bytes in `A` and `B`:

```
Iu8vec8 simd_min(Iu8vec8 A, Iu8vec8 B);
```

  The corresponding intrinsic is: `_mm_min_pu8`
- Create an 8-bit mask from the most significant bits of the bytes in `A`:

```
int move_mask(I8vec8 A);
```

  The corresponding intrinsic is: `_mm_movemask_pi8`
- Conditionally store byte elements of `A` to address `p`. The high bit of each byte in the selector `B` determines whether the corresponding byte in `A` will be stored:

```
void mask_move(I8vec8 A, I8vec8 B, signed char *p);
```

  The corresponding intrinsic is: `_mm_maskmove_si64`
- Store the data in `A` to the address `p` without polluting the caches. `A` can be any `Ivec` type:

```
void store_nta(__m64 *p, M64 A);
```

  The corresponding intrinsic is: `_mm_stream_pi`
- Compute the element-wise average of the respective unsigned 8-bit integers in `A` and `B`:

```
Iu8vec8 simd_avg(Iu8vec8 A, Iu8vec8 B);
```

  The corresponding intrinsic is: `_mm_avg_pu8`
- Compute the element-wise average of the respective unsigned 16-bit integers in `A` and `B`:

```
Iu16vec4 simd_avg(Iu16vec4 A, Iu16vec4 B)
```

The corresponding intrinsic is: `_mm_avg_pu16`

Conversions between Fvec and Ivec

- Convert the lower double-precision floating-point value of `A` to a 32-bit integer with truncation:

```
int F64vec2ToInt(F64vec42 A);
r := (int)A0;
```

- Convert the four floating-point values of `A` to two the two least significant double-precision floating-point values:

```
F64vec2 F32vec4ToF64vec2(F32vec4 A);
r0 := (double)A0;
r1 := (double)A1;
```

- Convert the two double-precision floating-point values of `A` to two single-precision floating-point values:

```
F32vec4 F64vec2ToF32vec4(F64vec2 A);
r0 := (float)A0;
r1 := (float)A1;
```

- Convert the signed `int` in `B` to a double-precision floating-point value and pass the upper double-precision value from `A` through to the result:

```
F64vec2 InttoF64vec2(F64vec2 A, int B);
r0 := (double)B;
r1 := A1;
```

- Convert the lower floating-point value of `A` to a 32-bit integer with truncation:

```
int F32vec4ToInt(F32vec4 A);
r := (int)A0;
```

- Convert the two lower floating-point values of `A` to two 32-bit integer with truncation, returning the integers in packed form:

```
Is32vec2 F32vec4ToIs32vec2 (F32vec4 A);
r0 := (int)A0;
r1 := (int)A1;
```

- Convert the 32-bit integer value `B` to a floating-point value; the upper three floating-point values are passed through from `A`:

```
F32vec4 IntToF32vec4(F32vec4 A, int B);
r0 := (float)B;
r1 := A1;
r2 := A2;
r3 := A3;
```

- Convert the two 32-bit integer values in packed form in `B` to two floating-point values; the upper two floating-point values are passed through from `A`:

```
F32vec4 Is32vec2ToF32vec4(F32vec4 A, Is32vec2 B);
r0 := (float)B0;
r1 := (float)B1;
r2 := A2;
r3 := A3;
```

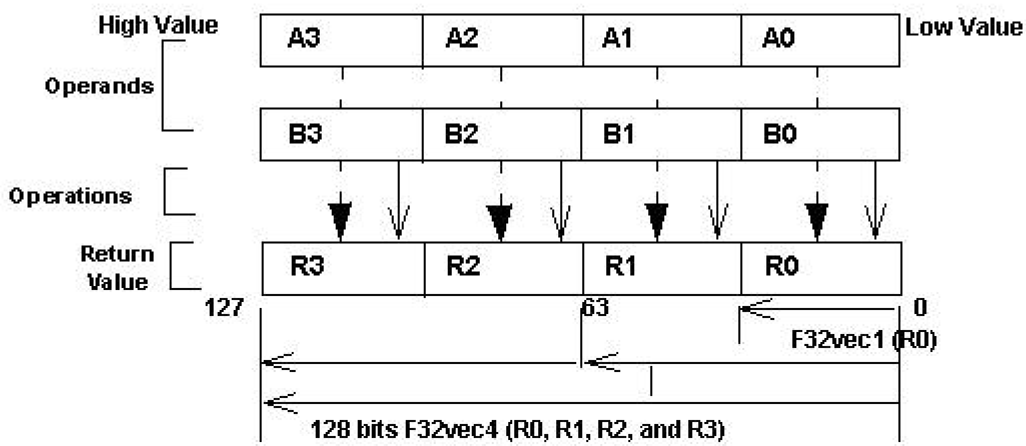
Floating-point Vector Classes

The floating-point vector classes, `F64vec2`, `F32vec4`, and `F32vec1`, provide an interface to SIMD operations. The class specifications are as follows:

- `F64vec2 A(double x, double y);`
- `F32vec4 A(float z, float y, float x, float w);`
- `F32vec1 B(float w);`

The packed floating-point input values are represented with the right-most value lowest as shown in the following table.

Single-Precision Floating-point Elements



F32vec4 returns **four** packed **single-precision floating point** values (**R0, R1, R2, and R3**).
F32vec2 returns **one single-precision floating point** value (**R0**).

Fvec Syntax and Notation

This reference uses the following conventions for syntax and return values.

Fvec Classes Syntax Notation

Fvec classes use one of the following the syntax conventions, where

- `[operator]` is an operator (for example, `&`, `|`, or `^`)
- `[Fvec_Class]` is any `Fvec` class (`F64vec2`, `F32vec4`, or `F32vec1`)
- `R`, `A`, `B` are declared `Fvec` variables of the type indicated

Syntax Convention One

Syntax:

```
[Fvec_Class] R = [Fvec_Class] A [operator][Ivec_Class] B;
```

Example:

```
F64vec2 R = F64vec2 A & F64vec2 B;
```

Syntax Convention Two

Syntax:

```
[Fvec_Class] R = [operator]([Fvec_Class] A,[Fvec_Class] B);
```

Example:

```
F64vec2 R = andnot(F64vec2 A, F64vec2 B);
```

Syntax Convention Three

Syntax:

```
[Fvec_Class] R [operator]= [Fvec_Class] A;
```

Example:

```
F64vec2 R &= F64vec2 A;
```

Return Value Notation

Because the `Fvec` classes have packed elements, the return values typically follow the conventions presented in the following table. `F32vec4` returns four single-precision, floating-point values (R0, R1, R2, and R3); `F64vec2` returns two double-precision, floating-point values, and `F32vec1` returns the lowest single-precision floating-point value (R0).

Syntax Convention One Example	Syntax Convention Two Example	Syntax Convention Three Example	F32vec 4	F64vec 2	F32vec 1
`R0 := A0 & B0;`	`R0 := A0 andnot B0;`	`R0 &= A0;`	x	x	x
`R1 := A1 & B1;`	`R1 := A1 andnot B1;`	`R1 &= A1;`	x	x	N/A
`R2 := A2 & B2;`	`R2 := A2 andnot B2;`	`R2 &= A2;`	x	N/A	N/A
`R3 := A3 & B3`	`R3 := A3 andhot B3;`	`R3 &= A3;`	x	N/A	N/A

Data Alignment

Memory operations using the Intel® Streaming SIMD Extensions should be performed on 16-byte-aligned data whenever possible. Memory operations using the Intel® Advanced Vector Extensions should be performed on 32-byte-aligned data whenever possible.

`F32vec4` and `F64vec2` object variables are properly aligned by default. Note that floating point arrays are not automatically aligned. To get 16-byte alignment, you can use the alignment `__declspec`:

```
__declspec( align(16) ) float A[4];
```

Conversions

All `Fvec` object variables can be implicitly converted to `__m128` data types. For example, the results of computations performed on `F32vec4` or `F32vec1` object variables can be assigned to `__m128` data types:

```
__m128d mm = A & B; /* where A,B are F64vec2 object variables */
__m128 mm = A & B; /* where A,B are F32vec4 object variables */
__m128 mm = A & B; /* where A,B are F32vec1 object variables */
```

Constructors and Initialization

The following tables show how to create and initialize F32vec objects with the Fvec classes.

Constructor Declaration

Example	Intrinsic	Returns
`F64vec2 A;` `F32vec4 B;` `F32vec1 C;`	N/A	N/A

__m128 Object Initialization

Example	Intrinsic	Returns
`F64vec2 A(__m128d mm);` `F32vec4 B(__m128 mm);` `F32vec1 C(__m128 mm);`	N/A	N/A

Double Initialization

Example	Intrinsic	Returns
`/* Initializes two doubles. */` `F64vec2 A(double d0, double d1);` `F64vec2 A = F64vec2(double d0, double d1);`	`_mm_set_pd`	A0 := d0; A1 := d1;
`F64vec2 A(double d0);` `/* Initializes both return values` `with the same double precision value */.`	`_mm_set1_pd`	A0 := d0; A1 := d0;

Float Initialization

Example	Intrinsic	Returns
`F32vec4 A(float f3, float f2,` `float f1, float f0);` `F32vec4 A = F32vec4(float f3, float f2,` `float f1, float f0);`	`_mm_set_ps`	A0 := f0; A1 := f1; A2 := f2; A3 := f3;
`F32vec4 A(float f0);` `/* Initializes all return values` `with the same floating point value. */`	`_mm_set1_ps`	A0 := f0; A1 := f0; A2 := f0; A3 := f0;
`F32vec4 A(double d0);` `/* Initialize all return values with` `the same double-precision value. */`	`_mm_set1_ps(d)`	A0 := d0; A1 := d0; A2 := d0; A3 := d0;
`F32vec1 A(double d0);` `/* Initializes the lowest value of A`	`_mm_set_ss(d)`	A0 := d0; A1 := 0; A2 := 0; A3 := 0;

Example	Intrinsic	Returns
with d0 and the other values with 0.*/		
F32vec1 B(float f0); /* Initializes the lowest value of B with f0 and the other values with 0.*/	_mm_set_ss	B0 := f0; B1 := 0; B2 := 0; B3 := 0;
F32vec1 B(int I); /* Initializes the lowest value of B with f0, other values are undefined.*/	_mm_cvtsi32_ss	B0 := f0; B1 := {} B2 := {} B3 := {}

Arithmetic Operators

The following table lists the arithmetic operators of the `Fvec` classes and generic syntax. The operators have been divided into standard and advanced operations, which are described in more detail later in this section.

Standard Arithmetic Operators

Operation	Operators	Generic Syntax
Addition	+ +=	R = A + B; R += A;
Subtraction	- -=	R = A - B; R -= A;
Multiplication	* *=	R = A * B; R *= A;
Division	/ /=	R = A / B; R /= A;

Advanced Arithmetic Operators

Operation	Operators	Generic Syntax
Square Root	sqrt	R = sqrt(A);
Reciprocal (Newton-Raphson)	rcp rcp_nr	R = rcp(A); R = rcp_nr(A);
Reciprocal Square Root (Newton-Raphson)	rsqrt rsqrt_nr	R = rsqrt(A); R = rsqrt_nr(A);

Standard Arithmetic Operator Usage

The following two tables show the return values for each class of the standard arithmetic operators, which use the syntax styles described earlier in the Return Value Notation section.

Standard Arithmetic Return Value Mapping

R	A	Operators				B	F32vec 4	F64vec 2	F32vec 1
R0:=	A0	+	-	*	/	B0	X	X	X
R1:=	A1	+	-	*	/	B1	X	X	N/A
R2:=	A2	+	-	*	/	B2	X	N/A	N/A
R3:=	A3	+	-	*	/	B3	X	N/A	N/A

Arithmetic with Assignment Return Value Mapping

R	Operators				A	F32vec4	F64vec2	F32vec1
R0:=	+=	-=	*=	/=	A0	X	X	X
R1:=	+=	-=	*=	/=	A1	X	X	N/A
R2:=	+=	-=	*=	/=	A2	X	N/A	N/A
R3:=	+=	-=	*=	/=	A3	X	N/A	N/A

Standard Arithmetic Operations for Fvec Classes

This table lists standard arithmetic operator syntax and intrinsics.

Operation	Returns	Example Syntax Usage	Intrinsic
Addition	4 floats	`F32vec4 R = F32vec4 A + F32vec4 B;` `F32vec4 R += F32vec4 A;`	`_mm_add_ps`
	2 doubles	`F64vec2 R = F64vec2 A + F32vec2 B;` `F64vec2 R += F64vec2 A;`	`_mm_add_pd`
	1 float	`F32vec1 R = F32vec1 A + F32vec1 B;` `F32vec1 R += F32vec1 A;`	`_mm_add_ss`
Subtraction	4 floats	`F32vec4 R = F32vec4 A - F32vec4 B;` `F32vec4 R -= F32vec4 A;`	`_mm_sub_ps`
	2 doubles	`F64vec2 R - F64vec2 A + F32vec2 B;` `F64vec2 R -= F64vec2 A;`	`_mm_sub_pd`

Operation	Returns	Example Syntax Usage	Intrinsic
	1 float	`F32vec1 R = F32vec1 A - F32vec1 B;` `F32vec1 R -= F32vec1 A;`	`_mm_sub_ss`
Multiplication	4 floats	`F32vec4 R = F32vec4 A * F32vec4 B;` `F32vec4 R *= F32vec4 A;`	`_mm_mul_ps`
	2 doubles	`F64vec2 R = F64vec2 A * F364vec2 B;` `F64vec2 R *= F64vec2 A;`	`_mm_mul_pd`
	1 float	`F32vec1 R = F32vec1 A * F32vec1 B;` `F32vec1 R *= F32vec1 A;`	`_mm_mul_ss`
Division	4 floats	`F32vec4 R = F32vec4 A / F32vec4 B;` `F32vec4 R /= F32vec4 A;`	`_mm_div_ps`
	2 doubles	`F64vec2 R = F64vec2 A / F64vec2 B;` `F64vec2 R /= F64vec2 A;`	`_mm_div_pd`
	1 float	`F32vec1 R = F32vec1 A / F32vec1 B;` `F32vec1 R /= F32vec1 A;`	`_mm_div_ss`

Advanced Arithmetic Operator Usage

Advanced Arithmetic Return Value Mapping

The following table shows the return values classes of the advanced arithmetic operators, which use the syntax styles described earlier in the Return Value Notation section.

R	Operators					A	F32vec 4	F64vec 2	F32vec 1
R0:=	`sqrt`	`rcp`	`rsqrt`	`rcp_nr`	`rsqrt_nr`	A0	X	X	X
R1:=	`sqrt`	`rcp`	`rsqrt`	`rcp_nr`	`rsqrt_nr`	A1	X	X	N/A
R2:=	`sqrt`	`rcp`	`rsqrt`	`rcp_nr`	`rsqrt_nr`	A2	X	N/A	N/A

R	Operators	A	F32vec4	F64vec2	F32vec1
R3:=	`sqrt` `rcp` `rsqrt` `rcp_nr` `rsqrt_nr`	A3	X	N/A	N/A
`f :=`	add_horizontal `(A0 + A1 + A2 + A3)`		X	N/A	N/A
d :=	add_horizontal `(A0 + A1)`		N/A	X	N/A

Advanced Arithmetic Operations for Fvec Classes

The following table show examples for advanced arithmetic operators.

Operation	Returns	Example Syntax Usage	Intrinsic
Square Root	4 floats	`F32vec4 R = sqrt(F32vec4 A);`	`_mm_sqrt_ps`
	2 doubles	`F64vec2 R = sqrt(F64vec2 A);`	`_mm_sqrt_pd`
	1 float	`F32vec1 R = sqrt(F32vec1 A);`	`_mm_sqrt_ss`
Reciprocal	4 floats	`F32vec4 R = rcp(F32vec4 A);`	`_mm_rcp_ps`
	2 doubles	`F64vec2 R = rcp(F64vec2 A);`	`_mm_rcp_pd`
	1 float	`F32vec1 R = rcp(F32vec1 A);`	`_mm_rcp_ss`
Reciprocal Square Root	4 floats	`F32vec4 R = rsqrt(F32vec4 A);`	`_mm_rsqrt_ps`
	2 doubles	`F64vec2 R = rsqrt(F64vec2 A);`	`_mm_rsqrt_pd`
	1 float	`F32vec1 R = rsqrt(F32vec1 A);`	`_mm_rsqrt_ss`
Reciprocal Newton Raphson	4 floats	`F32vec4 R = rcp_nr(F32vec4 A);`	`_mm_sub_ps` `_mm_add_ps` `_mm_mul_ps` `_mm_rcp_ps`
	2 doubles	`F64vec2 R = rcp_nr(F64vec2 A);`	`_mm_sub_pd` `_mm_add_pd` `_mm_mul_pd` `_mm_rcp_pd`

Operation	Returns	Example Syntax Usage	Intrinsic
	1 float	`F32vec1 R = rcp_nr(F32vec1 A);`	`_mm_sub_ss` `_mm_add_ss` `_mm_mul_ss` `_mm_rcp_ss`
Reciprocal Square Root Newton Raphson	4 float	`F32vec4 R = rsqrt_nr(F32vec4 A);`	`_mm_sub_pd` `_mm_mul_pd` `_mm_rsqrt_ps`
	2 doubles	`F64vec2 R = rsqrt_nr(F64vec2 A);`	`_mm_sub_pd` `_mm_mul_pd` `_mm_rsqrt_pd`
	1 float	`F32vec1 R = rsqrt_nr(F32vec1 A);`	`_mm_sub_ss` `_mm_mul_ss` `_mm_rsqrt_ss`
Horizontal Add	1 float	`float f = add_horizontal(F32vec4 A);`	`_mm_add_ss` `_mm_shuffle_ss`
	1 double	`double d = add_horizontal(F64vec2 A);`	`_mm_add_sd` `_mm_shuffle_sd`

Minimum and Maximum Operators

- Compute the minimums of the two double precision floating-point values of `A` and `B`.

```
F64vec2 R = simd_min(F64vec2 A, F64vec2 B)
R0 := min(A0,B0);
R1 := min(A1,B1);
```

  Corresponding intrinsic: `_mm_min_pd`
- Compute the minimums of the four single precision floating-point values of `A` and `B`.

```
F32vec4 R = simd_min(F32vec4 A, F32vec4 B)
R0 := min(A0,B0);
R1 := min(A1,B1);
R2 := min(A2,B2);
R3 := min(A3,B3);
```

  Corresponding intrinsic: `_mm_min_ps`
- Compute the minimum of the lowest single precision floating-point values of `A` and `B`.

```
F32vec1 R = simd_min(F32vec1 A, F32vec1 B)
R0 := min(A0,B0);
```

  Corresponding intrinsic: `_mm_min_ss`

- Compute the maximums of the two double precision floating-point values of A and B.

```
F64vec2 simd_max(F64vec2 A, F64vec2 B)
R0 := max(A0,B0);
R1 := max(A1,B1);
```

  Corresponding intrinsic: _mm_max_pd
- Compute the maximums of the four single precision floating-point values of A and B.

```
F32vec4 R = simd_man(F32vec4 A, F32vec4 B)
R0 := max(A0,B0);
R1 := max(A1,B1);
R2 := max(A2,B2);
R3 := max(A3,B3);
```

  Corresponding intrinsic: _mm_max_ps
- Compute the maximum of the lowest single precision floating-point values of A and B.

```
F32vec1 simd_max(F32vec1 A, F32vec1 B)
R0 := max(A0,B0);
```

  Corresponding intrinsic: _mm_max_ss

Logical Operators

The following table lists the logical operators of the Fvec classes and generic syntax. The logical operators for F32vec1 classes use only the lower 32 bits.

Bitwise Operation	Operators	Generic Syntax
AND	& &=	R = A & B; R &= A;
OR	| |=	R = A | B; R |= A;
XOR	^ ^=	R = A ^ B; R ^= A;
andnot	andnot	R = andnot(A);

The following table lists standard logical operators syntax and corresponding intrinsics. Note that there is no corresponding scalar intrinsic for the F32vec1 classes, which accesses the lower 32 bits of the packed vector intrinsics.

Operation	Returns	Example Syntax Usage	Intrinsic
AND	4 floats	F32vec4 & = F32vec4 A & F32vec4 B; F32vec4 & &= F32vec4 A;	_mm_and_ps
	2 doubles	F64vec2 R = F64vec2 A & F64vec2 B; F64vec2 R &= F64vec2 A;	_mm_and_pd

Operation	Returns	Example Syntax Usage	Intrinsic
	1 float	F32vec1 R = F32vec1 A & F32vec1 B; F32vec1 R &= F32vec1 A;	_mm_and_ps
OR	4 floats	F32vec4 R = F32vec4 A | F32vec4 B; F32vec4 R |= F32vec4 A;	_mm_or_ps
	2 doubles	F64vec2 R = F64vec2 A | F64vec2 B; F64vec2 R |= F64vec2 A;	_mm_or_pd
	1 float	F32vec1 R = F32vec1 A | F32vec1 B; F32vec1 R |= F32vec1 A;	_mm_or_ps
XOR	4 floats	F32vec4 R = F32vec4 A ^ F32vec4 B; F32vec4 R ^= F32vec4 A;	_mm_xor_ps
	2 doubles	F64vec2 R = F64vec2 A ^ F64vec2 B; F64vec2 R ^= F64vec2 A;	_mm_xor_pd
	1 float	F32vec1 R = F32vec1 A ^ F32vec1 B; F32vec1 R ^= F32vec1 A;	_mm_xor_ps
ANDNOT	2 doubles	F64vec2 R = andnot(F64vec2 A, F64vec2 B);	_mm_andnot_pd

Compare Operators

The operators described in this section compare the single precision floating-point values of A and B. Comparison between objects of any Fvec class return the same class being compared.

The following table lists the compare operators for the Fvec classes:

Comparison	Operators	Syntax
Equality	cmpeq	R = cmpeq(A, B)
Inequality	cmpneq	R = cmpneq(A, B)
Greater Than	cmpgt	R = cmpgt(A, B)
Greater Than or Equal To	cmpge	R = cmpge(A, B)
Not Greater Than	cmpngt	R = cmpngt(A, B)

Comparison	Operators	Syntax
Not Greater Than or Equal To	`cmpnge`	`R = cmpnge(A, B)`
Less Than	`cmplt`	`R = cmplt(A, B)`
Less Than or Equal To	`cmple`	`R = cmple(A, B)`
Not Less Than	`cmpnlt`	`R = cmpnlt(A, B)`
Not Less Than or Equal To	`cmpnle`	`R = cmpnle(A, B)`

Compare Operators

The mask is set to `0xffffffff` for each floating-point value where the comparison is true and `0x00000000` where the comparison is false. The following table shows the return values for each class of the compare operators, which use the syntax described earlier in the Return Value Notation section:

R	A0	For Any Operators	B	If True	If False	F32vec4	F64vec2	F32vec1
R0 :=	(A1 !(A1	cmp[eq | lt | le | gt | ge] cmp[ne | nlt | nle | ngt | nge]	B1) B1)	0xffffffff	0x0000000	X	X	X
R1 :=	(A1 !(A1	cmp[eq | lt | le | gt | ge] cmp[ne | nlt | nle | ngt | nge]	B2) B2)	0xffffffff	0x0000000	X	X	N/A
R2 :=	(A1 !(A1	cmp[eq | lt | le | gt | ge] cmp[ne | nlt | nle | ngt | nge]	B3) B3)	0xffffffff	0x0000000	X	N/A	N/A
R3 :=	A3	cmp[eq | lt | le | gt | ge] cmp[ne | nlt | nle | ngt | nge]	B3) B3)	0xffffffff	0x0000000	X	N/A	N/A

The following table shows examples for comparison operators and intrinsics:

Comparison	Returns	Example Syntax Usage	Intrinsic
Equality	4 floats	`F32vec4 R = cmpeq(F32vec4 A);`	`_mm_cmpeq_ps`
	2 doubles	`F64vec2 R = cmpeq(F64vec2 A);`	`_mm_cmpeq_pd`
	1 float	`F32vec1 R = cmpeq(F32vec1 A);`	`_mm_cmpeq_ss`
Inequality	4 floats	`F32vec4 R = cmpneq(F32vec4 A);`	`_mm_cmpneq_ps`

Comparison	Returns	Example Syntax Usage	Intrinsic
	2 doubles	`F64vec2 R = cmpneq(F64vec2 A);`	`_mm_cmpneq_pd`
	1 float	`F32vec1 R = cmpneq(F32vec1 A);`	`_mm_cmpneq_ss`
Greater Than	4 floats	`F32vec4 R = cmpgt(F32vec4 A);`	`_mm_cmpgt_ps`
	2 doubles	`F64vec2 R = cmpgt(F32vec42 A);`	`_mm_cmpgt_pd`
	1 float	`F32vec1 R = cmpgt(F32vec1 A);`	`_mm_cmpgt_ss`
Greater Than or Equal To	4 floats	`F32vec4 R = cmpge(F32vec4 A);`	`_mm_cmpge_ps`
	2 doubles	`F64vec2 R = cmpge(F64vec2 A);`	`_mm_cmpge_pd`
	1 float	`F32vec1 R = cmpge(F32vec1 A);`	`_mm_cmpge_ss`
Not Greater Than	4 floats	`F32vec4 R = cmpngt(F32vec4 A);`	`_mm_cmpngt_ps`
	2 doubles	`F64vec2 R = cmpngt(F64vec2 A);`	`_mm_cmpngt_pd`
	1 float	`F32vec1 R = cmpngt(F32vec1 A);`	`_mm_cmpngt_ss`
Not Greater Than or Equal To	4 floats	`F32vec4 R = cmpnge(F32vec4 A);`	`_mm_cmpnge_ps`
	2 doubles	`F64vec2 R = cmpnge(F64vec2 A);`	`_mm_cmpnge_pd`
	1 float	`F32vec1 R = cmpnge(F32vec1 A);`	`_mm_cmpnge_ss`
Less Than	4 floats	`F32vec4 R = cmplt(F32vec4 A);`	`_mm_cmplt_ps`
	2 doubles	`F64vec2 R = cmplt(F64vec2 A);`	`_mm_cmplt_pd`
	1 float	`F32vec1 R = cmplt(F32vec1 A);`	`_mm_cmplt_ss`
Less Than or Equal To	4 floats	`F32vec4 R = cmple(F32vec4 A);`	`_mm_cmple_ps`
	2 doubles	`F64vec2 R = cmple(F64vec2 A);`	`_mm_cmple_pd`
	1 float	`F32vec1 R = cmple(F32vec1 A);`	`_mm_cmple_pd`
Not Less Than	4 floats	`F32vec4 R = cmpnlt(F32vec4 A);`	`_mm_cmpnlt_ps`
	2 doubles	`F64vec2 R = cmpnlt(F64vec2 A);`	`_mm_cmpnlt_pd`

Comparison	Returns	Example Syntax Usage	Intrinsic
	1 float	`F32vec1 R = cmpnlt(F32vec1 A);`	`_mm_cmpnlt_ss`
Not Less Than or Equal To	4 floats	`F32vec4 R = cmpnle(F32vec4 A);`	`_mm_cmpnle_ps`
	2 doubles	`F64vec2 R = cmpnle(F64vec2 A);`	`_mm_cmpnle_pd`
	1 float	`F32vec1 R = cmpnle(F32vec1 A);`	`_mm_cmpnle_ss`

Conditional Select Operators for Fvec Classes

Each conditional function compares single-precision floating-point values of A and B. The C and D parameters are used for return value. Comparison between objects of any Fvec class returns the same class.

Conditional Select Operators for Fvec Classes

Conditional Select	Operators	Syntax
Equality	`select_eq`	`R = select_eq(A, B)`
Inequality	`select_neq`	`R = select_neq(A, B)`
Greater Than	`select_gt`	`R = select_gt(A, B)`
Greater Than or Equal To	`select_ge`	`R = select_ge(A, B)`
Not Greater Than	`select_gt`	`R = select_gt(A, B)`
Not Greater Than or Equal To	`select_ge`	`R = select_ge(A, B)`
Less Than	`select_lt`	`R = select_lt(A, B)`
Less Than or Equal To	`select_le`	`R = select_le(A, B)`
Not Less Than	`select_nlt`	`R = select_nlt(A, B)`
Not Less Than or Equal To	`select_nle`	`R = select_nle(A, B)`

Conditional Select Operator Usage

For conditional select operators, the return value is stored in C if the comparison is true or in D if false. The following table shows the return value mapping for each class of the conditional select operators, using the Return Value Notation.

R	A0	Operators	B	C	D	F32vec4	F64vec2	F32vec1
`R0:=`	`(A1` `!(A1`	`select_[eq | lt | le | gt | ge]` `select_[ne | nlt | nle | ngt | nge]`	`B0)` `B0)`	`C0` `C0`	`D0` `D0`	X	X	X

R	A0	Operators	B	C	D	F32vec4	F64vec2	F32vec1
R1:=	(A2 !(A2	select_[eq | lt | le | gt | ge] select_[ne | nlt | nle | ngt | nge]	B1) B1)	C1 C1	D1 D1	X	X	N/A
R2:=	(A2 !(A2	select_[eq | lt | le | gt | ge] select_[ne | nlt | nle | ngt | nge]	B2) B2)	C2 C2	D2 D2	X	N/A	N/A
R3:=	(A3 !(A3	select_[eq | lt | le | gt | ge] select_[ne | nlt | nle | ngt | nge]	B3) B3)	C3 C3	D3 D3	X	N/A	N/A

The following table shows examples for conditional select operations and corresponding intrinsics:

Comparison	Returns	Example Syntax Usage	Intrinsic
Equality	4 floats	F32vec4 R = select_eq(F32vec4 A);	_mm_cmpeq_ps
	2 doubles	F64vec2 R = select_eq(F64vec2 A);	_mm_cmpeq_pd
	1 float	F32vec1 R = select_eq(F32vec1 A);	_mm_cmpeq_ss
Inequality	4 floats	F32vec4 R = select_neq(F32vec4 A);	_mm_cmpneq_ps
	2 doubles	F64vec2 R = select_neq(F64vec2 A);	_mm_cmpneq_pd
	1 float	F32vec1 R = select_neq(F32vec1 A);	_mm_cmpneq_ss
Greater Than	4 floats	F32vec4 R = select_gt(F32vec4 A);	_mm_cmpgt_ps
	2 doubles	F64vec2 R = select_gt(F64vec2 A);	_mm_cmpgt_pd

Comparison	Returns	Example Syntax Usage	Intrinsic
	1 float	`F32vec1 R = select_gt(F32vec1 A);`	`_mm_cmpgt_ss`
Greater Than or Equal To	4 floats	`F32vec1 R = select_ge(F32vec4 A);`	`_mm_cmpge_ps`
	2 doubles	`F64vec2 R = select_ge(F64vec2 A);`	`_mm_cmpge_pd`
	1 float	`F32vec1 R = select_ge(F32vec1 A);`	`_mm_cmpge_ss`
Not Greater Than	4 floats	`F32vec1 R = select_ngt(F32vec4 A);`	`_mm_cmpngt_ps`
	2 doubles	`F64vec2 R = select_ngt(F64vec2 A);`	`_mm_cmpngt_pd`
	1 float	`F32vec1 R = select_ngt(F32vec1 A);`	`_mm_cmpngt_ss`
Not Greater Than or Equal To	4 floats	`F32vec1 R = select_nge(F32vec4 A);`	`_mm_cmpnge_ps`
	2 doubles	`F64vec2 R = select_nge(F64vec2 A);`	`_mm_cmpnge_pd`
	1 float	`F32vec1 R = select_nge(F32vec1 A);`	`_mm_cmpnge_ss`
Less Than	4 floats	`F32vec4 R = select_lt(F32vec4 A);`	`_mm_cmplt_ps`
	2 doubles	`F64vec2 R = select_lt(F64vec2 A);`	`_mm_cmplt_pd`
	1 float	`F32vec1 R = select_lt(F32vec1 A);`	`_mm_cmplt_ss`

Comparison	Returns	Example Syntax Usage	Intrinsic
Less Than or Equal To	4 floats	F32vec4 R = select_le(F32vec4 A);	_mm_cmple_ps
	2 doubles	F64vec2 R = select_le(F64vec2 A);	_mm_cmple_pd
	1 float	F32vec1 R = select_le(F32vec1 A);	_mm_cmple_ps
Not Less Than	4 floats	F32vec1 R = select_nlt(F32vec4 A);	_mm_cmpnlt_ps
	2 doubles	F64vec2 R = select_nlt(F64vec2 A);	_mm_cmpnlt_pd
	1 float	F32vec1 R = select_nlt(F32vec1 A);	_mm_cmpnlt_ss
Not Less Than or Equal To	4 floats	F32vec1 R = select_nle(F32vec4 A);	_mm_cmpnle_ps
	2 doubles	F64vec2 R = select_nle(F64vec2 A);	_mm_cmpnle_pd
	1 float	F32vec1 R = select_nle(F32vec1 A);	_mm_cmpnle_ss

Cacheability Support Operators

- Stores (non-temporal) the two double-precision, floating-point values of `A`. Requires a 16-byte aligned address.

```
void store_nta(double *p, F64vec2 A);
```

  Corresponding intrinsic: `_mm_stream_pd`
- Stores (non-temporal) the four single-precision, floating-point values of `A`. Requires a 16-byte aligned address.

```
void store_nta(float *p, F32vec4 A);
```

  Corresponding intrinsic: `_mm_stream_ps`

Debug Operations

The debug operations do not map to any compiler intrinsics for MMX™ technology or Intel® Streaming SIMD Extensions . They are provided for debugging programs only. Use of these operations may result in loss of performance, so you should not use them outside of debugging.

Output Operations

- The two single, double-precision floating-point values of A are placed in the output buffer and printed in decimal format as follows:

```
cout << F64vec2 A;
"[1]:A1 [0]:A0"
```

  Corresponding intrinsics: none
- The four, single-precision floating-point values of A are placed in the output buffer and printed in decimal format as follows:

```
cout << F32vec4 A;
"[3]:A3 [2]:A2 [1]:A1 [0]:A0"
```

  Corresponding intrinsics: none
- The lowest, single-precision floating-point value of A is placed in the output buffer and printed.

```
cout << F32vec1 A;
```

  Corresponding intrinsics: none

Element Access Operations

- `double d = F64vec2 A[int i]`

  Read one of the two, double-precision floating-point values of A without modifying the corresponding floating-point value. Permitted values of i are 0 and 1. For example:

```
double d = F64vec2 A[1];
```

  If DEBUG is enabled and i is not one of the permitted values (0 or 1), a diagnostic message is printed and the program aborts. Corresponding intrinsics: none
- `float f = F32vec4 A[int i]`

  Read one of the four, single-precision floating-point values of A without modifying the corresponding floating point value. Permitted values of i are 0, 1, 2, and 3. For example:

```
float f = F32vec4 A[2];
```

  If DEBUG is enabled and i is not one of the permitted values (0-3), a diagnostic message is printed and the program aborts.
  Corresponding intrinsics: none

Element Assignment Operations

- `F64vec4 A[int i] = double d;`

  Modify one of the two, double-precision floating-point values of A. Permitted values of `int i` are 0 and 1. For example:

```
F32vec4 A[1] = double d;
F32vec4 A[int i] = float f;
```

- Modify one of the four, single-precision floating-point values of A. Permitted values of int i are 0, 1, 2, and 3. For example:

```
F32vec4 A[3] = float f;
```

  If DEBUG is enabled and int i is not one of the permitted values (0-3), a diagnostic message is printed and the program aborts.
  Corresponding intrinsics: none.

Load and Store Operators

- Loads two, double-precision floating-point values, copying them into the two, floating-point values of A. No assumption is made for alignment.

```
void loadu(F64vec2 A, double *p)
```

  Corresponding intrinsic: _mm_loadu_pd
- Stores the two, double-precision floating-point values of A. No assumption is made for alignment.

```
void storeu(float *p, F64vec2 A);
```

  Corresponding intrinsic: _mm_storeu_pd
- Loads four, single-precision floating-point values, copying them into the four floating-point values of A. No assumption is made for alignment.

```
void loadu(F32vec4 A, double *p)
```

  Corresponding intrinsic: _mm_loadu_ps
- Stores the four, single-precision floating-point values of A. No assumption is made for alignment.

```
void storeu(float *p, F32vec4 A);
```

  Corresponding intrinsic: _mm_storeu_ps

Unpack Operators

- Selects and interleaves the lower, double-precision floating-point values from A and B.

```

F64vec2 R = unpack_low(F64vec2 A, F64vec2 B);
```

  Corresponding intrinsic: _mm_unpacklo_pd(a, b)
- Selects and interleaves the higher, double-precision floating-point values from A and B.

```
F64vec2 R = unpack_high(F64vec2 A, F64vec2 B);
```

  Corresponding intrinsic: _mm_unpackhi_pd(a, b)
- Selects and interleaves the lower two, single-precision floating-point values from A and B.

```
F32vec4 R = unpack_low(F32vec4 A, F32vec4 B);
```

  Corresponding intrinsic: _mm_unpacklo_ps(a, b)
- Selects and interleaves the higher two, single-precision floating-point values from A and B.

```
F32vec4 R = unpack_high(F32vec4 A F32vec4 B);
```

  Corresponding intrinsic: _mm_unpackhi_ps(a, b)

Move Mask Operators

- Creates a 2-bit mask from the most significant bits of the two, double-precision floating-point values of A, as follows:

```
int i = move_mask(F64vec2 A)
i := sign(a1)<<1 | sign(a0)<<0
```

Corresponding intrinsic: `_mm_movemask_pd`

- Creates a 4-bit mask from the most significant bits of the four, single-precision floating-point values of `A`, as follows:

```
int i = move_mask(F32vec4 A)
i := sign(a3)<<3 | sign(a2)<<2 | sign(a1)<<1 | sign(a0)<<0
```

Corresponding intrinsic: `_mm_movemask_ps`

Classes Quick Reference

This appendix contains tables listing operators to perform various SIMD operations, corresponding intrinsics to perform those operations, and the classes that implement those operations. The classes listed here belong to the Intel® C++ Class Libraries for SIMD Operations.

In the following tables,

- N/A indicates that the operator is not implemented in that particular class. For example, in the Logical Operations table, the `Andnot` operator is not implemented in the `F32vec4` and `F32vec1` classes.
- All other entries under Classes indicate that those operators are implemented in those particular classes, and the entries under the Classes columns provide the suffix for the corresponding intrinsic. For example, consider the Arithmetic Operations: Part1 table, where the corresponding intrinsic is `_mm_add_[x]` and the entry `epi16` is under the `I16vec8` column. It means that the `I16vec8` class implements the addition operators and the corresponding intrinsic is `_mm_add_epi16`.

Logical Operations:

Operators	Corresponding Intrinsic	Classes				
		I128vec1, I64vec2, I32vec4, I16vec8, I8vec16	**I64vec1, I32vec2, I16vec4, I8vec8**	**F64vec2**	**F32vec4**	**F32vec1**
`&, &=`	`_mm_and_[x]`	si128	si64	pd	ps	ps
`|, |=`	`_mm_or_[x]`	si128	si64	pd	ps	ps
`^, ^=`	`_mm_xor_[x]`	si128	si64	pd	ps	ps
`Andnot`	`_mm_andnot_[x]`	si128	si64	pd	N/A	N/A

Arithmetic Operations: Part 1

Operators	Corresponding Intrinsic	Classes			
		I64vec2	**I32vec4**	**I16vec8**	**I8vec16**
`+, +=`	`_mm_add_[x]`	epi64	epi32	epi16	epi8
`-, -=`	`_mm_sub_[x]`	epi64	epi32	epi16	epi8
`*, *=`	`_mm_mullo_[x]`	N/A	N/A	epi16	N/A
`/, /=`	`_mm_div_[x]`	N/A	N/A	N/A	N/A
`mul_high`	`_mm_mulhi_[x]`	N/A	N/A	epi16	N/A
`mul_add`	`_mm_madd_[x]`	N/A	N/A	epi16	N/A
`sqrt`	`_mm_sqrt_[x]`	N/A	N/A	N/A	N/A

Operators	Corresponding Intrinsic	Classes			
		I64vec2	I32vec4	I16vec8	I8vec16
rcp	_mm_rcp_[x]	N/A	N/A	N/A	N/A
rcp_nr	_mm_rcp_[x] _mm_add_[x] _mm_sub_[x] _mm_mul_[x]	N/A	N/A	N/A	N/A
rsqrt	_mm_rsqrt_[x]	N/A	N/A	N/A	N/A
rsqrt_nr	_mm_rsqrt_[x] _mm_sub_[x] _mm_mul_[x]	N/A	N/A	N/A	N/A

Arithmetic Operations: Part 2

Operators	Corresponding Intrinsic	Classes					
		I32vec2	I16vec4	I8vec8	F64vec2	F32vec4	F32vec1
+, +=	_mm_add_[x]	pi32	pi16	pi8	pd	ps	ss
-, -=	_mm_sub_[x]	pi32	pi16	pi8	pd	ps	ss
*, *=	_mm_mullo_[x]	N/A	pi16	N/A	pd	ps	ss
/, /=	_mm_div_[x]	N/A	N/A	N/A	pd	ps	ss
mul_high	_mm_mulhi_[x]	N/A	pi16	N/A	N/A	N/A	N/A
mul_add	_mm_madd_[x]	N/A	pi16	N/A	N/A	N/A	N/A
sqrt	_mm_sqrt_[x]	N/A	N/A	N/A	pd	ps	ss
rcp	_mm_rcp_[x]	N/A	N/A	N/A	pd	ps	ss
rcp_nr	_mm_rcp_[x] _mm_add_[x] _mm_sub_[x] _mm_mul_[x]	N/A	N/A	N/A	pd	ps	ss
rsqrt	_mm_rsqrt_[x]	N/A	N/A	N/A	pd	ps	ss
rsqrt_nr	_mm_rsqrt_[x] _mm_sub_[x] _mm_mul_[x]	N/A	N/A	N/A	pd	ps	ss

Shift Operations: Part 1

Operators	Corresponding Intrinsic	Classes				
		I128vec1	I64vec2	I32vec4	I16vec8	I8vec16
>>,>>=	_mm_srl_[x] _mm_srli_[x] _mm_sra__[x]	N/A N/A N/A	epi64 epi64 N/A	epi32 epi32 epi32	epi16 epi16 epi16	N/A N/A N/A

Operators	Corresponding Intrinsic	Classes				
		I128vec1	I64vec2	I32vec4	I16vec8	I8vec16
	_mm_srai_[x]	N/A	N/A	epi32	epi16	N/A
<<, <<=	_mm_sll_[x] _mm_slli_[x]	N/A N/A	epi64 epi64	epi32 epi32	epi16 epi16	N/A N/A

Shift Operations: Part 2

Operators	Corresponding Intrinsic	Classes			
		I64vec1	I32vec2	I16vec4	I8vec8
>>,>>=	_mm_srl_[x] _mm_srli_[x] _mm_sra__[x] _mm_srai_[x]	si64 si64 N/A N/A	pi32 pi32 pi32 pi32	pi16 pi16 pi16 pi16	N/A N/A N/A N/A
<<, <<=	_mm_sll_[x] _mm_slli_[x]	si64 si64	pi32 pi32	pi16 pi16	N/A N/A

Comparison Operations: Part 1

Operators	Corresponding Intrinsic	Classes					
		I32vec4	I16vec8	I8vec16	I32vec2	I16vec4	I8vec8
cmpeq	_mm_cmpeq_[x]	epi32	epi16	epi8	pi32	pi16	pi8
cmpneq	_mm_cmpeq_[x] _mm_andnot_[y]*	epi32 si128	epi16 si128	epi8 si128	pi32 si64	pi16 si64	pi8 si64
cmpgt	_mm_cmpgt_[x]	epi32	epi16	epi8	pi32	pi16	pi8
cmpge	_mm_cmpge_[x] _mm_andnot_[y]*	epi32 si128	epi16 si128	epi8 si128	pi32 si64	pi16 si64	pi8 si64
cmplt	_mm_cmplt_[x]	epi32	epi16	epi8	pi32	pi16	pi8
cmple	_mm_cmple_[x] _mm_andnot_[y]*	epi32 si128	epi16 si128	epi8 si128	pi32 si64	pi16 si64	pi8 si64
cmpngt	_mm_cmpngt_[x]	epi32	epi16	epi8	pi32	pi16	pi8
cmpnge	_mm_cmpnge_[x]	N/A	N/A	N/A	N/A	N/A	N/A
cmpnlt	_mm_cmpnlt_[x]	N/A	N/A	N/A	N/A	N/A	N/A
cmpnle	_mm_cmpnle_[x]	N/A	N/A	N/A	N/A	N/A	N/A

* Note that _mm_andnot_[y] intrinsics do not apply to the fvec classes.

Comparison Operations: Part 2

Operators	Corresponding Intrinsic	Classes		
		F64vec2	**F32vec4**	**F32vec1**
cmpeq	_mm_cmpeq_[x]	pd	ps	ss
cmpneq	_mm_cmpeq_[x] _mm_andnot_[y]*	pd	ps	ss
cmpgt	_mm_cmpgt_[x]	pd	ps	ss
cmpge	_mm_cmpge_[x] _mm_andnot_[y]*	pd	ps	ss
cmplt	_mm_cmplt_[x]	pd	ps	ss
cmple	_mm_cmple_[x] _mm_andnot_[y]*	pd	ps	ss
cmpngt	_mm_cmpngt_[x]	pd	ps	ss
cmpnge	_mm_cmpnge_[x]	pd	ps	ss
cmpnlt	_mm_cmpnlt_[x]	pd	ps	ss
cmpnle	_mm_cmpnle_[x]	pd	ps	ss

* Note that `_mm_andnot_[y]` intrinsics do not apply to the `fvec` classes.

Conditional Select Operations: Part 1

Operators	Corresponding Intrinsic	Classes					
		I32vec4	**I16vec8**	**I8vec16**	**I32vec2**	**I16vec4**	**I8vec8**
select_eq	_mm_cmpeq_[x] _mm_and_[y] _mm_andnot_[y]* _mm_or_[y]	epi32 si128 si128 si128	epi16 si128 si128 si128	epi8 si128 si128 si128	pi32 si64 si64 si64	pi16 si64 si64 si64	pi8 si64 si64 si64
select_neq	_mm_cmpeq_[x] _mm_and_[y] _mm_andnot_[y]* _mm_or_[y]	epi32 si128 si128 si128	epi16 si128 si128 si128	epi8 si128 si128 si128	pi32 si64 si64 si64	pi16 si64 si64 si64	pi8 si64 si64 si64
select_gt	_mm_cmpgt_[x] _mm_and_[y] _mm_andnot_[y]* _mm_or_[y]	epi32 si128 si128 si128	epi16 si128 si128 si128	epi8 si128 si128 si128	pi32 si64 si64 si64	pi16 si64 si64 si64	pi8 si64 si64 si64
select_ge	_mm_cmpge_[x] _mm_and_[y] _mm_andnot_[y]* _mm_or_[y]	epi32 si128 si128 si128	epi16 si128 si128 si128	epi8 si128 si128 si128	pi32 si64 si64 si64	pi16 si64 si64 si64	pi8 si64 si64 si64
select_lt	_mm_cmplt_[x] _mm_and_[y] _mm_andnot_[y]* _mm_or_[y]	epi32 si128 si128 si128	epi16 si128 si128 si128	epi8 si128 si128 si128	pi32 si64 si64 si64	pi16 si64 si64 si64	pi8 si64 si64 si64

Operators	Corresponding Intrinsic	Classes					
		I32vec4	I16vec8	I8vec16	I32vec2	I16vec4	I8vec8
select_le	_mm_cmple_[x] _mm_and_[y] _mm_andnot_[y]* _mm_or_[y]	epi32 si128 si128 si128	epi16 si128 si128 si128	epi8 si128 si128 si128	pi32 si64 si64 si64	pi16 si64 si64 si64	pi8 si64 si64 si64
select_ngt	_mm_cmpgt_[x]	N/A	N/A	N/A	N/A	N/A	N/A
select_nge	_mm_cmpge_[x]	N/A	N/A	N/A	N/A	N/A	N/A
select_nlt	_mm_cmplt_[x]	N/A	N/A	N/A	N/A	N/A	N/A
select_nle	_mm_cmple_[x]	N/A	N/A	N/A	N/A	N/A	N/A

* Note that `_mm_andnot_[y]` intrinsics do not apply to the `fvec` classes.

Conditional Select Operations: Part 2

Operators	Corresponding Intrinsic	Classes		
		F64vec2	F32vec4	F32vec1
select_eq	_mm_cmpeq_[x] _mm_and_[y] _mm_andnot_[y]* _mm_or_[y]	pd	ps	ss
select_neq	_mm_cmpeq_[x] _mm_and_[y] _mm_andnot_[y]* _mm_or_[y]	pd	ps	ss
select_gt	_mm_cmpgt_[x] _mm_and_[y] _mm_andnot_[y]* _mm_or_[y]	pd	ps	ss
select_ge	_mm_cmpge_[x] _mm_and_[y] _mm_andnot_[y]* _mm_or_[y]	pd	ps	ss
select_lt	_mm_cmplt_[x] _mm_and_[y] _mm_andnot_[y]* _mm_or_[y]	pd	ps	ss
select_le	_mm_cmple_[x] _mm_and_[y] _mm_andnot_[y]* _mm_or_[y]	pd	ps	ss
select_ngt	_mm_cmpgt_[x]	pd	ps	ss
select_nge	_mm_cmpge_[x]	pd	ps	ss
select_nlt	_mm_cmplt_[x]	pd	ps	ss

Operators	Corresponding Intrinsic	Classes		
		F64vec2	F32vec4	F32vec1
select_nle	_mm_cmple_[x]	pd	ps	ss

* Note that _mm_andnot_[y] intrinsics do not apply to the fvec classes.

Packing and Unpacking Operations: Part 1

Operators	Corresponding Intrinsic	Classes				
		I64vec2	I32vec4	I16vec8	I8vec16	I32vec2
unpack_high	_mm_unpackhi_[x]	epi64	epi32	epi16	epi8	pi32
unpack_low	_mm_unpacklo_[x]	epi64	epi32	epi16	epi8	pi32
pack_sat	_mm_packs_[x]	N/A	epi32	epi16	N/A	pi32
packu_sat	_mm_packus_[x]	N/A	N/A	epi16	N/A	N/A
sat_add	_mm_adds_[x]	N/A	N/A	epi16	epi8	N/A
sat_sub	_mm_subs_[x]	N/A	N/A	epi16	epi8	N/A

Packing and Unpacking Operations: Part 2

Operators	Corresponding Intrinsic	Classes				
		I16vec4	I8vec8	F64vec2	F32vec4	F32vec1
unpack_high	_mm_unpackhi_[x]	pi16	pi8	pd	ps	N/A
unpack_low	_mm_unpacklo_[x]	pi16	pi8	pd	ps	N/A
pack_sat	_mm_packs_[x]	pi16	N/A	N/A	N/A	N/A
packu_sat	_mm_packus_[x]	pu16	N/A	N/A	N/A	N/A
sat_add	_mm_adds_[x]	pi16	pi8	pd	ps	ss
sat_sub	_mm_subs_[x]	pi16	pi8	pi16	pi8	pd

Conversions Operations:

Conversion operations can be performed using intrinsics only. There are no classes implemented to correspond to these intrinsics.

Operators	Corresponding Intrinsic
F64vec2ToInt	_mm_cvttsd_si32
F32vec4ToF64vec2	_mm_cvtps_pd
F64vec2ToF32vec4	_mm_cvtpd_ps
IntToF64vec2	_mm_cvtsi32_sd
F32vec4ToInt	_mm_cvtt_ss2si

Operators	Corresponding Intrinsic
F32vec4ToIs32vec2	_mm_cvttps_pi32
IntToF32vec4	_mm_cvtsi32_ss
Is32vec2ToF32vec4	_mm_cvtpi32_ps

Programming Example

This sample program uses the `F32vec4` class to average the elements of a twenty element floating point array.

```
//Include Intel® Streaming SIMD Extension (Intel® SSE) Class Definitions
 #include <fvec.h>

//Shuffle any two single precision floating point from a
//into low two SP FP and shuffle any two SP FP from b
//into high two SP FP of destination

#define SHUFFLE(a,b,i) (F32vec4)_mm_shuffle_ps(a,b,i)
#include <stdio.h>
#define SIZE 20

//Global variables
 float result;
_MM_ALIGN16 float array[SIZE];

//*****************************************************
// Function: Add20ArrayElements
// Add all the elements of a twenty element array
//*****************************************************
void Add20ArrayElements (F32vec4 *array, float *result) {
   F32vec4 vec0, vec1;
   vec0 = _mm_load_ps ((float *) array); // Load array's first four floats

   //*****************************************************
   // Add all elements of the array, four elements at a time
   //*****************************************************
   vec0 += array[1]; // Add elements 5-8
   vec0 += array[2]; // Add elements 9-12
   vec0 += array[3]; // Add elements 13-16
   vec0 += array[4]; // Add elements 17-20

   //*****************************************************
   // There are now four partial sums.
   // Add the two lowers to the two raises,
   // then add those two results together
   //*****************************************************
   vec1 = SHUFFLE(vec1, vec0, 0x40);
   vec0 += vec1;
   vec1 = SHUFFLE(vec1, vec0, 0x30);
   vec0 += vec1;
   vec0 = SHUFFLE(vec0, vec0, 2);
   _mm_store_ss (result, vec0); // Store the final sum
}
```

```
void main(int argc, char *argv[]) {
   int i;

//Initialize the array
   for (i=0; i < SIZE; i++) { array[i] = (float) i; }

//Call function to add all array elements
   Add20ArrayElements (array, &result);

//Print average array element value
   printf ("Average of all array values = %f\n", result/20.);
   printf ("The correct answer is %f\n\n\n", 9.5);
}
```

Intel's valarray Implementation

The Intel® oneAPI DPC++/C++ Compiler provides a high performance implementation of specialized one-dimensional valarray operations for the C++ standard STL valarray container.

The standard C++ valarray template consists of array/vector operations for high performance computing. These operations are designed to exploit high performance hardware features such as parallelism and achieve performance benefits.

Intel's valarray implementation uses the Intel® Integrated Performance Primitives (Intel® IPP), which is part of the product. Select Intel® IPP when you install the product.

The valarray implementation consists of a replacement header, `<valarray>`, that provides a specialized, high-performance implementation for the following operators and types:

Operator	Type
abs, acos, acosh, asin, asinh, atan, atan2, atanh, cbrt, cdfnorm, ceil, cos, cosh, erf, erfc, erfinv, exp, expm1, floor, hypot, inv, invcbrt, invsqrt, ln, log, log10, log1p, nearbyint, pow, pow2o3, pow3o2, powx, rint, round, sin, sinh, sqrt, tan, tanh, trunk	float, double
add, conj, div, mul, mulbyconj, mul, sub	Ipp32fc, Ipp64fc
addition, subtraction, division, multiplication	float, double
bitwise or, and, xor	(all unsigned) char, short, int
min, max, sum	signed or short/signed int, float, double

Use valarray in Source Code

valarray is not available for SYCL.

Intel's valarray implementation allows you to declare large arrays for parallel processing. Improved implementation of valarray is tied up with calling the Intel® IPP libraries that are part of Intel® IPP.

To use valarrays in your source code, include the valarray header file, `<valarray>`. The `<valarray>` header file is located in the path `<installdir>/perf_header`.

The following example shows a valarray addition operation (+) specialized through use of Intel's implementation of valarray:

```
#include <valarray>
void test( )
{
```

```
    std::valarray<float> vi(N), va(N);
    …
    vi = vi + va;  //array addition
    …
}
```

> **NOTE**
> To use the static merged library containing all CPU-specific optimized versions of the library code, you need to call the `ippStaticInit` function first, before any Intel® IPP calls. This ensures automatic dispatch to the appropriate version of the library code for Intel® processor and the generic version of the library code for non-Intel processors at runtime. If you do not call `ippStaticInit` first, the merged library will use the generic instance of the code. If you are using the dynamic version of the libraries, you do not need to call `ippStaticInit`.

Compiling valarray Source Code

To compile your valarray source code, the compiler option, `/Quse-intel-optimized-headers` (for Windows) or `-use-intel-optimized-headers` (for Linux), is used to include the required valarray header file and all the necessary Intel® IPP library files.

The following examples illustrate how to compile and link a program to include the Intel valarray replacement header file and link with the Intel® IPP libraries. Refer to the Intel® IPP documentation for details.

In the following examples, the merged libraries use a static library that contains all the CPU-specific variants of the library code.

Linux Examples

The following command line performs a one-step compilation for a system based on Intel® 64 architecture, running Linux OS:

```
icpx -use-intel-optimized-headers source.cpp
```

The following command lines perform separate compile and link steps for a system based on Intel® 64 architecture, running Linux OS:

so (dynamic):

```
icpx -use-intel-optimized-headers -c source.cpp
```

```
icpx source.o -use-intel-optimized-headers -shared-intel
```

Merged (static):

```
icpx -use-intel-optimized-headers -c source.cpp
```

```
icpx source.o -use-intel-optimized-headers
```

Windows Examples

The following command line performs a one-step compilation for a system based on IA-32 architecture, running Windows OS:

```
icx /Quse-intel-optimized-headers source.cpp
```

The following command lines perform separate compile and link steps for a system based on IA-32 architecture, running Windows OS:

DLL (dynamic):

```
icx /Quse-intel-optimized-headers /c source.cpp
```

```
icx source.obj /Quse-intel-optimized-headers
```

Merged (static):

```
icx /Quse-intel-optimized-headers /Qipp-link:static /c source.cpp
icx source.obj /Quse-intel-optimized-headers /Qipp-link:static
```

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.
Notice revision #20201201

Intel's C++ Asynchronous I/O Extensions for Windows

Intel's C/C++ asynchronous input/output (Intel's C/C++ AIO) extensions, like library functions or classes, can be used to improve the performance of C/C++ applications by executing I/O operations in asynchronous mode. The extensions initiate I/O operation and immediately resume normal tasks while the I/O operations are executed in parallel.

Intel's C/C++ asynchronous I/O extensions are supported on IA-32 architecture-based (C/C++ only) and Intel® 64 architecture-based Windows platforms.

Intel's C/C++ AIO library functions and template class are implemented in the `libicaio.lib` library. This library is supplied as part of the Intel® oneAPI DPC++/C++ Compiler package and is installed into the common directory: `<install-dir>/lib`.

Types of Intel's C/C++ Asynchronous I/O Extensions

Intel's C/C++ asynchronous I/O extensions comprise the following:

- **Asynchronous I/O Library:** A set of POSIX-based asynchronous I/O library functions, supported on Windows operating systems, for applications written in C/C++ language. The interface file is `aio.h`.
- **Asynchronous I/O Template Class:** An `asych_class` template class, supported on Windows operating systems, for applications written in C++ language. This template class can be used to introduce asynchronous execution of I/O operations with the Standard Template Library's (STL's) streams classes. The interface file is `aiostream.h`.

See Also

Intel's C++ Asynchronous I/O Library for Windows
Intel's C++ Asynchronous I/O Class for Windows

Intel's C++ Asynchronous I/O Library for Windows

Intel's C/C++ asynchronous I/O (AIO) library implementation for the Windows operating system (on IA-32 (C/C++ only) and Intel® 64 platforms) is similar to the POSIX AIO library implementation for the Linux operating system.

The differences between Intel's C/C++ AIO Windows OS implementation and the standard POSIX AIO implementation are listed below:

- In `struct aiocb`,
  - The Windows OS compatible type `HANDLE` replaces the POSIX AIO type `unsigned int` for the file descriptor `aio_fildes`.
  - The type `intptr_t` replaces the POSIX AIO types `ssize_t` and `__off_t`.

- The structure specifying the signal event descriptor, `struct sigevent` is similar to the Linux operating system implementation of the POSIX AIO library. It differs from the Linux implementation in the following ways:
  - Signal notification and non-notification for thread call-back is supported
  - Signal notification on completion of the AIO operation is *not* supported

    This is true for programs that were already written for Linux/Unix and ported to Windows OS that wish to setup an AIO completion handler without the name of the handler set in the `aiocb struct`. Because of the way that signals are supported in Windows, this is impossible to implement. For new applications, or to port existing applications, the programmer should set the name of the handler before calling the `aio_read` or `aio_write` routines. For example:

```
static void aio_CompletionRoutine(sigval_t sigval)
{
    // … code …
}

… code …

my_aio.aio_sigevent.sigev_notify          = SIGEV_THREAD;
my_aio.aio_sigevent.sigev_notify_function = aio_CompletionRoutine;
```

> **NOTE**
> The POSIX AIO library and the Microsoft SDK provide similar AIO functions. The main difference between the POSIX AIO functions and the Windows operating system-based AIO functions is that while POSIX allows you to execute AIO operations with any file, the Windows operating system executes AIO operations only with files flagged with `FILE_FLAG_OVERLAPPED`.

Intel's asynchronous I/O library functions listed below are all based on POSIX AIO functions. They are defined in the `aio.h` file.

- `aio_read()`
- `aio_write()`
- `aio_suspend()`
- `aio_error()`
- `aio_return()`
- `aio_fsync()`
- `aio_cancel()`
- `lio_listio()`

aio_read

Performs an asynchronous read operation.

Syntax

`int aio_read(struct aiocb *aiocbp);`

Description

The `aio_read()` function requests an asynchronous read operation, calling the function,

```
"ReadFile(hFile, lpBuffer, nNumberOfBytesToRead, lpNumberOfBytesRead, NULL);"
```

where,

- `hFile` is given by *aiocbp->aio_fildes*
- `lpBuffer` is given by *aiocbp->aio_buf*
- `nNumberOfBytesToRead` is given by *aiocbp->aio_nbytes*

Use the function `aio_return()` to retrieve the actual bytes read in `lpNumberOfBytesRead`.

Use the extension *aiocb->aio_offset == (intptr_t)-1* to start the read operation after the last read record. This extension avoids extra file positioning and enhances performance.

Returns

0: On success

-1: On error

To get the correct error code, use `errno`. To get the error that occurred during asynchronous read operation, use `aio_error()` function.

See Also

Example Code for aio_read()

aio_write

Performs an asynchronous write operation.

Syntax

```
int aio_write(struct aiocb *aiocbp);
```

Description

The `aio_write()` function requests an asynchronous write operation, calling the function,

```
"WriteFile(hFile, lpBuffer, nNumberOfBytesToWrite, lpNumberOfBytesWritten, NULL);
```

where,

- `hFile` is given by *aiocbp->aio_fildes*
- `lpBuffer` is given by *aiocbp->aio_buf*
- `nNumberOfBytesToWrite` is given by *aiocbp->aio_nbytes*

Use the function `aio_return()` to retrieve the actual bytes written in `lpNumberOfBytesWritten`.

Use the extension *aiocb->aio_offset == (intptr_t)-1* to start the write operation after the last written record. This extension avoids extra file positioning and enhances performance.

Returns

0: On success

-1: On error

To get the correct error code, use `errno`. To get the error that occurred during asynchronous write operation, use `aio_error()` function.

See Also

Example Code for aio_write()

Example for aio_read and aio_write Functions

The example illustrates the performance gain of the asynchronous I/O usage in comparison with synchronous I/O usage. In the example, 5.6 MB of data is asynchronously written with the main program computation, which is the scalar multiplication of two vectors with some normalization.

C-source File Executing a Scalar Multiplication

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
```

```
double do_compute(double A, double B, int arr_len)
{
  int i;
  double   res = 0;
  double  *xA = malloc(arr_len * sizeof(double));
  double  *xB = malloc(arr_len * sizeof(double));
  if ( !xA || !xB )
   abort();
   for (i = 0; i < arr_len; i++) {
    xA[i] = sin(A);
    xB[i] = cos(B);
    res = res + xA[i]*xA[i];
   }
  free(xA);
  free(xB);
 return res;
}
```

C-main-source File Using Asynchronous I/O Implementation (Example One)

```
#define DIM_X   123/*123*/
#define DIM_Y   70000
double  aio_dat[DIM_Y /*12MB*/] = {0};
double  aio_dat_tmp[DIM_Y /*12MB*/];

#include <stdio.h>
#include <aio.h>

typedef struct aiocb  aiocb_t;
   aiocb_t   my_aio;
   aiocb_t  *my_aio_list[1] = {&my_aio};

int main()
{
  double do_compute(double A, double B, int arr_len);
  int i, j;
  HANDLE fd = CreateFile("aio.dat",
  GENERIC_READ | GENERIC_WRITE,
  FILE_SHARE_READ,
  NULL,
  OPEN_ALWAYS,
  FILE_ATTRIBUTE_NORMAL,
  NULL);
/* Do some complex computation */
for (i = 0; i < DIM_X; i++) {
 for ( j = 0; j < DIM_Y; j++ )
 aio_dat[j] = do_compute(i, j, DIM_X);

  if (i) aio_suspend(my_aio_list, 1, 0);
  my_aio.aio_fildes = fd;
  my_aio.aio_buf    = memcpy(aio_dat_tmp, aio_dat, sizeof(aio_dat_tmp));
  my_aio.aio_nbytes = sizeof(aio_dat_tmp);
  my_aio.aio_offset = (intptr_t)-1;
  my_aio.aio_sigevent.sigev_notify = SIGEV_NONE;

  if ( aio_write((void*)&my_aio) == -1 ) {
  printf("ERROR!!! %s\n", "aio_write()==-1");
  abort();}
  }
```

```
 aio_suspend(my_aio_list, 1, 0);
 return 0;
}
```

C-main-source File Using Asynchronous I/O Implementation (Example Two)

```
// icx -c do_compute.c
// icx aio_sample2.c do_compute.obj
// aio_sample2.exe

#define DIM_X   123
#define DIM_Y    70
double  aio_dat[DIM_Y] = {0};
double  aio_dat_tmp[DIM_Y];
static volatile int aio_flg = 1;

#include <aio.h>
typedef struct aiocb  aiocb_t;
aiocb_t               my_aio;
#define WAIT { while (!aio_flg); aio_flg = 0; }
#define aio_OPEN(_fname )\
CreateFile(_fname,                        \
           GENERIC_READ | GENERIC_WRITE, \
           FILE_SHARE_READ,              \
           NULL,                         \
           OPEN_ALWAYS,                  \
           FILE_ATTRIBUTE_NORMAL,        \
           NULL)

static void aio_CompletionRoutine(sigval_t sigval)
{
    aio_flg = 1;
}

int main()
{
    double do_compute(double A, double B, int arr_len);
    int      i, j, res;
    char    *fname = "aio_sample2.dat";
    HANDLE   aio_fildes = aio_OPEN(fname);

    my_aio.aio_fildes = aio_fildes;
    my_aio.aio_nbytes = sizeof(aio_dat_tmp);
    my_aio.aio_sigevent.sigev_notify          = SIGEV_THREAD;
    my_aio.aio_sigevent.sigev_notify_function = aio_CompletionRoutine;

    /*
    ** writing
    */
    my_aio.aio_offset = -1;
    printf("Writing\n");
    for (i = 0; i < DIM_X; i++) {
        for (j = 0; j < DIM_Y; j++)
            aio_dat[j] = do_compute(i, j, DIM_X);
        WAIT;
        my_aio.aio_buf = memcpy(aio_dat_tmp, aio_dat, sizeof(aio_dat_tmp));
        res = aio_write(&my_aio);
        if (res) {printf("res!=0\n");abort();}
    }
```

```
    //
    // flushing
    //
    printf("Flushing\n");
    WAIT;
    res = aio_fsync(O_SYNC, &my_aio);
    if (res) {printf("res!=0\n");abort();}
    WAIT;

    //
    // reading
    //
    printf("Reading\n");
    my_aio.aio_offset = 0;
    my_aio.aio_buf    = (volatile char*)aio_dat_tmp;
    for (i = 0; i < DIM_X; i++) {
        aio_read(&my_aio);
        for (j = 0; j < DIM_Y; j++)
            aio_dat[j] = do_compute(i, j, DIM_X);
        WAIT;
        res = aio_return(&my_aio);
        if (res != sizeof(aio_dat)) {
            printf("aio_read() did read %d bytes, expecting %d bytes\n", res, sizeof(aio_dat));
        }

        for (j = 0; j < DIM_Y; j++)
            if ( aio_dat[j] != aio_dat_tmp[j] )
                {printf("ERROR: aio_dat[j] != aio_dat_tmp[j]\n I=%d J=%d\n", i, j); abort();}
        my_aio.aio_offset += my_aio.aio_nbytes;
    }

    CloseHandle(aio_fildes);

    printf("\nDone\n");

return 0;
}
```

See Also

aio_read()

aio_write()

aio_suspend

Suspends the calling process until one of the asynchronous I/O operations completes.

Syntax

`int aio_suspend(const struct aiocb * const cblist[], int n, const struct timespec *timeout);`

Arguments

cblist[]	Pointer to a control block on which I/O is initiated

n	Length of *cblist* list
**timeout*	Time interval to suspend the calling process

Description

The `aio_suspend()` function is like a wait operation. It suspends the calling process until,

- At least one of the asynchronous I/O requests in the list *cblist* of length *n* has completed
- A signal is delivered
- The time interval indicated in *timeout* is not `NULL` and has passed.

Each item in the *cblist* list must either be `NULL` (when it is ignored), or a pointer to a control block on which I/O was initiated using `aio_read()`, `aio_write()`, or `lio_listio()` functions.

Returns

0: On success

-1: On error

To get the correct error code, use `errno`.

See Also

Example Code for aio_suspend()

Example for aio_suspend Function

The following example illustrates a wait operation execution using the `aio_suspend()` function.

```
int aio_ex_2(HANDLE fd)
{
    static struct aiocb   aio[2];
    static struct aiocb  *aio_list[2] = {&aio[0], &aio[1]};
    int i, ret;

/* Data initialization */
IC_AIO_DATA_INIT(aio[0], fd, "rec#1\n", strlen("rec#1\n"), 0)
IC_AIO_DATA_INIT(aio[1], fd, "rec#2\n", strlen("rec#2\n"), aio[0].aio_nbytes)

/* Asynch-write */
if (aio_write(&aio[0]) == -1) return errno;
if (aio_write(&aio[1]) == -1) return errno;

/* Do some complex computation */
printf("do_compute(1000, 1.123)=%f", do_compute(1000, 1.123));

/* do the wait operation using sleep() */
ret = aio_suspend(aio_list, 2, 0);
if (ret == -1) return errno;

return 0;
}/* aio_ex_2 */
```

Result upon execution:

```
-bash-3.00$ ./a.out
-bash-3.00$ cat dat
rec#1
rec#2
```

Remarks:

1. In the example, the `IC_AIO_DATA_INIT` is defined as follows:

```
#define IC_AIO_DATA_INIT(_aio, _fd, _dat, _len, _off)\
  {memset(&_aio, 0, sizeof(_aio)); \
  _aio.aio_fildes = _fd;          \
  _aio.aio_buf    = _dat;         \
  _aio.aio_nbytes = _len;         \
  _aio.aio_offset = _off;}
```

2. The file descriptor `fd` is obtained as:

```
HANDLE fd = CreateFile("dat",
  GENERIC_READ | GENERIC_WRITE,
  FILE_SHARE_READ,
  NULL,
  OPEN_ALWAYS,
  FILE_ATTRIBUTE_NORMAL/*|FILE_FLAG_OVERLAPPED*/,
  NULL);
```

See Also

aio_suspend()

aio_error

Returns error status for asynchronous I/O requests.

Syntax

`int aio_error(const struct aiocb *aiocbp);`

Arguments

**aiocbp*	Pointer to control block from where asynchronous I/O request is generated

Description

The `aio_error()` function returns the error status for the asynchronous I/O request in the control block, which is pointed to by *aiocbp*.

Returns

EINPROGRESS: When asynchronous I/O request is not completed

ECANCELED: When asynchronous I/O request is cancelled

0: On success

Error value: On error

To get the correct error value/code, use `errno`. This is the same error value returned when an error occurs during a `ReadFile()`, `WriteFile()`, or a `FlushFileBuffers()` operation.

See Also

Example Code for aio_error()

aio_return

Returns the final return status for the asynchronous I/O request.

Syntax

`ssize_t aio_return(struct aiocb *aiocbp);`

Arguments

**aiocbp*	Pointer to control block from where asynchronous I/O request is generated

Description

The `aio_return` function returns the final return status for the asynchronous I/O request with control block pointed to by *aiocbp*.

Call this function only once for any given request, after `aio_error()` returns a value other than *EINPROGRESS*.

Returns

Return value for synchronous ReadFile()/WriteFile()/FlushFileBuffer() requests: When asynchronous I/O operation is completed

Undefined return value: When asynchronous I/O operation is not completed

Error value: When an error occurs

To get the correct error code/value, use `errno`.

See Also

Example Code for aio_return()

Example for aio_error and aio_return Functions

The following example illustrates how the `aio_error()` and `aio_return()` functions can be used.

```
int aio_ex_3(HANDLE fd)
{
 static struct aiocb aio;
 static struct aiocb *aio_list[] = {&aio};
 int    ret;
 char  *dat = "Hello from Ex-3\n";

/* Data initialization and asynchronously writing */

 IC_AIO_DATA_INIT(aio, fd, dat, strlen(dat), 0);
 if (aio_write(& aio) == -1) return errno;

 ret = aio_error(&aio);
 if ( ret == EINPROGRESS ) {
 fprintf(stderr, "ERRNO=%d STR=%s\n", ret, strerror(ret));

 ret = aio_suspend(aio_list, 1, NULL);
 if (ret == -1) return errno;}
 else if (ret)
 return ret;

 ret = aio_error(&aio);
 if (ret) return ret;

 ret = aio_return(&aio);
 printf("ret=%d\n", ret);

 return 0;
}/* aio_ex_3 */
```

Result upon execution:

```
-bash-3.00$ ./a.out
ERRNO=115 STR=Operation now in progress
ret=16
-bash-3.00$ cat dat
Hello from Ex-3
```

Remarks:

1. In the example, the `IC_AIO_DATA_INIT` is defined as follows:

```
#define IC_AIO_DATA_INIT(_aio, _fd, _dat, _len, _off)\
 {memset(&_aio, 0, sizeof(_aio)); \
 _aio.aio_fildes = _fd;  \
 _aio.aio_buf    = _dat;  \
 _aio.aio_nbytes = _len;  \
 _aio.aio_offset = _off;}
```

2. The file descriptor `fd` is obtained as:

```
HANDLE fd  = CreateFile("dat",
  GENERIC_READ | GENERIC_WRITE,
  FILE_SHARE_READ,
  NULL,
  OPEN_ALWAYS,
  FILE_ATTRIBUTE_NORMAL/*|FILE_FLAG_OVERLAPPED*/,
  NULL);
```

See Also

aio_error()

aio_return()

aio_fsync

Synchronizes all outstanding asynchronous I/O operations.

Syntax

`int aio_fsync(int op, struct aiocb *aiocbp);`

Arguments

op	Type of synchronization request operation
**aiocbp*	Pointer to control block from where asynchronous I/O request is generated

Description

The `aio_fsync()` function performs a synchronization request operation on all outstanding asynchronous I/O operations associated with *aiocbp->aio_fildes*.

Returns

0: On successfully performing a synchronization request.

-1: On error; to get the correct error code, use `errno`.

aio_cancel

Cancels outstanding asynchronous I/O requests for the file descriptor fd.

Syntax

`int aio_cancel(HANDLE fd, struct aiocb *aiocbp);`

Arguments

fd	File descriptor
**aiocbp*	Pointer to control block from where asynchronous I/O request is generated

Description

The `aio_cancel()` function cancels outstanding asynchronous I/O requests for the file descriptor *fd*. If *aiocbp* is NULL, all outstanding asynchronous I/O requests are cancelled. If *aiocbp* is not NULL, only the requests described by the control block pointed to by *aiocbp* are cancelled.

Normal asynchronous notification occurs for cancelled requests. The request return status is set to -1, and the request error status is set to ECANCELED. The control block of requests that cannot be cancelled is not changed.

Unspecified results occur if *aiocbp* is not NULL and the *fd* differs from the file descriptor with which the asynchronous operation was initiated.

Returns

AIO_CANCELLED: When all specified requests are cancellled successfully.

AIO_NOTCANCELLED: When at least one of the specified requests is still in process of being cancelled; check the status of request using `aio_error`.

AIO_ALLDONE: When all specified requests were completed before cancel call was placed.

-1: When some error occurs. To get the correct error code, use `errno`.

See Also

Example Code for aio_cancel()

Example for aio_cancel Function

The following example illustrates how `aio_cancel()` function can be used.

```
int aio_ex_4(HANDLE fd)
{
  static struct aiocb   aio;
  static struct aiocb  *aio_list[] = {&aio};
  int    ret;
  char  *dat = "Hello from Ex-4\n";

 printf("AIO_CANCELED=%d AIO_NOTCANCELED=%d\n",
 AIO_CANCELED,   AIO_NOTCANCELED);

/* Data initialization and asynchronously writing */

 IC_AIO_DATA_INIT(aio, fd, dat, strlen(dat), 0);
 if (aio_write(&aio) == -1) return errno;
```

```
 ret = aio_cancel(fd, &aio);
 if ( ret == AIO_NOTCANCELED ) {
 fprintf(stderr, "ERRNO=%d STR=%s\n", ret, strerror(ret));
 ret = aio_suspend(aio_list, 1, NULL);
 if (ret == -1) return errno;}

 ret = aio_cancel(fd, &aio);
 if ( ret == AIO_CANCELED )
 fprintf(stderr, "ERRNO=%d STR=%s\n", ret, strerror(ret));
 else if (ret) return ret;

 return 0;
}/* aio_ex_4 */
```

Result upon execution:

```
-bash-3.00$ ./a.out
AIO_CANCELED=0 AIO_NOTCANCELED=1
ERRNO=1 STR=Operation not permitted
-bash-3.00$ cat dat
Hello from Ex-4
-bash-3.00$
```

Remarks:

1. In the example, the `IC_AIO_DATA_INIT` is defined as follows:

```
#define IC_AIO_DATA_INIT(_aio, _fd, _dat, _len, _off)\
 {memset(&_aio, 0, sizeof(_aio)); \
 _aio.aio_fildes = _fd;          \
 _aio.aio_buf    = _dat;         \
 _aio.aio_nbytes = _len;         \
 _aio.aio_offset = _off;}
```

2. The file descriptor `fd` is obtained as:

```
HANDLE fd = CreateFile("dat",
  GENERIC_READ | GENERIC_WRITE,
  FILE_SHARE_READ,
  NULL,
  OPEN_ALWAYS,
  FILE_ATTRIBUTE_NORMAL/*|FILE_FLAG_OVERLAPPED*/,
  NULL);
```

See Also

aio_cancel()

lio_listio

Performs an asynchronous read operation.

Syntax

`int lio_listio(int mode, struct aiocb *list[], int nent, struct sigevent *sig);`

Arguments

mode	Takes following values declared in `<aio.h>` file:

	• `LIO_WAIT`: Use when you want the function to return only after completing I/O operations (synchronous I/O operations) • `LIO_NOWAIT`: Use when you want the function to return as soon as I/O operations are queued (asynchronous I/O requests)
**list[]*	Array of the `aiocb` pointers specifying the submitted I/O requests; NULL elements in the array are ignored
nent	Number of elements in the array
**sig*	Determines if asynchronous notification is sent after all I/O operations completes; takes following values: • 0: Asynchronous notification occurs; a queued signal, with an application-defined value, is generated when an asynchronous I/O request occurs • 1: Asynchronous notification does not occur even when asynchronous I/O requests are processed • 2: Asynchronous notification occurs; a notification function is called to perform notification

Description

The `lio_listio()` function initiates a list of I/O requests with a single function call.

The *mode* argument determines whether the function returns when all the I/O operations are completed, or as soon as the operations are queued.

If the mode argument is `LIO_WAIT`, the function waits until all I/O operations are complete. The *sig* argument is ignored in this case.

If the *mode* argument is `LIO_NOWAIT`, the function returns immediately. Asynchronous notification occurs according to the *sig* argument after all the I/O operations complete.

Returns

When *mode*=`LIO_NOWAIT` the `lio_listio()` function returns:

- **0**: I/O operations are successfully queued
- **-1**: Error; I/O operations not queued; to get the proper error code, use `errno`.

When *mode*=`LIO_WAIT` the `lio_listio()` function returns:

- **0**: I/O operations specified completed successfully
- **-1**: Error; I/O operations not completed; to get the proper error code, use `errno`.

See Also

Example Code for lio_listio()

Example for lio_listio Function

The following example illustrates how the `lio_listio()` function can be used.

```
int aio_ex_5(HANDLE fd)
{
  static struct aiocb   aio[2];
  static struct aiocb  *aio_list[2] = {&aio[0], &aio[1]};
   int                i, ret;
```

```
/*
 ** Data initialization and Synchronously writing
*/
 IC_AIO_DATA_INIT(aio[0], fd, "rec#1\n", strlen("rec#1\n"), 0)
 IC_AIO_DATA_INIT(aio[1], fd, "rec#2\n", strlen("rec#2\n"),
 aio[0].aio_nbytes)
 aio[0].aio_lio_opcode = aio[1].aio_lio_opcode = LIO_WRITE;
 ret = lio_listio(LIO_WAIT, aio_list, 2, 0);
 if (ret) return ret;

 return 0;
}/* aio_ex_5 */
```

Result upon execution:

```
-bash-3.00$ ./a.out
-bash-3.00$ cat dat
rec#1
rec#2
-bash-3.00$
```

Remarks:

1. In the example, the `IC_AIO_DATA_INIT` is defined as follows:

```
#define IC_AIO_DATA_INIT(_aio, _fd, _dat, _len, _off)\
  {memset(&_aio, 0, sizeof(_aio)); \
   _aio.aio_fildes = _fd;          \
   _aio.aio_buf    = _dat;         \
   _aio.aio_nbytes = _len;         \
   _aio.aio_offset = _off;}
```

2. The file descriptor `fd` is obtained as:

```
HANDLE fd = CreateFile("dat",
   GENERIC_READ | GENERIC_WRITE,
   FILE_SHARE_READ,
   NULL,
   OPEN_ALWAYS,
   FILE_ATTRIBUTE_NORMAL/*|FILE_FLAG_OVERLAPPED*/,
   NULL);
```

3. The `aio_lio_opcode` refers to the field of each `aiocb` structure that specifies the operation to be performed. The supported operations are `LIO_READ` (do a 'read' operation), `LIO_WRITE` (do a 'write' operation), and `LIO_NOP` (do no operation); these symbols are defined in `<aio.h>`.

See Also

lio_listio()

Asynchronous I/O Function Errors

This topic only applies to Windows* OS.

The `errno` macro is used to obtain the errors that occur during asynchronous request functions such as `aio_read()`, `aio_write()`, `aio_fsync()`, and `lio_listio()` or asynchronous control functions, such as `aio_cancel()`, `aio_error()`, `aio_return()`, and `aio_suspend()`.

The following example illustrates how `errno` can be used.

```
#include <stdio.h>
#include <stdlib.h>
#include <aio.h>
```

```
struct aiocb    my_aio;
struct aiocb   *my_aio_list[1] = {&my_aio};

int main()
{
  int   res;
  double  arr[123456];
 timespec_t   my_t = {1, 0};

/* Data initialization */
 my_aio.aio_fildes = CreateFile("dat",
   GENERIC_READ | GENERIC_WRITE,
   FILE_SHARE_READ,
   NULL,
   OPEN_ALWAYS,
   FILE_ATTRIBUTE_NORMAL,
   NULL);
 my_aio.aio_buf    = (volatile char *)arr;
 my_aio.aio_nbytes = sizeof(arr);

/* Do asynchronous writing with computation overlapping */
 aio_write(&my_aio);
 do_compute(arr, 123456);

/* Suspend the asynchronous writing for 1 sec */
 res = aio_suspend(my_aio_list, 1, &my_t);
 if ( res ) {

/* The call was ended by timeout, before the indicated operations had completed. */
  if ( errno == EAGAIN ) {
  res = aio_suspend(my_aio_list, 1, 0);
  if ( res ) {
  printf("aio_suspend returned non-0\n"); return errno;}
  }
  else
  if ( res ) {
  printf("aio_suspend returned neither 0 nor EAGAIN\n");
  return errno;
  }
 }

 CloseHandle(my_aio.aio_fildes);
 printf("\nPass\n");

 return 0;
}
```

In the example, the program executes an asynchronous write operation, using `aio_write()`, overlapping with some computation, the `do_compute()` function execution. The pending write operation is suspended for one second using `aio_suspend()`.

On successful execution of the asynchronous write operation, zero is returned. `EAGAIN` or any other error value is returned when the call is ended by timeout before the indicated operation has completed.

You can check `EAGAIN` using the `errno` macro.

Intel's C++ Asynchronous I/O Class for Windows

Intel's C++ asynchronous I/O template class, `async_class`, is an implementation for the Windows operating system on IA-32 (for C/C++ only) and Intel® 64 architectures.

The `async_class` template class allows users to perform I/O operations asynchronously to the main program thread. In particular, the `async_class` template class can be used to introduce asynchronous execution of I/O operations with the STL streams classes. Users can quickly switch any of the I/O operations of the STL streams to asynchronous mode with minimal changes to the application code.

The template class `async_class` is defined in the `aiostream.h` file.

See Also

Details of template class async_class

Template Class async_class

This topic only applies to Windows* OS.

Intel's C++ asynchronous I/O class implementation contains two main classes within the `async` namespace: the `async_class` template class and the `thread_control` base class.

The header/typedef definitions are as follows:

```
namespace async {

template<class A>
class async_class:
public thread_control, public A
}
```

The template class `async_class` inherits support for asynchronous execution of I/O operations that are integrated within the base `thread_control` class.

All functionality to control asynchronous execution of a queue of STL stream operations is encapsulated in the base class `thread_control` and is inherited by template class `async_class`.

In most cases it is enough to add the header file `aiostream.h` to the source file and declare the file object as an instance of the new template class `async:async_class`. The initial stream class must be the parameter for the template class. Consequently, the defined output operator << and input operator >> are executed asynchronously.

> **NOTE**
> The header file `aiostream.h` includes all necessary declarations for the STL stream I/O operations to add asynchronous functionality of the `thread_control` class. It also contains the necessary declarations of extensions for the standard C++ STL streams I/O operations: output operator >> and input operator <<.

You can call synchronization method `wait()` to wait for completion of any I/O operations with the file object. If the `wait()` method is not called explicitly, it is called implicitly in the object destructor.

Public Interface of Template Class async_class

The following methods define the public interface of the template class `async_class`:

- `get_last_operation_id()`
- `wait()`
- `get_status()`
- `get_last_error()`

- `get_error_operation_id()`
- `stop_queue()`
- `resume_queue()`
- `clear_queue()`

Library Restrictions

Intel's C++ asynchronous I/O template class does not control the integrity or validity of the objects during asynchronous operation. Such control should be done by the user.

For application stability in the Visual Studio 2003 environment, link the C++ part of `libacaio.lib` library with multi-threaded `msvcrt` run-time library. Use `/MT` or `/MTd` compiler option.

See Also

Example of Using async_class Template Class

get_last_operation_id

Returns ID of the last added operation.

Syntax

```
void get_last_operation_id(void)
```

Description

This method returns the ID of the last added operation. Use this ID to get the status of operation or to wait for the operation to complete.

Return Values

Nothing

wait

Stops execution of current thread.

Syntax

```
int wait(void)
```

```
int wait(unsigned int operation_id)
```

Description

Method `wait(void)` stops execution of the current thread until all the asynchronous operations are completed.

Method `wait(operation_id)` stops execution of the current thread until the operation identified by *operation_id* is completed.

Return Values

-1 : On error during queue execution

Call the `get_last_error()` method to check the error code.

get_status

Returns status of specified operation.

Syntax

```
void get_status(unsigned int operation_id)
```

Description

This method returns the status of an operation, specified by *operation_id,* without stopping current thread execution.

Return Values

STATUS_WAIT: Operation is waiting for execution.

STATUS_COMPLETED: Operation finished execution.

STATUS_ERROR: An error occurred during operation execution.

STATUS_EXECUTE: Operation is executing.

STATUS_BLOCKED: Execution of the queue was blocked after some earlier errors.

get_last_error

Returns the error code of the last failed operation.

Syntax

```
unsigned int get_last_error()
```

Description

This method returns the error code of the last failed operation. If the error occurs during the execution of an asynchronous operation, the asynchronous thread stops executing the queue of asynchronous operations and waits for new user requests.

To obtain the error status, use the `wait()` and `get_status()` methods.

Return Values

Error code of last failed operation.

This error code is equal to the value returned by `GetLastError()` function on the Windows* platform. If the error occurs during the execution of an asynchronous operation, the asynchronous thread stops executing the queue of asynchronous operations and waits for new user requests.

get_error_operation_id

Returns the ID of the last failed operation.

Syntax

```
unsigned int get_error_operation_id()
```

Description

This method returns the ID of the last failed operation. If the error occurs during the execution of an asynchronous operation, the asynchronous thread stops executing the queue of the asynchronous operations and waits for new user requests.

To obtain the error status of the failed operation, use the `wait()` and `get_status()` methods.

Return Values

ID of last failed operation.

stop_queue

Stops queue execution.

Syntax

`int stop_queue()`

Description

This method allows you to control the asynchronous operations queue by stopping queue execution.

Return Values

0: On success

-1: On error

resume_queue

Resumes queue execution.

Syntax

`int resume_queue()`

Description

This method allows you to control the asynchronous operations queue by resuming queue execution.

Return Values

0: On success

-1: On error

clear_queue

Clears stopped or error-interrupted queues.

Syntax

`void push_back_operation(class base_operation*)`

Description

This method clears the content of stopped queues or queues interrupted by errors.

Return Values

0: On success

-1: On error

Example for Using async_class Template Class

The following example illustrates how Intel's C++ asynchronous I/O template class can be used. Consider the following code that writes arrays of floats to an external file.

```
// Data is array of floats
std::vector<float>  v(10000);

// User defines new operator << for std::vector<float> type
std::ofstream& operator << (std::ofstream & str, std::vector<float> & vec)
```

```
{
// User’s output actions
...
 }
...
// Output file declaration - object of standard ofstream STL class
std::ofstream external_file(“output.txt”);
...
// Output operations
external_file << v;
```

The following code illustrates the changes to be made to the above code to execute the output operation asynchronously.

```
// Add new header to support STL asynchronous IO operations

#include <aiostream.h>
...

std::vector<float>  v(10000);

std::ofstream& operator << (std::ofstream & str, std::vector<float> & vec)
{... }
...
// Declare output file as the instance of new async::async_class template
// class.
// New inherited from STL ofstream type is declared
async::async_class<std::ofstream> external_file(“output.txt”);
...
external_file << v;
...
// Add stop operation, to wait the completion of all asynchronous IO //operations
external_file.wait();
…
```

Performance Recommendations

It is recommended not to use asynchronous mode for small objects. For example, do not use asynchronous mode when the output standard type value in a loop where execution of other loop operations takes less time than output of the same value to the STL stream.

However, if you can find the balance between output of small data and its previous calculation inside the loop, you still have some stable performance improvement.

For example, in the following code, the program reads two matrices from external files, calculates the elements of a third matrix, and prints out the elements inside the loop.

```
#define ARR_LEN 900
{
  std::ifstream fA("A.txt");
  fA >> A;
  std::ifstream fB("B.txt");
  fB >> B;
  std::ofstream fC(f);

  for(int i=0; i< ARR_LEN; i++)
   {
    for(int j=0; j< ARR_LEN; j++)
    {
     C[i][j] = 0;
     for(int k=0; k < ARR_LEN; k++)
```

```
      C[i][j]+ = A[i][k]*B[k][j]*sin((float)(k))*cos((float)(-k))*sin((float)(k+1)
      )*cos((float)(-k-1));
      fC << C[i][j] << std::endl;
     }
    }
}
```

By increasing matrix size, you can also achieve performance improvement during parallel data reading from two files.

IEEE 754-2008 Binary Floating-Point Conformance Library

The Intel® IEEE 754-2008 Binary Floating-Point Conformance Library provides all operations mandated by the IEEE 754-2008 standard for binary32 and binary64 binary floating-point interchange formats.

Many routines in the *libbfp754* Library are more optimized for Intel® microprocessors than for non-Intel microprocessors.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Intel® IEEE 754-2008 Binary Floating-Point Conformance Library and Usage

The Intel® IEEE 754-2008 Binary Floating-Point Conformance Library provides all operations mandated by the IEEE 754-2008 standard for binary32 and binary64 binary floating-point interchange formats. The minimum requirements for correct operation of the library are an Intel® Pentium® 4 processor and an operating system supporting Intel® Streaming SIMD Extensions 2 (Intel® SSE2) instructions.

The library supports all four rounding-direction attributes mandated by the IEEE 754-2008 standard for binary floating-point arithmetic: `roundTiesToEven`, `roundTowardPositive`, `roundTowardNegative`, `roundTowardZero`. The additional rounding-direction attribute, `roundTiesToAway`, is not required by the standard, hence, not fully supported in this library. The default rounding-direction attribute is set as `roundTiesToEven`.

The library also supports all mandated exceptions (invalid operation, division by zero, overflow, underflow, and inexact) and sets flags accordingly under default exception handling. Alternate exception handling, which is optional in the standard, is not supported.

The `bfp754.h` header file includes prototypes for the library functions. For a complete list of the functions available, refer to the Function List. The user also needs to specify linker option `-lbfp754` and floating-point semantics control option `-fp-model strict` in order to use the library.

Note: The `libbfp754` library is not available for SYCL.

Many routines in the *libbfp754* Library are more optimized for Intel® microprocessors than for non-Intel microprocessors.

Operations

The IEEE standard 754-2008 defines four types of operations.

1. General-computational operations that produce correctly rounded floating-point or integer results. These operations might signal the floating-point exceptions.
2. Quiet-computational operations that produce floating-point results. These operations do not signal any floating-point exceptions.

3. Signaling-computational operations that produce no floating-point results. These operations might signal floating-point exceptions.
4. Non-computational operations that produce no floating-point results. These operations do not signal floating-point exceptions.

	Produce result	Produce no result
Might signal FP exception	General-computational	Signaling-computational
Do not signal FP exception	Quiet-computational	Non-computational

The standard also distinguishes among operations by their floating-point operand formats and result format for general-computational operations:

1. Homogenous general-computational operations whose floating-point operands and floating-point result are in the same format.
2. *formatOf* general-computational operations whose floating-point operands and floating-point result have different formats.

NOTE
The IEEE 754-2008 standard requires that all *formatOf* general-computational operations be computed without any loss of precision before converting to the destination format. This may differ from how these operations are implemented on most hardware and software.

For example, when all operands are in binary64 format and the destination format is binary32, most hardware and software implementations would first compute an intermediate result rounded in binary64 and then convert the intermediate result to binary32. This double rounding procedure may produce a result different from what is defined in the standard under certain rounding mode. For example: `x = 0x3ff0000010000000 = 1.000000000000000000000001_2`, `y = 0x3ca0000000000000 = 1.0_2*2^(-53) x+y = 1.00000000000000000000000100000000000000000000000000001_2`

When the rounding-direction attribute is set to `roundTiesToEven`, using double rounding procedure, the addition result rounds to `1.000000000000000000000001_2 (0x3ff0000010000000)` in binary64, which would then round to `1 (0x3f800000)` in binary32. On the other hand, according to the standard, the addition result should round to `1.00000000000000000000001_2 (0x3f800001)` in binary32.

Data Types

The following table correlates the names of the formats used in defining operations in the standard with their C99 types used in this library.

Format Name	Definition	C99 Type
`binary32`	IEEE 754-2008 binary32 interchange format	`float`
`binary64`	IEEE 754-2008 binary64 interchange format	`double`
`int`	Integer operand formats	`int, unsigned int, long long int, unsigned long long int`
`int32`	Signed 32-bit integer	`int`

Format Name	Definition	C99 Type
`uint32`	Unsigned 32-bit integer	`unsigned int`
`int64`	Signed 64-bit integer	`long long int`
`uint64`	Unsigned 64-bit integer	`unsigned long long int`
`boolean`	Boolean value represented by generic integer type	`int`
`enum`	Enumerated values of floating-point class	`int`
	Enumerated values of floating-point radix	`int`
`logBFormat`	Type for the destination of the `logB` operation and the scale exponent operand of the `scaleB` operation	`int`
`decimalCharacterSequence`	Decimal character sequence	`char*`
`hexCharacterSequence`	Hexadecimal-significand character sequence	
`exceptionGroup`	Set of exceptions as a set of booleans	`int`
`flags`	Set of status flags	`int`
`binaryRoundingDirection`	Rounding direction for binary	`int`
`modeGroup`	Dynamically-specifiable modes	`int`
`void`	No explicit operand or result	`void`

Use the Intel® IEEE 754-2008 Binary Floating-Point Conformance Library

Many routines in the *libbfp754* Library are more optimized for Intel® microprocessors than for non-Intel microprocessors.

To use the library, include the header file, `bfp754.h`, in your program.

Here is an example program illustrating the use of the library on Linux* OS.

You cannot use these libraries with SYCL kernels.

```
//binary.c
#include <stdio.h>
#include <bfp754.h>
int main(){
  double a64, b64;
  float c32;
  a64 = 1.0000000596046447753906 25;
  b64 = 1.1102230246251565404236316680908203125e-16;
  c32 = __binary32_add_binary64_binary64(a64, b64);
  printf("The addition result using the libary: %8.8f\n", c32);
  c32 = a64 + b64;
```

```
    printf("The addition result without the libary: %8.8f\n", c32);
    return 0;
}
```

To compile `binary.c`, use the command:

```
icx -fp-model source -fp-model except binary.c -lbfp754
```

The output of `a.out` will look similar to the following:

```
The addition result using the libary: 1.00000012
The addition result without the libary: 1.00000000
```

See Also

Function List

Function List

Many routines in the *libbfp754* Library are more optimized for Intel® microprocessors than for non-Intel microprocessors.

The Intel® IEEE 754-2008 Binary Conformance Library supports the following functions for homogeneous general-computational operations:

Function Group	Function	IEEE standard equivalent
Homogeneous General-Computational Operations Functions	ilogb	logB
	maxnum	maxNum
	maxnum_mag	maxNumMag
	minnum	minNum
	minnum_mag	minNumMag
	next_down	nextDown
	next_up	nextUp
	rem	remainder
	round_integral_exact	roundToIntegralExact
	round_integral_nearest_away	roundToIntegralTiesToAway
	round_integral_nearest_even	roundToIntegralTiesToEven
	round_integral_negative	roundToIntegralTowardNegative
	round_integral_positive	roundToIntegralTowardPositive
	round_integral_zero	roundToIntegralTowardZero
	scalbn	scaleB
General-Computational Operations Functions	add	addition
	binary32_to_binary64	convertFormat
	binary64_to_binary32	
	div	division
	fma	fusedMultiplyAdd
	from_int32	convert

Function Group	Function	IEEE standard equivalent
	from_uint32	
	from_int64	
	from_uint64	
	from_hexstring	convertFromHexCharacter
	from_string	convertFromDecimalCharacter
	mul	multiplication
	sqrt	squareRoot
	sub	subtraction
	to_hexstring	convertToHexCharacter
	to_int32_ceil to_uint32_ceil to_int64_ceil to_uint64_ceil	convertToIntegerTowardPositive
	to_int32_floor to_uint32_floor to_int64_floor to_uint64_floor	convertToIntegerTowardNegative
	to_int32_int to_uint32_int to_int64_int to_uint64_int	convertToIntegerTowardZero
	to_int32_rnint to_uint32_rnint to_int64_rnint to_uint64_rnint	convertToIntegerTiesToEven
	to_int32_xrnint to_uint32_xrnint to_int64_xrnint to_uint64_xrnint	convertToIntegerExactTiesToEven
	to_int32_rninta to_uint32_rninta to_int64_rninta to_uint64_rninta	convertToIntegerTiesToAway
	to_int32_xceil to_uint32_xceil	convertToIntegerExactTowardPositive

Function Group	Function	IEEE standard equivalent
	to_int64_xceil	
	to_uint64_xceil	
	to_int32_xfloor	convertToIntegerExactTowardNegative
	to_uint32_xfloor	
	to_int64_xfloor	
	to_uint64_xfloor	
	to_int32_xint	convertToIntegerExactTowardZero
	to_uint32_xint	
	to_int64_xint	
	to_uint64_xint	
	to_int32_xrninta	convertToIntegerExactTiesToAway
	to_uint32_xrninta	
	to_int64_xrninta	
	to_uint64_xrninta	
	to_string	convertToDecimalCharacter
Quiet-Computational Operations Functions	abs	abs
	copy	copy
	copysign	copySign
	negate	negate
Signaling-Computational Operations Functions	quiet_equal	compareQuietEqual
	quiet_greater	compareQuietGreater
	quiet_greater_equal	compareQuietGreaterEqual
	quiet_greater_unordered	compareQuietGreaterUnordered
	quiet_less	compareQuietLess
	quiet_less_equal	compareQuietLessEqual
	quiet_less_unordered	compareQuietLessUnordered
	quiet_not_equal	compareQuietNotEqual
	quiet_not_greater	compareQuietNotGreater
	quiet_not_less	compareQuietNotLess
	quiet_ordered	compareQuietOrdered
	quiet_unordered	compareQuietUnordered
	signaling_equal	compareSignalingEqual
	signaling_greater	compareSignalingGreater
	signaling_greater_equal	compareSignalingGreaterEqual
	signaling_greater_unordered	compareSignalingGreaterUnordered
	signaling_less	compareSignalingLess
	signaling_less_equal	compareSignalingLessEqual
	signaling_less_unordered	compareSignalingLessUnordered

Function Group	Function	IEEE standard equivalent
	signaling_not_equal	compareSignalingNotEqual
	signaling_not_greater	compareSignalingNotGreater
	signaling_not_less	compareSignalingNotLess
Non-Computational Operations Functions	class	class
	defaultMode	defaultModes
	getBinaryRoundingDirection	getBinaryRoundingDirection
	is754version1985	is754version1985
	is754version2008	is754version2008
	isCanonical	isCanonical
	isFinite	isFinite
	isInfinite	isInfinite
	isNaN	isNaN
	isNormal	isNormal
	isSignaling	isSignaling
	isSignMinus	isSignMinus
	isSubnormal	isSubnormal
	isZero	isZero
	lowerFlags	lowerFlags
	radix	radix
	raiseFlags	raiseFlags
	restoreFlags	restoreFlags
	restoreModes	restoreModes
	saveFlags	saveAllFlags
	saveModes	saveModes
	setBinaryRoundingDirection	setBinaryRoundingDirection
	testFlags	testFlags
	testSavedFlags	testSavedFlags
	totalOrder	totalOrder
	totalOrderMag	totalOrderMag

Homogeneous General-Computational Operations Functions

Many routines in the *libbfp754* Library are more optimized for Intel® microprocessors than for non-Intel microprocessors.

The Intel® IEEE 754-2008 Binary Conformance Library supports the following functions for homogeneous general-computational operations:

round_integral_nearest_even

Description: The function rounds floating-point number `x` to its nearest integral value, with the halfway (tied) case rounding to even.

Calling interface:

```
float __binary32_round_integral_nearest_even(float x);
double __binary64_round_integral_nearest_even(double x);
```

round_integral_nearest_away

Description: The function rounds floating-point number `x` to its nearest integral value, with the halfway (tied) case rounding away from zero.

Calling interface:

```
float __binary32_round_integral_nearest_away(float x);
double __binary64_round_integral_nearest_away(double x);
```

round_integral_zero

Description: The function rounds floating-point number `x` to the closest integral value toward zero.

Calling interface:

```
float __binary32_round_integral_zero(float x);
double __binary64_round_integral_zero(double x);
```

round_integral_positive

Description: The function rounds floating-point number `x` to the closest integral value toward positive infinity.

Calling interface:

```
float __binary32_round_integral_positive(float x);
double __binary64_round_integral_positive(double x);
```

round_integral_negative

Description: The function rounds floating-point number `x` to the closest integral value toward negative infinity.

Calling interface:

```
float __binary32_round_integral_negative(float x);
double __binary64_round_integral_negative(double x);
```

round_integral_exact

Description: The function rounds floating-point number `x` to the closest integral value according to the rounding-direction applicable.

Calling interface:

```
float __binary32_round_integral_exact(float x);
double __binary64_round_integral_exact(double x);
```

next_up

Description: The function returns the least floating-point number in the same format as `x` that is greater than `x`.

Calling interface:

```
float __binary32_next_up(float x);
double __binary64_next_up(double x);
```

next_down

Description: The function returns the largest floating-point number in the same format as `x` that is less than `x`.

Calling interface:

```
float __binary32_next_down(float x);
```

```
double __binary64_next_down(double x);
```

rem

Description: The function returns the remainder of `x` and `y`.

Calling interface:

```
float __binary32_rem(float x, float y);
double __binary64_rem(double x, double y);
```

minnum

Description: The function returns the minimal value of `x` and `y`.

Calling interface:

```
float __binary32_minnum(float x, float y);
double __binary64_minnum(double x, double y);
```

maxnum

Description: The function returns the maximal value of `x` and `y`.

Calling interface:

```
float __binary32_maxnum(float x, float y);
double __binary64_maxnum(double x, double y);
```

minnum_mag

Description: The function returns the minimal absolute value of `x` and `y`.

Calling interface:

```
float __binary32_minnum_mag(float x, float y);
double __binary64_minnum_mag(double x, double y);
```

maxnum_mag

Description: The function returns the maximal absolute value of `x` and `y`.

Calling interface:

```
float __binary32_maxnum_mag(float x, float y);
double __binary64_maxnum_mag(double x, double y);
```

scalbn

Description: The function computes x X 2^n for integer value `n`.

Calling interface:

```
float __binary32_scalbn(float x, int n);
double __binary64_scalbn(double x, int n);
```

ilogb

Description: The function returns the exponent part of `x` as integer.

Calling interface:

```
int __binary32_ilogb(float x);
int __binary64_ilogb(double x);
```

General-Computational Operations Functions

Many routines in the *libbfp754* Library are more optimized for Intel® microprocessors than for non-Intel microprocessors.

The Intel® IEEE 754-2008 Binary Conformance Library supports the following functions for *formatOf* general-computational operations:

add

Description: The function computes the addition of two floating-point numbers; the result is then converted to the destination format.

Calling interface:

```
float __binary32_add_binary32_binary32(float x, float y);
float __binary32_add_binary32_binary64(float x, double y);
float __binary32_add_binary64_binary32(double x, float y);
float __binary32_add_binary64_binary64(double x, double y);
double __binary64_add_binary32_binary32(float x, float y);
double __binary64_add_binary32_binary64(float x, double y);
double __binary64_add_binary64_binary32(double x, float y);
double __binary64_add_binary64_binary64(double x, double y);
```

sub

Description: The function computes the subtraction of two floating-point numbers; the result is then converted to the destination format.

Calling interface:

```
float __binary32_sub_binary32_binary32(float x, float y);
float __binary32_sub_binary32_binary64(float x, double y);
float __binary32_sub_binary64_binary32(double x, float y);
float __binary32_sub_binary64_binary64(double x, double y);
double __binary64_sub_binary32_binary32(float x, float y);
double __binary64_sub_binary32_binary64(float x, double y);
double __binary64_sub_binary64_binary32(double x, float y);
double __binary64_sub_binary64_binary64(double x, double y);
```

mul

Description: The function computes the multiplication of two floating-point numbers; the result is then converted to the destination format.

Calling interface:

```
float __binary32_mul_binary32_binary32(float x, float y);
float __binary32_mul_binary32_binary64(float x, double y);
float __binary32_mul_binary64_binary32(double x, float y);
float __binary32_mul_binary64_binary64(double x, double y);
double __binary64_mul_binary32_binary32(float x, float y);
double __binary64_mul_binary32_binary64(float x, double y);
double __binary64_mul_binary64_binary32(double x, float y);
double __binary64_mul_binary64_binary64(double x, double y);
```

div

Description: The function computes the division of two floating-point numbers; the result is then converted to the destination format.

Calling interface:

```
float __binary32_div_binary32_binary32(float x, float y);
float __binary32_div_binary32_binary64(float x, double y);
float __binary32_div_binary64_binary32(double x, float y);
float __binary32_div_binary64_binary64(double x, double y);
double __binary64_div_binary32_binary32(float x, float y);
double __binary64_div_binary32_binary64(float x, double y);
double __binary64_div_binary64_binary32(double x, float y);
double __binary64_div_binary64_binary64(double x, double y);
```

sqrt

Description: The function computes the square root of floating-point number; the result is then converted to the destination format.

Calling interface:

```
float __binary32_sqrt_binary32(float x);
float __binary32_sqrt_binary64(double x);
double __binary32_sqrt_binary32(float x);
double __binary32_sqrt_binary64(double x);
```

fma

Description: The function computes the fused multiply and add of three floating-point numbers *x*, *y*, and *z* as (*x*×*y*) +*z*; the result is then converted to the destination format.

Calling interface:

```
float __binary32_fma_binary32_binary32_binary32(float x, float y, float z);
float __binary32_fma_binary32_binary32_binary64(float x, float y, double z);
float __binary32_fma_binary32_binary64_binary32(float x, double y, float z);
float __binary32_fma_binary32_binary64_binary64(float x, double y, double z);
float __binary32_fma_binary64_binary32_binary32(double x, float y, float z);
float __binary32_fma_binary64_binary32_binary64(double x, float y, double z);
float __binary32_fma_binary64_binary64_binary32(double x, double y, float z);
float __binary32_fma_binary64_binary64_binary64(double x, double y, double z);
double __binary64_fma_binary32_binary32_binary32(float x, float y, float z);
double __binary64_fma_binary32_binary32_binary64(float x, float y, double z);
double __binary64_fma_binary32_binary64_binary32(float x, double y, float z);
double __binary64_fma_binary32_binary64_binary64(float x, double y, double z);
double __binary64_fma_binary64_binary32_binary32(double x, float y, float z);
double __binary64_fma_binary64_binary32_binary64(double x, float y, double z);
double __binary64_fma_binary64_binary64_binary32(double x, double y, float z);
double __binary64_fma_binary64_binary64_binary64(double x, double y, double z);
```

from_int32 / from_uint32 / from_int64 / from_uint64

Description: This function converts integral values in the specified integer format to floating-point number.

Calling interface:

```
float __binary32_from_int32(int n);
double __binary64_from_int32(int n);
```

```
float __binary32_from_uint32(unsigned int n);
double __binary64_from_uint32(unsigned int n);
float __binary32_from_int64(long long int n);
double __binary64_from_int64(long long int n);
float __binary32_from_uint64(unsigned long long int n);
double __binary64_from_uint64(unsigned long long int n);
```

to_int32_rnint / to_uint32_rnint / to_int64_rnint / to_uint64_rnint

Description: This function rounds floating-point number to the nearest integral value in the specified integer format, with halfway cases rounded to even, without signaling the inexact exception.

Calling interface:

```
int __binary32_to_int32_rnint(float x);
int __binary64_to_int32_rnint(double x);
unsigned int __binary32_to_uint32_rnint(float x);
unsigned int __binary64_to_uint32_rnint(double x);
long long int __binary32_to_int64_rnint(float x);
long long int __binary64_to_int64_rnint(double x);
unsigned long long int __binary32_to_uint64_rnint(float x);
unsigned long long int __binary64_to_uint64_rnint(double x);
```

to_int32_int / to_uint32_int / to_int64_int / to_uint64_int

Description: This function rounds floating-point number to the nearest integral value in the specified integer format toward zero, without signaling the inexact exception.

Calling interface:

```
int __binary32_to_int32_int(float x);
int __binary64_to_int32_int(double x);
unsigned int __binary32_to_uint32_int(float x);
unsigned int __binary64_to_uint32_int(double x);
long long int __binary32_to_int64_int(float x);
long long int __binary64_to_int64_int(double x);
unsigned long long int __binary32_to_uint64_int(float x);
unsigned long long int __binary64_to_uint64_int(double x);
```

to_int32_ceil/ to_uint32_ceil / to_int64_ceil / to_uint64_ceil

Description: This function rounds floating-point number to the nearest integral value in the specified integer format toward positive infinity, without signaling the inexact exception.

Calling interface:

```
int __binary32_to_int32_ceil(float x);
int __binary64_to_int32_ceil(double x);
unsigned int __binary32_to_uint32_ceil(float x);
unsigned int __binary64_to_uint32_ceil(double x);
long long int __binary32_to_int64_ceil(float x);
long long int __binary64_to_int64_ceil(double x);
unsigned long long int __binary32_to_uint64_ceil(float x);
unsigned long long int __binary64_to_uint64_ceil(double x);
```

to_int32_floor/ to_uint32_floor / to_int64_floor / to_uint64_floor

Description: This function rounds floating-point number to the nearest integral value in the specified integer format toward negative infinity, without signaling the inexact exception.

Calling interface:

```
int __binary32_to_int32_floor(float x);
int __binary64_to_int32_floor(double x);
unsigned int __binary32_to_uint32_floor(float x);
unsigned int __binary64_to_uint32_floor(double x);
long long int __binary32_to_int64_floor(float x);
long long int __binary64_to_int64_floor(double x);
unsigned long long int __binary32_to_uint64_floor(float x);
unsigned long long int __binary64_to_uint64_floor(double x);
```

to_int32_rninta / to_uint32_rninta / to_int64_rninta / to_uint64_rninta

Description: This function rounds floating-point number to the nearest integral value in the specified integer format, with halfway cases rounded away from zero, without signaling the inexact exception.

Calling interface:

```
int __binary32_to_int32_rninta(float x);
int __binary64_to_int32_rninta(double x);
unsigned int __binary32_to_uint32_rninta(float x);
unsigned int __binary64_to_uint32_rninta(double x);
long long int __binary32_to_int64_rninta(float x);
long long int __binary64_to_int64_rninta(double x);
unsigned long long int __binary32_to_uint64_rninta(float x);
unsigned long long int __binary64_to_uint64_rninta(double x);
```

to_int32_xrnint / to_uint32_xrnint / to_int64_xrnint / to_uint64_xrnint

Description: This function rounds floating-point number to the nearest integral value in the specified integer format, with halfway cases rounded to even, signaling if inexact.

Calling interface:

```
int __binary32_to_int32_xrnint(float x);
int __binary64_to_int32_xrnint(double x);
unsigned int __binary32_to_uint32_xrnint(float x);
unsigned int __binary64_to_uint32_xrnint(double x);
long long int __binary32_to_int64_xrnint(float x);
long long int __binary64_to_int64_xrnint(double x);
unsigned long long int __binary32_to_uint64_xrnint(float x);
unsigned long long int __binary64_to_uint64_xrnint(double x);
```

to_int32_xint / to_uint32_xint / to_int64_xint / to_uint64_xint

Description: This function rounds floating-point number to the nearest integral value in the specified integer format toward zero, signaling if inexact.

Calling interface:

```
int __binary32_to_int32_xint(float x);
int __binary64_to_int32_xint(double x);
unsigned int __binary32_to_uint32_xint(float x);
unsigned int __binary64_to_uint32_xint(double x);
long long int __binary32_to_int64_xint(float x);
```

```
long long int __binary64_to_int64_xint(double x);
unsigned long long int __binary32_to_uint64_xint(float x);
unsigned long long int __binary64_to_uint64_xint(double x);
```

to_int32_xceil / to_uint32_xceil / to_int64_xceil / to_uint64_xceil

Description: This function rounds floating-point number to the nearest integral value in the specified integer format toward positive infinity, signaling if inexact.

Calling interface:

```
int __binary32_to_int32_xceil(float x);
int __binary64_to_int32_xceil(double x);
unsigned int __binary32_to_uint32_xceil(float x);
unsigned int __binary64_to_uint32_xceil(double x);
long long int __binary32_to_int64_xceil(float x);
long long int __binary64_to_int64_xceil(double x);
unsigned long long int __binary32_to_uint64_xceil(float x);
unsigned long long int __binary64_to_uint64_xceil(double x);
```

to_int32_xfloor / to_uint32_xfloor / to_int64_xfloor / to_uint64_xfloor

Description: This function rounds floating-point number to the nearest integral value in the specified integer format toward negative infinity, signaling if inexact.

Calling interface:

```
int __binary32_to_int32_xfloor(float x);
int __binary64_to_int32_xfloor(double x);
unsigned int __binary32_to_uint32_xfloor(float x);
unsigned int __binary64_to_uint32_xfloor(double x);
long long int __binary32_to_int64_xfloor(float x);
long long int __binary64_to_int64_xfloor(double x);
unsigned long long int __binary32_to_uint64_xfloor(float x);
unsigned long long int __binary64_to_uint64_xfloor(double x);
```

to_int32_xrninta / to_uint32_xrninta / to_int64_xrninta / to_uint64_xrninta

Description: This function rounds floating-point number to the nearest integral value in the specified integer format, with halfway cases rounded away from zero, signaling if inexact.

Calling interface:

```
int __binary32_to_int32_xrninta(float x);
int __binary64_to_int32_xrninta(double x);
unsigned int __binary32_to_uint32_xrninta(float x);
unsigned int __binary64_to_uint32_xrninta(double x);
long long int __binary32_to_int64_xrninta(float x);
long long int __binary64_to_int64_xrninta(double x);
unsigned long long int __binary32_to_uint64_xrninta(float x);
unsigned long long int __binary64_to_uint64_xrninta(double x);
```

binary32_to_binary64

Description: This function converts floating-point number in binary32 format to binary64 format.

Calling interface:

```
double __binary32_to_binary64(float x);
```

binary64_to_binary32

Description: This function rounds floating-point number in binary64 format to binary32 format.

Calling interface:

```
float __binary64_to_binary32(double x);
```

from_string

Description: This function converts decimal character sequence to floating-point number.

Calling interface:

```
float __binary32_from_string(char * s);
double __binary64_from_string(char * s);
```

to_string

Description: This function converts floating-point number to decimal character sequence.

Calling interface:

```
void__binary32_to_string(char * s, float x);
void__binary64_to_string(char * s, double x);
```

from_hexstring

Description: This function converts hexadecimal character sequence to floating-point number.

Calling interface:

```
float __binary32_from_hexstring(char * s);
double __binary64_from_hexstring(char * s);
```

to_hexstring

Description: This function converts floating-point number to hexadecimal character sequence.

Calling interface:

```
void__binary32_to_hexstring(cgar * s, float x);
void__binary64_to_hexstring(char * s, double x);
```

Quiet-Computational Operations Functions

Many routines in the *libbfp754* Library are more optimized for Intel® microprocessors than for non-Intel microprocessors.

The Intel® IEEE 754-2008 Binary Conformance Library supports the following functions for quiet-computational operations:

copy

Description: The function copies input floating-point number `x` to output in the same floating-point format, without any change to the sign.

Calling interface:

```
float __binary32_copy(float x);
double __binary64_copy(double x);
```

negate

Description: The function copies input floating-point number `x` to output in the same floating-point format, reversing the sign.

Calling interface:

```
float __binary32_negate(float x);
double __binary64_negate(double x);
```

abs

Description: The function copies input floating-point number `x` to output in the same floating-point format, setting the sign to positive.

Calling interface:

```
float __binary32_abs(float x);
double __binary64_abs(double x);
```

copysign

Description: The function copies input floating-point number `x` to output in the same floating-point format, with the same sign as `y`.

Calling interface:

```
float __binary32_copysign(float x, float y);
double __binary64_copysign(double x, double y);
```

> **NOTE**
> For the listed quiet-computational operations functions, when the first input is a signaling `NaN`, two different outcomes are allowed by the standard. The operation could either signal invalid exception with quieted signaling `NaN` as output, or deliver signaling `NaN` as output without signaling any exception.

Signaling-Computational Operations Functions

Many routines in the *libbfp754* Library are more optimized for Intel® microprocessors than for non-Intel microprocessors.

The Intel® IEEE 754-2008 Binary Conformance Library supports the following functions for signaling-computational operations:

quiet_equal

Description: The function returns 1 (true) if the relation between the two inputs `x` and `y` is equal, returns 0 (false) otherwise. The function signals invalid operation exception when signaling `NaN` is in the inputs.

Calling interface:

```
int __binary32_quiet_equal_binary32 (float x, float y);
int __binary32_quiet_equal_binary64(float x, double y);
int __binary64_quiet_equal_binary32(double x, float y);
int __binary64_quiet_equal_ binary64(double x, double y);
```

quiet_not_equal

Description: The function returns 1 (true) if the relation between the two inputs `x` and `y` is not equal, returns 0 (false) otherwise. The function signals invalid operation exception when signaling `NaN` is one of the inputs.

Calling interface:

```
int __binary32_quiet_not_equal_binary32(float x, float y);
int __binary32_quiet_not_equal_binary64(float x, double y);
```

```
int __binary64_quiet_not_equal_binary32(double x, float y);
int __binary64_quiet_not_equal_binary64(double x, double y);
```

signaling_equal

Description: The function returns 1 (true) if the relation between the two inputs `x` and `y` is equal, returns 0 (false) otherwise. The function signals invalid operation exception when `NaN` is in the inputs.

Calling interface:

```
int __binary32_signaling_equal_binary32(float x, float y);
int __binary32_signaling_equal_binary64(float x, double y);
int __binary64_signaling_equal_binary32(double x, float y);
int __binary64_signaling_equal_binary64(double x, double y);
```

signaling_greater

Description: The function returns 1 (true) if the relation between the two inputs `x` and `y` is greater, returns 0 (false) otherwise. The function signals invalid operation exception when `NaN` is in the inputs.

Calling interface:

```
int __binary32_signaling_greater_binary32(float x, float y);
int __binary32_signaling_greater_binary64(float x, double y);
int __binary64_signaling_greater_binary32(double x, float y);
int __binary64_signaling_greater_binary64(double x, double y);
```

signaling_greater_equal

Description: The function returns 1 (true) if the relation between the two inputs `x` and `y` is greater or equal, returns 0 (false) otherwise. The function signals invalid operation exception when `NaN` is in the inputs.

Calling interface:

```
int __binary32_signaling_greater_equal_binary32(float x, float y);
int __binary32_signaling_greater_equal_binary64(float x, double y);
int __binary64_signaling_greater_equal_binary32(double x, float y);
int __binary64_signaling_greater_equal_binary64(double x, double y);
```

signaling_less

Description: The function returns 1 (true) if the relation between the two inputs `x` and `y` is less, returns 0 (false) otherwise. The function signals invalid operation exception when `NaN` is in the inputs.

Calling interface:

```
int __binary32_signaling_less_binary32(float x, float y);
int __binary32_signaling_less_binary64(float x, double y);
int __binary64_signaling_less_binary32(double x, float y);
int __binary64_signaling_less_binary64(double x, double y);
```

signaling_less_equal

Description: The function returns 1 (true) if the relation between the two inputs `x` and `y` is less or equal, returns 0 (false) otherwise. The function signals invalid operation exception when `NaN` is in the inputs.

Calling interface:

```
int __binary32_signaling_less_equal_binary32(float x, float y);
int __binary32_signaling_less_equal_binary64(float x, double y);
int __binary64_signaling_less_equal_binary32(double x, float y);
int __binary64_signaling_less_equal_binary64(double x, double y);
```

signaling_not_equal

Description: The function returns 1 (true) if the relation between the two inputs `x` and `y` is not equal, returns 0 (false) otherwise. The function signals invalid operation exception when `NaN` is in the inputs.

Calling interface:

```
int __binary32_signaling_not_equal_binary32(float x, float y);
int __binary32_signaling_not_equal_binary64(float x, double y);
int __binary64_signaling_not_equal_binary32(double x, float y);
int __binary64_signaling_not_equal_binary64(double x, double y);
```

signaling_not_greater

Description: The function returns 1 (true) if the relation between the two inputs `x` and `y` is not greater, returns 0 (false) otherwise. The function signals invalid operation exception when `NaN` is in the inputs.

Calling interface:

```
int __binary32_signaling_not_greater_binary32(float x, float y);
int __binary32_signaling_not_greater_binary64(float x, double y);
int __binary64_signaling_not_greater_binary32(double x, float y);
int __binary64_signaling_not_greater_binary64(double x, double y);
```

signaling_less_unordered

Description: The function returns 1 (true) if the relation between the two inputs `x` and `y` is less or unordered, returns 0 (false) otherwise. The function signals invalid operation exception when `NaN` is in the inputs.

Calling interface:

```
int __binary32_signaling_less_unordered_binary32(float x, float y);
int __binary32_signaling_less_unordered_binary64(float x, double y);
int __binary64_signaling_less_unordered_binary32(double x, float y);
int __binary64_signaling_less_unordered_binary64(double x, double y);
```

signaling_not_less

Description: The function returns 1 (true) if the relation between the two inputs `x` and `y` is not less, returns 0 (false) otherwise. The function signals invalid operation exception when `NaN` is in the inputs.

Calling interface:

```
int __binary32_signaling_not_less_ binary32(float x, float y);
int __binary32_signaling_not_less_binary64(float x, double y);
int __binary64_signaling_not_less_binary32(double x, float y);
int __binary64_signaling_not_less_binary64 (double x, double y);
```

signaling_greater_unordered

Description: The function returns 1 (true) if the relation between the two inputs `x` and `y` is greater or unordered, returns 0 (false) otherwise. The function signals invalid operation exception when `NaN` is in the inputs.

Calling interface:

```
int __binary32_signaling_greater_unordered_binary32(float x, float y);
int __binary32_signaling_greater_unordered_binary64(float x, double y);
int __binary64_ signaling_greater_unordered_binary32(double x, float y);
int __binary64_signaling_greater_unordered_binary64(double x, double y);
```

quiet_greater

Description: The function returns 1 (true) if the relation between the two inputs x and y is greater, returns 0 (false) otherwise. The function signals invalid operation exception when signaling NaN is one of the inputs.

Calling interface:

```
int __binary32_quiet_greater_binary32(float x, float y);
int __binary32_quiet_greater_binary64(float x, double y);
int __binary64_quiet_greater_binary32(double x, float y);
int __binary64_quiet_greater_binary64(double x, double y);
```

quiet_greater_equal

Description: The function returns 1 (true) if the relation between the two inputs x and y is greater or equal, returns 0 (false) otherwise. The function signals invalid operation exception when signaling NaN is one of the inputs.

Calling interface:

```
int __binary32_quiet_greater_equal_binary32(float x, float y);
int __binary32_quiet_greater_equal_binary64(float x, double y);
int __binary64_quiet_greater_equal_binary32(double x, float y);
int __binary64_quiet_greater_equal_binary64(double x, double y);
```

quiet_less

Description: The function returns 1 (true) if the relation between the two inputs x and y is less, returns 0 (false) otherwise. The function signals invalid operation exception when signaling NaN is one of the inputs.

Calling interface:

```
int __binary32_quiet_less_binary32(float x, float y);
int __binary32_quiet_less_binary64(float x, double y);
int __binary64_quiet_less_binary32(double x, float y);
int __binary64_quiet_less_binary64(double x, double y);
```

quiet_less_equal

Description: The function returns 1 (true) if the relation between the two inputs x and y is less or equal, returns 0 (false) otherwise. The function signals invalid operation exception when signaling NaN is one of the inputs.

Calling interface:

```
int __binary32_quiet_less_equal_binary32(float x, float y);
int __binary32_quiet_less_equal_binary64(float x, double y)
int __binary64_quiet_less_equal_binary32(double x, float y);
int __binary64_quiet_less_equal_binary64(double x, double y);
```

quiet_unordered

Description: The function returns 1 (true) if the relation between the two inputs x and y is unordered, returns zero (false) otherwise. The function signals invalid operation exception when signaling NaN is one of the inputs

Calling interface:

```
int __binary32_quiet_unordered_binary32(float x, float y);
int __binary32_quiet_unordered_binary64(float x, double y);
int __binary64_quiet_unordered_binary32(double x, float y);
int __binary64_quiet_unordered_binary64(double x, double y);
```

quiet_not_greater

Description: The function returns 1 (true) if the relation between the two inputs x and y is not greater, returns zero (false) otherwise. The function signals invalid operation exception when signaling NaN is one of the inputs.

Calling interface:

```
int __binary32_quiet_not_greater_binary32(float x, float y);
int __binary32_quiet_not_greater_binary64(float x, double y);
int __binary64_quiet_not_greater_binary32(double x, float y);
int __binary64_quiet_not_greater_binary64(double x, double y);
```

quiet_less_unordered

Description: The function returns 1 (true) if the relation between the two inputs x and y is less or unordered, returns 0 (false) otherwise. The function signals invalid operation exception when signaling NaN is one of the inputs.

Calling interface:

```
int __binary32_quiet_less_unordered_binary32(float x, float y);
int __binary32_quiet_less_unordered_binary64(float x, double y);
int __binary64_quiet_less_unordered_binary32(double x, float y);
int __binary64_quiet_less_unordered_binary64(double x, double y);
```

quiet_not_less

Description: The function returns 1 (true) if the relation between the two inputs x and y is not less, returns zero (false) otherwise. The function signals invalid operation exception when signaling NaN is one of the inputs.

Calling interface:

```
int __binary32_quiet_not_less_binary32(float x, float y);
int __binary32_quiet_not_less_binary64(float x, double y);
int __binary64_quiet_not_less_binary32(double x, float y);
int __binary64_quiet_not_less_binary64(double x, double y);
```

quiet_greater_unordered

Description: The function returns 1 (true) if the relation between the two inputs x and y is greater or unordered, returns 0 (false) otherwise. The function signals invalid operation exception when signaling NaN is one of the inputs.

Calling interface:

```
int __binary32_quiet_greater_unordered_binary32(float x, float y);
int __binary32_quiet_greater_unordered_binary64(float x, double y);
int __binary64_quiet_greater_unordered_binary32(double x, float y);
int __binary64_quiet_greater_unordered_binary64(double x, double y);
```

quiet_ordered

Description: The function returns 1 (true) if the relation between the two inputs x and y is ordered, returns 0 (false) otherwise. The function signals invalid operation exception when signaling NaN is one of the inputs.

Calling interface:

```
int __binary32_quiet_ordered_binary32(float x, float y);
int __binary32_quiet_ordered_binary64(float x, double y);
int __binary64_quiet_ordered_binary32(double x, float y);
```

```
int __binary64_quiet_ordered_binary64(double x, double y);
```

Non-Computational Operations Functions

Many routines in the *libbfp754* Library are more optimized for Intel® microprocessors than for non-Intel microprocessors.

The Intel® IEEE 754-2008 Binary Conformance Library supports the following functions for non-computational operations:

is754version1985

Description: The function returns 1, if and only if this programming environment conforms to IEEE Std. 754-1985, otherwise returns 0.

Calling interface:

```
int __binary_is754version1985(void);
```

> **NOTE**
> This function in this library always returns 0.

is754version2008

Description: The function returns 1, if and only if this programming environment conforms to IEEE Std. 754-2008, otherwise returns 0.

Calling interface:

```
int __binary_is754version2008(void);
```

> **NOTE**
> This function in this library always returns 1.

class

Description: The function returns which class of the ten classes (`signalingNaN`, `quietNaN`, `negativeInfinity`, `negativeNormal`, `negativeSubnormal`, `negativeZero`, `positiveZero`, `positiveSubnormal`, `positiveNormal`, `positiveInfinity`) the input floating-point number `x` belongs.

Return value	Class
0	`signalingNaN`
1	`quietNaN`
2	`negativeInfinity`
3	`negativeNormal`
4	`negativeSubnormal`
5	`negativeZero`
6	`positiveZero`
7	`positiveSubnormal`

Return value	Class
8	`positiveNormal`
9	`positiveInfinity`

Calling interface:

```
int __binary32_class(float x);
int __binary64_class(double x);
```

isSignMinus

Description: The function returns 1, if and only if its argument has negative sign.

Calling interface:

```
int __binary32_isSignMinus(float x);
int __binary64_isSignMinus(double x);
```

isNormal

Description: The function returns 1, if and only if its argument is normal (not `zero`, `subnormal`, `infinite`, or `NaN`).

Calling interface:

```
int __binary32_isNormal(float x);
int __binary64_isNormal(double x);
```

isFinite

Description: The function returns 1, if and only if its argument is `finite` (not `infinite` or `NaN`).

Calling interface:

isZero

Description: The function returns 1, if and only if its argument is ±0.

Calling interface:

```
int __binary32_isZero(float x);
int __binary64_isZero(double x);
```

isSubnormal

Description: The function returns 1, if and only if its argument is `subnormal`.

Calling interface:

```
int __binary32_isSubnormal(float x);
int __binary64_isSubnormal(double x);
```

isInfinite

Description: The function returns 1, if and only if its argument is `infinite`

Calling interface:

```
int __binary32_isInfinite(float x);
int __binary64_isInfinite(double x);
```

isNaN

Description: The function returns 1, if and only if its argument is a `NaN`.

Calling interface:

```
int __binary32_isNaN(float x);
int __binary64_isNaN(double x);
```

isSignaling

Description: The function returns 1, if and only if its argument is a signaling `NaN`.

Calling interface:

```
int __binary32_isSignaling(float x);
int __binary64_isSignaling(double x);
```

isCanonical

Description: The function returns 1, if and only if its argument is a finite number, `infinity`, or `NaN` that is canonical.

Calling interface:

```
int __binary32_isCanonical(float x);
int __binary64_isCanonical(double x);
```

NOTE
This function in this library always returns 1, as only canonical floating-point numbers are expected.

radix

Description: The function returns the radix of the format of the input floating-point number.

Calling interface:

```
int __binary32_radix(float x);
int __binary64_radix(double x);
```

NOTE
This function in this library always returns 2, as the library is intended for binary floating-point numbers.

totalOrder

Description: The function returns 1 if and only if two floating-point inputs `x` and `y` is total ordered and 0 otherwise.

Calling interface:

```
int _binary32_totalOrder(float x, float y);
int _binary64_totalOrder(double x, double y);
```

totalOrderMag

Description:`totalOrderMag(x, y)` is the same as `totalOrder(abs(x), abs(y))`.

Calling interface:

```
int _binary32_totalOrderMag(float x, float y);
int _binary64_totalOrderMag(double x, double y);
```

lowerFlags

Description: The function lowers the flags of the exception group specified by the input.

Value	Exception name
1	`__BFP754_INVALID`
2	`__BFP754_DIVBYZERO`
4	`__BFP754_OVERFLOW`
8	`__BFP754_UNDERFLOW`
16	`__BFP754_INEXACT`

Calling interface:

```
void __binary_lowerFlags(int x);
```

raiseFlags

Description: The function raises the flags of the exception group specified by the input.

Calling interface:

```
void __binary_raiseFlags(int x);
```

testFlags

Description: The function returns 1, if and only if any flag of the exception group specified by the input is raised, and 0 otherwise.

Calling interface:

```
int __binary_testFlags(int x);
```

testSavedFlags

Description: The function returns 1, if and only if any flag of the exception group specified by the input `y` is raised in `x`, and 0 otherwise.

Calling interface:

```
int __binary_testSavedFlags(int x, int y);
```

restoreFlags

Description: The function restores the flags to their states represented in `x`.

Calling interface:

```
void __binary_restoreFlags(int x);
```

saveFlags

Description: The function returns a representation of the state of all status flags.

Calling interface:

```
int __binary_saveFlags(void);
```

getBinaryRoundingDirection

Description: The function returns an integer representing the rounding direction in use.

Value	Exception name
0	`__BFP754_ROUND_TO_NEAREST_EVEN`
1	`__BFP754_ROUND_TOWARD_POSITIVE`
2	`__BFP754_ROUND_TOWARD_NEGATIVE`
3	`__BFP754_ROUND_TOWARD_ZERO`

Calling interface:

`int __binary_getBinaryRoundingDirection(void);`

setBinaryRoundingDirection

Description: The function sets the rounding direction based on input integer.

Calling interface:

`void __binary_setBinaryRoundingDirection(int x);`

saveModes

Description: The function saves the values of all dynamic-specifiable modes.

Calling interface:

`int __binary_saveModes(void);`

> **NOTE**
> `saveModes` behaves in the same way as `getBinaryRoundingDirection` does, as the rounding mode is the only dynamic-specifiable mode supported.

restoreModes

Description: The function restores the values of all dynamic-specifiable modes to the input.

Calling interface:

`int __binary_restoreModes(void);`

> **NOTE**
> `restoreModes` behaves in the same way as `setBinaryRoundingDirection` does, as the rounding mode is the only dynamic-specifiable mode supported.

defaultMode

Description: The function sets the values of all dynamic-specifiable modes to default.

Calling interface:

`void __binary_defaultMode(void);`

> **NOTE**
> `defaultMode` sets the rounding-direction attribute to `roundTiesToEven`, as the rounding mode is the only dynamic-specifiable mode supported.

Intel's Numeric String Conversion Library

Intel's Numeric String Conversion Library, `libistrconv`, provides a collection of routines for converting between ASCII strings and C data types, which are optimized for performance.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Use Intel's Numeric String Conversion Library

Intel's Numeric String Conversion Library, `libistrconv`, provides a collection of routines for converting between ASCII strings and C data types, which are optimized for performance. The `istrconv.h` header file declares prototypes for the library functions.

You can link the `libistrconv` library as a static or shared library on Linux* platforms.On Windows* platforms, you must link `libistrconv` as a static library only.

Using Intel's Numeric String Conversion Library

To use the `libistrconv` library, include the header file, `istrconv.h`, in your program.

Consider the following example `conv.c` file that illustrates how to use the library to convert between string and floating-point data type.

```
// conv.c
#include <stdio.h>
#include <istrconv.h>
#define LENGTH 20

int main() {
 const char pi[] = "3.14159265358979323";
 char s[LENGTH];
 int prec;
 float fx;
 double dx;
 printf("PI: %s\n", pi);
 printf("single-precision\n");
 fx = __IML_string_to_float(pi, NULL);
 prec = 6;
 __IML_float_to_string(s, LENGTH, prec, fx);
 printf("prec: %2d, val: %s\n", prec, s);
 printf("double-precision\n");
 dx = __IML_string_to_double(pi, NULL);
 prec = 15;
 __IML_double_to_string(s, LENGTH, prec, dx);
 printf("prec: %2d, val: %s\n", prec, s);
 return 0;
}
```

To compile the `conv.c` file with Intel's Numeric String Conversion Library (`libistrconv`) use one of the following commands. See Invoke the Compiler for information about all available compilers and drivers.

Linux

```
icpx conv.c -libistrconv
```

Windows

```
icx conv.c libistrconv.lib
```

After you compile this example and run the program, you should get the following results:

```
PI: 3.14159265358979323

single-precision
prec: 6, val: 3.14159

double-precision
prec: 15, val: 3.14159265358979
```

Integer Conversion Functions Optimized with SSE4.2 Instructions

The following integer conversion functions are optimized for better performance with SSE4.2 string processing instructions:

- `__IML_int_to_string`
- `__IML_uint_to_string`
- `__IML_int64_to_string`
- `__IML_uint64_to_string`
- `__IML_i_to_str`
- `__IML_u_to_str`
- `__IML_ll_to_str`
- `__IML_ull_to_str`
- `__IML_string_to_int`
- `__IML_string_to_uint`
- `__IML_string_to_int64`
- `__IML_string_to_uint64`
- `__IML_str_to_i`
- `__IML_str_to_u`
- `__IML_str_to_ll`
- `__IML_str_to_ull`

The SSE4.2 optimized versions of these functions can be deployed in the following situations:

- Used automatically on post-SSE4.2 processors through Intel run-time processor dispatching
- Called directly by defining the "__SSE4_2__" macro to the C preprocessor where <istrconv.h> is included.

The generic versions of these functions can be deployed in the following situations:

- Used automatically on pre-SSE4.2 processors through Intel run-time processor dispatching
- Called directly by adding _generic suffix to the function names

The SSE4.2 optimized versions of these functions moves strings from memory to XMM registers and vice versa directly to maximize performance. The functions would not overwrite the memory beyond the boundary; however, this may introduce memory access violation when the memory location immediately trailing the strings is not allocated or accessible. Users with concerns about potential memory access violation should use the generic versions instead.

Function List

Intel's Numeric String Conversion library (`libistrconv`) functions are listed in this topic.

Routines to Convert Floating-point Numbers to ASCII Strings

Intel's Numeric String Conversion Library supports the following functions to convert floating-point number *x* to string *s* in various formats, where *l* represents the length of the formatted string allowing for full conversion (not including the null terminator).

`__IML_float_to_string, __IML_double_to_string`

Description: These functions are similar to `snprintf(s, n, "%.*g", p, x)` in `stdio.h`, where *p* specifies the maximum number of significant digits in either fixed-point or exponential notation format. If *n* is zero, nothing is written and *s* may be a null pointer. Output characters beyond the $(n-1)^{th}$ character are discarded and a null character is appended at the end. *l* is returned on success; otherwise the result is undefined.

Calling interface:

```
int __IML_float_to_string(char * s, size_t n, int p, float x);
int __IML_double_to_string(char * s, size_t n, int p, double x);
```

`__IML_float_to_string_f, __IML_double_to_string_f`

Description: These functions are similar to `snprintf(s, n, "%.*f", p, x)` in `stdio.h`, where *p* specifies the number of digits after the decimal point in the fixed-point notation format. If *n* is zero, nothing is written and *s* may be a null pointer. Output characters beyond the $(n-1)^{th}$ character are discarded and a null character is appended at the end. *l* is returned on success; otherwise the result is undefined.

Calling interface:

```
int __IML_float_to_string_f(char * s, size_t n, int p, float x);
int __IML_double_to_string_f(char * s, size_t n, int p, double x);
```

`__IML_float_to_string_e, __IML_double_to_string_e`

Description: These functions are similar to `snprintf(s, n, "%.*e", p, x)` in `stdio.h`, where *p* specifies the number of digits after the decimal point in the exponential notation format. If *n* is zero, nothing is written and *s* may be a null pointer. Output characters beyond the $(n-1)^{th}$ character are discarded and a null character is appended at the end. *l* is returned on success; otherwise, the result is undefined.

Calling interface:

```
int __IML_float_to_string_e(char * s, size_t n, int p, float x);
int __IML_double_to_string_e(char * s, size_t n, int p, double x);
```

`__IML_f_to_str, __IML_d_to_str`

Description: These functions are similar to `snprintf(s, n, “%.*g”, p, x)` in `stdio.h`, where *p* specifies the maximum number of significant digits in either fixed-point or exponential notation format. If $l < n$, all output characters are stored in *s* with a null terminator at the end. Otherwise, output characters beyond the n^{th} character are discarded and no null character is appended at the end. If *n* is zero, nothing is written and *s* may be a null pointer. *l* is returned on success; otherwise the result is undefined.

Calling interface:

```
int __IML_f_to_str(char * s, size_t n, int p, float x);
int __IML_d_to_str(char * s, size_t n, int p, double x);
```

`__IML_f_to_str_f, __IML_d_to_str_f`

Description: These functions are similar to `snprintf(s, n, "%.*f", p, x)` in `stdio.h`, where *p* specifies the number of digits after the decimal point in the fixed-point notation format. If *l* < *n*, all output characters are stored in *s* with a null terminator at the end. Otherwise, output characters beyond the n^{th} character are discarded and no null character is appended at the end. If *n* is zero, nothing is written and *s* may be a null pointer. *l* is returned on success; otherwise the result is undefined.

Calling interface:

```
int __IML_f_to_str_f(char * s, size_t n, int p, float x);
int __IML_d_to_str_f(char * s, size_t n, int p, double x);
```

`__IML_f_to_str_e, __IML_d_to_str_e`

Description: These functions are similar to `snprintf(s, n, "%.*e", p, x)` in `stdio.h`, where *p*specifies the number of digits after the decimal point in the exponential notation format. If *l* < *n*, all output characters are stored in *s* with a null terminator at the end. Otherwise, output characters beyond the n^{th} character are discarded and no null character is appended at the end. If *n* is zero, nothing is written and *s* may be a null pointer. *l* is returned on success; otherwise the result is undefined.

Calling interface:

```
int __IML_f_to_str_e(char * s, size_t n, int p, float x);
int __IML_d_to_str_e(char * s, size_t n, int p, double x);
```

Routines to Convert Integers to ASCII Strings

Intel's Numeric String Conversion Library supports the following functions to convert integer *x* to string *s*, where *l* represents the length of the formatted string allowing for full conversion (not including the null terminator).

`__IML_int_to_string, __IML_uint_to_string, __IML_int64_to_string, __IML_uint64_to_string`

Description: These functions are similar to `snprintf(s, n, "%[d|u|lld|llu]", x)` in `stdio.h`. If *n* is zero, nothing is written and *s* may be a null pointer. Output characters beyond the $(n-1)^{th}$ character are discarded and a null character is appended at the end. *l* is returned on success; otherwise the result is undefined.

Calling interface:

```
int __IML_int_to_string(char * s, size_t n, int x);
int __IML_uint_to_string(char * s, size_t n, unsigned int x);
int __IML_int64_to_string(char * s, size_t n, long long x);
int __IML_uint64_to_string(char * s, size_t n, unsigned long long x);
```

`__IML_int_to_oct_string, __IML_uint_to_oct_string, __IML_int64_to_oct_string, __IML_uint64_to_oct_string`

Description: These functions are similar to `snprintf(s, n, "%[o|llo]", x)` in `stdio.h`. If *n* is zero, nothing is written and *s* may be a null pointer. Output characters beyond the $(n-1)^{th}$ character are discarded and a null character is appended at the end. *l* is returned on success; otherwise the result is undefined.

Calling interface:

```
int __IML_int_to_oct_string(char * s, size_t n, int x);
int __IML_uint_to_oct_string(char * s, size_t n, unsigned int x);
int __IML_int64_to_oct_string(char * s, size_t n, long long x);
int __IML_uint64_to_oct_string(char * s, size_t n, unsigned long long x);
```

`__IML_int_to_hex_string`, `__IML_uint_to_hex_string`, `__IML_int64_to_hex_string`, `__IML_uint64_to_hex_string`

Description: These functions are similar to `snprintf(s, n, "%[x|llx]", x)` in `stdio.h`. If *n* is zero, nothing is written and *s* may be a null pointer. Output characters beyond the *(n-1)*th character are discarded and a null character is appended at the end. *l* is returned on success; otherwise the result is undefined.

Calling interface:

```
int __IML_int_to_hex_string(char * s, size_t n, int x);
int __IML_uint_to_hex_string(char * s, size_t n, unsigned int x);
int __IML_int64_to_hex_string(char * s, size_t n, long long x);
int __IML_uint64_to_hex_string(char * s, size_t n, unsigned long long x);
```

`__IML_i_to_str`, `__IML_u_to_str`, `__IML_ll_to_str`, `__IML_ull_to_str`

Description: These functions are similar to `snprintf(s, n, "%[d|u|lld|llu]", x)` in `stdio.h`. If *l* < *n*, all output characters are stored in *s* with a null terminator at the end. Otherwise, output characters beyond the n^{th} character are discarded and no null character is appended at the end. If *n* is zero, nothing is written, and *s* may be a null pointer. *l* is returned on success, otherwise the result is undefined.

Calling interface:

```
int __IML_i_to_str(char * s, size_t n, int x);
int __IML_u_to_str(char * s, size_t n, unsigned int x);
int __IML_ll_to_str(char * s, size_t n, long long x);
int __IML_ull_to_str(char * s, size_t n, unsigned long long x);
```

`__IML_i_to_oct_str`, `__IML_u_to_oct_str`, `__IML_ll_to_oct_str`, `__IML_ull_to_oct_str`

Description: These functions are similar to `snprintf(s, n, "%[o|llo]", x)` in `stdio.h`. If *l* < *n*, all output characters are stored in *s* with a null terminator at the end. Otherwise, output characters beyond the n^{th} character are discarded and no null character is appended at the end. If *n* is zero, nothing is written, and *s* may be a null pointer. *l* is returned on success, otherwise the result is undefined.

Calling interface:

```
int __IML_i_to_oct_str(char * s, size_t n, int x);
int __IML_u_to_oct_str(char * s, size_t n, unsigned int x);
int __IML_ll_to_oct_str(char * s, size_t n, long long x);
int __IML_ull_to_oct_str(char * s, size_t n, unsigned long long x);
```

`__IML_i_to_hex_str`, `__IML_u_to_hex_str`, `__IML_ll_to_hex_str`, `__IML_ull_to_hex_str`

Description: These functions are similar to `snprintf(s, n, "%[x|llx]", x)` in `stdio.h`. If *l* < *n*, all output characters are stored in *s* with a null terminator at the end. Otherwise, output characters beyond the n^{th} character are discarded and no null character is appended at the end. If *n* is zero, nothing is written, and *s* may be a null pointer. *l* is returned on success, otherwise the result is undefined.

Calling interface:

```
int __IML_i_to_hex_str(char * s, size_t n, int x);
int __IML_u_to_hex_str(char * s, size_t n, unsigned int x);
int __IML_ll_to_hex_str(char * s, size_t n, long long x);
int __IML_ull_to_hex_str(char * s, size_t n, unsigned long long x);
```

Routines to Convert ASCII Strings to Floating-point Numbers

Intel's Numeric String Conversion Library supports the following functions to convert the initial portion of decimal string *s* to floating-point number *x*. If no conversion could be performed, zero is returned. If the correct value is outside the range of the return type, plus (+) or minus (-) `HUGE_VALF`, `HUGE_VAL`, or `HUGE_VALL` is returned, and the value of macro `ERANGE` is stored in `errno`.

`__IML_string_to_float`, `__IML_string_to_double`, `__IML_string_to_long_double`

Description: These functions are similar to `strtof(nptr, endptr)`, `strtod(nptr, endptr)`, and `strtold(nptr, endptr)` in `stdlib.h`, where *endptr* points to the object that stores the final part of *nptr* when *endptr* is not a null pointer.

Calling interface:

```
float __IML_string_to_float(const char * nptr, char ** endptr);

double __IML_string_to_double(const char * nptr, char ** endptr);

long double __IML_string_to_long_double(const char * nptr, char ** endptr);
```

`__IML_str_to_f`, `__IML_str_to_d`, `__IML_str_to_ld`

Description: These functions convert the initial *n* decimal digits of the *significand* string multiplied by 10 raised to power of *exponent* to floating-point number as return. *endptr* points to the object that stores the final part of significand, provided that *endptr* is not a null pointer.

Calling interface:

```
float __IML_str_to_f(const char * significand, size_t n, int exponent, char ** endptr);

double __IML_str_to_d(const char * significand, size_t n, int exponent, char **
endptr);

long double __IML_str_to_ld(const char * significand, size_t n, int exponent, char **
endptr);
```

Routines to Convert ASCII Strings to Integers

Intel's Numeric String Conversion Library supports the following functions to convert the initial portion of string *s* to integer *x*. If no conversion could be performed, zero is returned. If the correct value is outside the range of the return type, `INT_MIN`, `INT_MAX`, `UINT_MAX`, `LLONG_MIN`, `LLONG_MAX`, `ULLONG_MAX` is returned, and the value of macro `ERANGE` is stored in `errno`.

`__IML_string_to_int`, `__IML_string_to_uint`, `__IML_string_to_int64`, `__IML_string_to_uint64`

Description: These functions are similar to `([unsigned] int)strto[u]l(nptr, endptr, 10)` and `strto[u]ll(nptr, endptr, 10)` functions in `stdlib.h`, where *endptr* points to the object that stores the final part of *nptr* when *endptr* is not a null pointer.

Calling interface:

```
int __IML_string_to_int(const char * nptr, char ** endptr);

unsigned int __IML_string_to_uint(const char * nptr, char ** endptr);

long long __IML_string_to_int64(const char * nptr, char ** endptr);

unsigned long long __IML_string_to_uint64(const char * nptr, char ** endptr);
```

`__IML_oct_string_to_int`, `__IML_oct_string_to_uint`, `__IML_oct_string_to_int64`, `__IML_oct_string_to_uint64`

Description: These functions are similar to `([unsigned] int)strto[u]l(nptr, endptr, 8)` and `strto[u]ll(nptr, endptr, 8)` functions in `stdlib.h`, where *endptr* points to the object that stores the final part of *nptr* when *endptr* is not a null pointer.

Calling interface:

```
int __IML_oct_string_to_int(const char * nptr,char ** endptr);
unsigned int __IML_oct_string_to_uint(const char * nptr,char ** endptr);
long long __IML_oct_string_to_int64(const char * nptr,char ** endptr);
unsigned long long __IML_oct_string_to_uint64(const char * nptr,char ** endptr);
```

`__IML_hex_string_to_int, __IML_hex_string_to_uint, __IML_hex_string_to_int64, __IML_hex_string_to_uint64`

Description: These functions are similar to `([unsigned] int)strto[u]l(nptr, endptr, 16)` and `strto[u]ll(nptr, endptr, 16)` functions in `stdlib.h`, where *endptr* points to the object that stores the final part of *nptr* when *endptr* is not a null pointer.

Calling interface:

```
int __IML_hex_string_to_int(const char * nptr,char ** endptr);
unsigned int __IML_hex_string_to_uint(const char * nptr,char ** endptr);
long long __IML_hex_string_to_int64(const char * nptr,char ** endptr);
unsigned long long __IML_hex_string_to_uint64(const char * nptr,char ** endptr);
```

`__IML_str_to_i, __IML_str_to_u, __IML_str_to_ll, __IML_str_to_ull`

Description: These functions convert the initial *n* decimal digits (including an optional + or - sign) pointed to by *nptr* to integral values. When *endptr* is not a null pointer it points to the object that stores the final part of *nptr*. These functions treat any leading whitespace as invalid.

Calling interface:

```
int __IML_str_to_i(const char * nptr, size_t n, char ** endptr);
unsigned int __IML_str_to_u(const char * nptr, size_t n, char ** endptr);
long long __IML_str_to_ll(const char * nptr, size_t n, char ** endptr);
unsigned long long __IML_str_to_ull(const char * nptr, size_t n, char ** endptr);
```

`__IML_oct_str_to_i, __IML_oct_str_to_u, __IML_oct_str_to_ll, __IML_oct_str_to_ull`

Description: These functions convert the initial *n* octal digits (including an optional + or - sign) pointed to by *nptr* to integral values. When *endptr* is not a null pointer it points to the object that stores the final part of *nptr*. These functions treat any leading whitespace as invalid.

Calling interface:

```
int __IML_oct_str_to_i(const char * nptr,size_t n,char ** endptr);
unsigned int __IML_oct_str_to_u(const char * nptr,size_t n,char ** endptr);
long long __IML_oct_str_to_ll(const char * nptr,size_t n,char ** endptr);
unsigned long long __IML_oct_str_to_ull(const char * nptr,size_t n,char ** endptr);
```

`__IML_hex_str_to_i, __IML_hex_str_to_u, __IML_hex_str_to_ll, __IML_hex_str_to_ull`

Description: These functions convert the initial *n* hexadecimal digits (including an optional + or - sign) pointed to by *nptr* to integral values. When *endptr* is not a null pointer it points to the object that stores the final part of *nptr*. These functions treat any leading whitespace as invalid.

Calling interface:

```
int __IML_hex_str_to_i(const char * nptr,size_t n,char ** endptr);
unsigned int __IML_hex_str_to_u(const char * nptr,size_t n,char ** endptr);
long long __IML_hex_str_to_ll(const char * nptr,size_t n,char ** endptr);
unsigned long long __IML_hex_str_to_ull(const char * nptr,size_t n,char ** endptr);
```

Macros

The Intel® oneAPI DPC++/C++ Compiler supports the ISO Standard predefined macros and additional predefined macros.

ISO Standard Predefined Macros

The ISO/ANSI standard for the C language requires that certain predefined macros be supplied with conforming compilers.

The compiler includes predefined macros in addition to those required by the standard. The default predefined macros differ among Windows*, Linux* operating systems. Differences also exist on Linux as a result of the `-std` compiler option.

The following table lists the macros that the Intel® oneAPI DPC++/C++ Compiler supplies in accordance with this standard:

Macro	Value
`__DATE__`	The date of compilation as an 11-character string literal in the form `mm dd yyyy`. If the day is less than 10 characters, a space is added before the day value.
`__FILE__`	A string literal representing the name of the file being compiled.
`__LINE__`	The current line number as a decimal constant.
`__STDC_HOSTED__`	Defined and value is 1 only when compiling a C translation unit with `/Qstd=c99`.
`__STDC_VERSION__`	Defined and value is 199901L only when compiling a C translation unit with `/Qstd=c99`.
`__TIME__`	The time of compilation as a string literal in the form `hh:mm:ss`.

See Also

Additional Predefined Macros

Additional Predefined Macros

The compiler includes predefined macros specified by the ISO/ANSI standard and it also supports the predefined macros listed in the table below.

Unless otherwise stated, the macros are supported on systems based on IA-32 (C/C++ only) and Intel® 64 architectures.

Macro	Description
`__AVX__`	On Linux, defined as '1' when option `-march=corei7-avx`, or higher processor targeting options are specified. **NOTE** Available only for compilations targeting Intel® 64 architecture.
`__AVX2__` (Linux)	On Linux, defined as '1' when option `-march=core-avx2`, or higher processor targeting options are specified. **NOTE** Available only for compilations targeting Intel® 64 architecture.
`__AVX512BW__` (Windows, Linux)	Defined as '1' for processors that support Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Byte and Word Instructions (BWI).
`__AVX512CD__` (Windows, Linux)	Defined as '1' for processors that support Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Conflict Detection Instructions (CDI).
`__AVX512DQ__` (Windows, Linux)	Defined as '1' for processors that support Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Doubleword and Quadword Instructions (DQI).
`__AVX512ER__` (Windows, Linux)	Defined as '1' for processors that support Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Exponential and Reciprocal Instructions.
`__AVX512F__` (Windows, Linux)	Defined as '1' for processors that support Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Foundation instructions.
`__AVX512PF__` (Windows, Linux)	Defined as '1' for processors that support Intel® Advanced Vector Extensions 512 (Intel® AVX-512) PreFetch Instructions (PFI).
`__AVX512VL__` (Windows, Linux)	Defined as '1' for processors that support Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Vector Length Extensions (VLE).
`__BASE_FILE__` (Linux)	Name of source file
`__COUNTER__` (Windows)	Defined as '0'.
`__cplusplus` (Linux)	Defined as '1' (for the Intel® oneAPI DPC++/C++ Compiler).
`__ELF__` (Linux)	Defined as '1' at the start of compilation.

Macro	Description
`__EXCEPTIONS` (Linux)	Defined as '1' when option `fno-exceptions` is *not* used.
`__gnu_linux__` (Linux)	Defined as '1' at the start of compilation.
`__GNUC__` (Linux)	The major version number of GCC installed on the system.
`__GNUC_MINOR__` (Linux)	The minor version number of GCC or G++ installed on the system.
`__GNUC_PATCHLEVEL__` (Linux)	The patch level version number of GCC or G++ installed on the system.
`__GNUG__` (Linux)	The major version number of G++ installed on the system.
`__i386__` `__i386` `i386` (Linux)	Defined as '1' for compilations targeting IA-32 architecture (C++ only).
`_INTEGRAL_MAX_BITS` (Windows)	64
`__INTEL_LLVM_COMPILER` (Windows, Linux)	The version of the compiler in the form VVVVMMUU , where *VVVV* is the major release version, *MM* is the minor release version, and *UU* is the update number. For example, the base release of 2021.1 is represented by the value 20210100. This symbol is also recognized by CMake. **NOTE** To identify the Intel® oneAPI DPC++/C++ Compiler, you must check for the existence of both __INTEL_LLVM_COMPILER and SYCL_LANGUAGE_VERSION, where SYCL_LANGUAGE_VERSION is part of the SYCL spec.
`__INTEL_MS_COMPAT_LEVEL` (Windows)	Defined as '1'.
`__LIBSYCL_MAJOR_VERSION`	Used to set the SYCL runtime library major version.
`__LIBSYCL_MINOR_VERSION`	Used to set the SYCL runtime library minor version.
`__LIBSYCL_PATCH_VERSION`	Used to set the SYCL runtime library patch version.
`__linux__` `__linux` `linux`	Defined as '1' at the start of compilation.

Macro	Description
(Linux)	
__LONG_DOUBLE_SIZE__ (Windows, Linux)	On Linux, defined as 80. On Windows, defined as 64; defined as 80 when option /Qlong-double is specified.
__LONG_MAX__ (Linux)	9223372036854775807L **NOTE** Available only for compilations targeting Intel® 64 architecture.
__LP64__ (Linux) __LP64 (Linux)	Defined as '1'. **NOTE** Available only for compilations targeting Intel® 64 architecture.
_M_IX86 (Windows)	700
_M_X64 (Windows)	Defined as '1' while building code targeting Intel® 64 architecture.
__MMX__ (Linux)	Defined as '1'. On Linux, it is available only on systems based on Intel® 64 architecture.
_MSC_EXTENSIONS (Windows)	This macro is defined when Microsoft extensions are enabled.
_MSC_FULL_VER (Windows)	The Visual C++ version being used.
_MSC_VER (Windows)	The Visual C++ version being used.
_MT (Windows)	On Windows, defined as '1' when a multithreaded delay-locked loop (DLL) or library is used (when option /MD[d] or /MT[d] is specified).
__NO_MATH_INLINES __NO_STRING_INLINES (Linux)	Defined as '1'.
_OPENMP (Windows, Linux)	201611 when you specify option [Q]openmp.
__OPTIMIZE__ (Linux)	Defined as '1'.
__pentium4	Defined as '1'.

Macro	Description
`__pentium4__` (Linux)	
`__PIC__` `__pic__` (Linux)	On Linux, defined as '1' when option `fPIC` is specified.
`__PTRDIFF_TYPE__` (Linux)	On Linux, defined as int on IA-32 architecture (C++ only); defined as long on Intel® 64 architecture.
`__QMSPP_` (Windows)	Defined as '1'.
`__REGISTER_PREFIX__` (Linux)	
RESTRICT_WRITE_ACCESS_TO_CONSTANT_PTR	The specification assumes that the SYCL implementation addresses space deduction. However, the deduction is performed in the middle end, where it is hard to provide user friendly diagnostics. When you write to raw pointers obtained from `constant_ptr`, there are no available diagnostics. You can enable diagnostics by enabling the `RESTRICT_WRITE_ACCESS_TO_CONSTANT_PTR` macro, which allows `constant_ptr` to use constant pointers as underlying pointer types. After enabling the macro, conversions from `constant_ptr` to raw pointers return constant pointers, and writing to const pointers is diagnosed by the front-end. This behavior does not follow the SYCL specification, since `constant_ptr` conversions to the underlying pointer type will return pointers without any additional qualifiers. The macro is disabled by default.
`__SIGNED_CHARS__` (Windows, Linux)	Defined as '1'.
`_SIZE_T_DEFINED` (Windows)	Defined, no value.
`__SIZE_TYPE__` (Linux)	On Linux, defined as unsigned on IA-32 architecture (C++ only); defined as unsigned long on Intel® 64 architecture.
`__SSE__` (Windows, Linux)	On Linux, defined as '1' for processors that support SSE instructions. On Windows, defined as '1'.
`__SSE2__` (Windows, Linux)	On Linux, defined as '1' for processors that support Intel® SSE2 instructions.
`__SSE3__`	On Linux, defined as '1' for processors that support Intel® SSE3 instructions.

Macro	Description
(Windows, Linux)	
__SSE4_1__ (Windows, Linux)	On Linux, defined as '1' for processors that support Intel® SSE4 instructions.
__SSE4_2__ (Windows, Linux)	On Linux, defined as '1' for processors that support SSSE4 instructions.
__SSSE3__ (Windows, Linux)	On Linux, defined as '1' for processors that support SSSE3 instructions.
__SYCL_COMPILER_VERSION (Windows, Linux)	The build date of the SYCL library, presented in the format YYYYMMDD. **NOTE** This is only available after the SYCL library headers are included in the source code.
SYCL_DISABLE_CPP_VERSION_CHECK_WARNING (Windows, Linux)	Disables the warning displayed if the user tries to compile using an unsupported version of C++.
SYCL_LANGUAGE_VERSION	The SYCL_LANGUAGE_VERSION is defined only when compiling SYCL code.
SYCL_USE_NATIVE_FP_ATOMICS	Enable functions to generate built-in floating-point atomics on the target device by enabling the SYCL_USE_NATIVE_FP_ATOMICS macro. If the target device does not support floating-point atomics, emulated atomics are used instead. The macro is disabled by default.
SYCL2020_CONFORMANT_APIS	Enable compliance with the SYCL 2020 specification for non-compliant functions. Some current SYCL function implementations do not conform to the SYCL 2020 specification and may result in an API break with a direct change. The following non-compliant functions are deprecated: • sycl::get_native<backend::opencl,event>() with return type cl_event. If a nonconforming function is used, the user is notified of the deprecated function and the compiler will suggest that the SYCL2020_CONFORMANT_APIS macro be enabled to enforce compliance. The user should re-write the non-compliant code to be compliant and re-compile with the SYCL2020_CONFORMANT_APIS macro enabled.
unix __unix __unix__ (Linux)	Defined as '1'.
__USER_LABEL_PREFIX__	

Macro	Description
(Linux)	
_VA_LIST_DEFINED (Windows)	Defined, no value.
__VERSION__ (Linux)	The compiler version string
__w64 (Windows)	Defined, no value.
__WCHAR_T (Linux)	Defined as '1'.
_WCHAR_T_DEFINED (Windows)	Defined when option /Zc:wchar_t is specified or "*wctype_t*" is defined in the header file.
__WCHAR_TYPE__ (Linux)	On Linux, defined as long int on IA-32 architecture (C++ only); defined as int on Intel® 64 architecture.
_WCTYPE_T_DEFINED (Windows)	Defined when "*wctype_t*" is defined in the header file.
_WIN32 (Windows)	Defined as '1' while building code targeting IA-32 (C++ only) or Intel® 64 architecture.
_WIN64 (Windows)	Defined as '1' while building code targeting Intel® 64 architecture.
__WINT_TYPE__ (Linux)	Defined as unsigned int.
__x86_64 __x86_64__ (Linux)	Defined as '1' while building code targeting Intel® 64 architecture.

See Also

march compiler option
D compiler option
U compiler option
qopenmp, Qopenmp compiler option
ISO Standard Predefined Macros

Use Predefined Macros to Specify Intel® Compilers

This topic shows how to use predefined macros to specify an Intel® compiler or version of an Intel compiler.

Predefined Macros to Specify Compiler and Version

When you install both the Intel® oneAPI Base Toolkit (Base Kit) and the Intel® oneAPI HPC Toolkit (HPC Kit), you will notice that there are three compilers installed:

- Intel® DPC++ Compiler
- Intel® C++ Compiler
- Intel® C++ Compiler Classic

You can use the following predefined macros to invoke a specific compiler or version of a compiler:

Compiler	**Predefined Macros to Differentiate from Other Compiler**	**Notes**
Intel® DPC++ Compiler	• `SYCL_LANGUAGE_VERSION` • `__INTEL_LLVM_COMPILER` • `__VERSION`	`SYCL_LANGUAGE_VERSION` is defined in SYCL specification and should be defined by all SYCL compilers. `__INTEL_LLVM_COMPILER` is used to select the compiler.
Intel® C++ Compiler	• `__INTEL_LLVM_COMPILER` • `__VERSION`	`__INTEL_LLVM_COMPILER` is used to select the compiler. `__VERSION` is used to select the compiler version.

Predefined Macros for Intel® DPC++ Compiler

The following example uses `#if defined(SYCL_LANGUAGE_VERSION) && defined (__INTEL_LLVM_COMPILER)` to define a code block specific to the Intel® DPC++ Compiler:

```
// dpcpp only
#if defined(SYCL_LANGUAGE_VERSION) && defined (__INTEL_LLVM_COMPILER)
   // code specific for DPC++ compiler below
   // ... ...

   // example only
   std::cout << "SYCL_LANGUAGE_VERSION: " << SYCL_LANGUAGE_VERSION << std::endl;
   std::cout << "__INTEL_LLVM_COMPILER: " << __INTEL_LLVM_COMPILER << std::endl;
   std::cout << "__VERSION__: " << __VERSION__ << std::endl;
#endif
```

Example output using the Intel® oneAPI Toolkit Gold release with an Intel DPC++ Compiler patch release of 2021.1.2:

Linux

```
SYCL_LANGUAGE_VERSION: 202001

__INTEL_LLVM_COMPILER: 202110

__VERSION__: Intel(R) Clang Based C++, gcc 4.2.1 mode
```

Windows

```
SYCL_LANGUAGE_VERSION: 202001

__INTEL_LLVM_COMPILER: 202110

__VERSION__: Intel(R) Clang Based C++, clang 12.0.0
```

Predefined Macros for Intel® C++ Compiler

The following example uses `#if !defined(SYCL_LANGUAGE_VERSION) && defined (__INTEL_LLVM_COMPILER)` to define a code block specific to the Intel® C++ Compiler:

```
// icx only
#if !defined(SYCL_LANGUAGE_VERSION) && defined (__INTEL_LLVM_COMPILER)
   // code specific for Intel C++ Compiler below
   // ... ...

   // example only
   std::cout << "__INTEL_LLVM_COMPILER: " << __INTEL_LLVM_COMPILER << std::endl;
   std::cout << "__VERSION__: " << __VERSION__ << std::endl;
#endif
```

Example output using the Intel® oneAPI Toolkit Gold release with an Intel C++ Compiler patch release of 2021.1.2:

Linux

```
__INTEL_LLVM_COMPILER: 202110

__VERSION__: Intel(R) Clang Based C++, gcc 4.2.1 mode
```

Windows

```
__INTEL_LLVM_COMPILER: 202110

__VERSION__: Intel(R) Clang Based C++, clang 12.0.0
```

Pragmas

Pragmas are directives that provide instructions to the compiler for use in specific cases. For example, you can use the `novector` pragma to specify that a loop should never be vectorized. The keyword `#pragma` is standard in the C++ language, but individual pragmas are machine-specific or operating system-specific, and vary by compiler.

Some pragmas provide the same functionality as compiler options. Pragmas override behavior specified by compiler options.

Some pragmas are available for both Intel® and non-Intel microprocessors but they may perform additional optimizations for Intel® microprocessors than they perform for non-Intel microprocessors. Refer to the individual pragma name for detailed description.

The Intel® oneAPI DPC++/C++ Compiler pragmas are categorized as follows:

- Intel-specific Pragmas - pragmas developed or modified by Intel to work specifically with the Intel oneAPI DPC++/C++ Compiler
- Intel Supported Pragmas - pragmas developed by external sources that are supported by the Intel oneAPI DPC++/C++ Compiler for compatibility reasons

Using Pragmas

You enter pragmas into your C++ source code using the following syntax:

```
#pragma <pragma name>
```

Individual Pragma Descriptions

Each pragma description has the following details:

Section	Description
Short Description	Contains a brief description of what the pragma does.
Syntax	Contains the pragma syntax.
Arguments	Contains a list of the arguments (parameters).
Description	Contains a detailed description of what the pragma does.
Example	Contains typical usage example/s.
See Also	Contains links or paths to other pragmas or related topics.

Intel-Specific Pragma Reference

Pragmas specific to the Intel® oneAPI DPC++/C++ Compiler are listed in the following table.

Most Intel-specific pragmas support host code only unless otherwise noted.

Some pragmas are available for both Intel® microprocessors and non-Intel microprocessors, but may perform additional optimizations for Intel® microprocessors than for non-Intel microprocessors.

Pragma	Description
block_loop/ noblock_loop	Enables or disables loop blocking for the immediately following nested loops. block_loop enables loop blocking for the nested loops. noblock_loop disables loop blocking for the nested loops.
distribute_point	Instructs the compiler to prefer loop distribution at the location indicated.
inline/noinline/ forceinline	Specifies inlining of all calls in a statement. This also describes pragmas forceinline and noinline.
ivdep	Instructs the compiler to ignore assumed vector dependencies.
loop_count	Specifies the iterations for a for loop.
nofusion	Prevents a loop from fusing with adjacent loops.
novector	Specifies that a particular loop should never be vectorized.
omp target variant dispatch	Conditionally calls a procedure offload variant if the specified device is available; otherwise, executes the procedure on the host.
prefetch/noprefetch	Invites the compiler to issue or disable requests to prefetch data from memory. This pragma applies only to Intel® Advanced Vector Extensions 512 (Intel® AVX-512).
unroll/nounroll	Tells the compiler to unroll or not to unroll a counted loop.
unroll_and_jam/ nounroll_and_jam	Enables or disables loop unrolling and jamming. These pragmas can only be applied to iterative for loops.
vector	Tells the compiler that the loop should be vectorized according to the argument keywords.

block_loop/noblock_loop

Enables or disables loop blocking for the immediately following nested loops. block_loop enables loop blocking for the nested loops. noblock_loop disables loop blocking for the nested loops.

Syntax

`#pragma block_loop [`*`clause`*`[,`*`clause`*`]...]`

`#pragma noblock_loop`

Arguments

clause	Can be any of the following:	
	`factor` (*expr*)	*expr* is a positive scalar constant integer expression representing the blocking factor for the specified loops. This clause is optional. If the `factor` clause is not present, the blocking factor will be determined based on processor type and memory access patterns and will be applied to the specified levels in the nested loop following the pragma. At most only one *factor* clause can appear in a `block_loop` pragma.
	`level` (*level_expr*[, *level_expr*]...)	*level_expr* is specified in the form *const1* or *const1:const2* where *const1* is a positive integer constant *m*<= 8 representing the loop at level *m*, where the immediate following loop is level 1. The *const2* is a positive integer constant *n*<= 8 representing the loop at level *n*, where *n* > *m*. *const1:const2* represents the nested loops from level *const1* through *const2*.
	The clauses can be specified in any order. If you do not specify any clause, the compiler chooses the best blocking factor to apply to all levels of the immediately following nested loop.	

Description

The `block_loop` pragma lets you exert greater control over optimizations on a specific loop inside a nested loop.

Using a technique called loop blocking, the `block_loop` pragma separates large iteration counted loops into smaller iteration groups. Execution of these smaller groups can increase the efficiency of cache space use and augment performance.

If there is no `level` and `factor` clause, the blocking factor will be determined based on the processor's type and memory access patterns and it will apply to all the levels in the nested loops following this pragma.

You can use the `noblock_loop` pragma to tune the performance by disabling loop blocking for nested loops.

The loop-carried dependence is ignored during the processing of `block_loop` pragmas.

The `block_loop` pragma is supported in host code only.

```
#pragma  block_loop factor(256) level(1)    /* applies blocking factor 256 to           */
#pragma  block_loop factor(512) level(2)    /* the top level loop in the following
                                               nested loop and blocking factor 512 to
                                               the 2nd level (1st nested) loop          */
```

```
#pragma  block_loop factor(256) level(2)
#pragma  block_loop factor(512) level(1)     /* levels can be specified in any order        */

#pragma  block_loop factor(256) level(1:2)   /* adjacent loops can be specified as a range  */

#pragma  block_loop factor(256)              /* the blocking factor applies to all levels
                                                of loop nest                                */

#pragma  block_loop               /* the blocking factor will be determined based on
                                     processor type and memory access patterns and will
                                     be applied to all the levels in the nested loop
                                     following the directive                                */

#pragma  noblock_loop             /* None of the levels in the nested loop following this
                                     directive will have a blocking factor applied          */
```

Consider the following:

```
#pragma block_loop factor(256) level(1:2)
for (j = 1 ; j<n ; j++){
  f  = 0 ;
  for (i =1 ;i<n  i++){
    f  =  f +   a[i]  *  b  [i] ;
  }
  c [j]  = c[j]  + f ;
}
```

The above code produces the following result after loop blocking:

```
for ( jj=1 ; jj<n/256+1 ; jj+){
  for ( ii = 1 ; ii<n/256+1 ;ii++){
    for ( j = (jj-1)*256+1 ;  min(jj*256, n) ;j++){
      f = 0 ;
      for ( i = (ii-1)*256+1 ;i<min(ii*256,n) ;i++){
        f = f + a[i] * b [i];
      }
      c[j]  = c[j] + f ;
    }
  }
}
```

distribute_point

Instructs the compiler to prefer loop distribution at the location indicated.

Syntax

`#pragma distribute_point`

Arguments

None

Description

The `distribute_point` pragma is used to suggest to the compiler to split large loops into smaller ones; this is particularly useful in cases where optimizations like vectorization cannot take place due to excessive register usage.

The following rules apply to this pragma:

- When the pragma is placed inside a loop, the compiler distributes the loop at that point. All loop-carried dependencies are ignored.
- When inside the loop, pragmas cannot be placed within an `if` statement.
- When the pragma is placed outside the loop, the compiler distributes the loop based on an internal heuristic. The compiler determines where to distribute the loops and observes data dependency. If the pragmas are placed inside the loop, the compiler supports multiple instances of the pragma.

The `distribute_point` pragma is supported in host code only.

Examples

Use the `distribute_point` pragma outside the loop:

```
#define NUM 1024
void loop_distribution_pragma1(
       double a[NUM], double b[NUM], double c[NUM],
       double x[NUM], double y[NUM], double z[NUM] ) {
  int i;

  // Before distribution or splitting the loop
  #pragma distribute_point
  for (i=0; i< NUM; i++) {
    a[i] = a[i] + i;
    b[i] = b[i] + i;
    c[i] = c[i] + i;
    x[i] = x[i] + i;
    y[i] = y[i] + i;
    z[i] = z[i] + i;
  }
}
```

Use the `distribute_point` pragma inside the loop:

```
#define NUM 1024
void loop_distribution_pragma2(
       double a[NUM], double b[NUM], double c[NUM],
       double x[NUM], double y[NUM], double z[NUM] ) {
  int i;

  // After distribution or splitting the loop.
  for (i=0; i< NUM; i++) {
    a[i] = a[i] +i;
    b[i] = b[i] +i;
    c[i] = c[i] +i;
    #pragma distribute_point
    x[i] = x[i] +i;
    y[i] = y[i] +i;
    z[i] = z[i] +i;
  }
}
```

Use the `distribute_point` pragma inside and outside the loop:

```
void dist1(int a[], int b[], int c[], int d[]) {
  #pragma distribute_point
    // Compiler will automatically decide where to
    // distribute. Data dependency is observed.
  for (int i=1; i<1000; i++) {
    b[i] = a[i] + 1;
    c[i] = a[i] + b[i];
```

```
    d[i] = c[i] + 1;
  }
}

void dist2(int a[], int b[], int c[], int d[]) {
  for (int i=1; i<1000; i++) {
    b[i] = a[i] + 1;

    #pragma distribute_point
      // Distribution will start here,
      // ignoring all loop-carried dependency.
      c[i] = a[i] + b[i];
      d[i] = c[i] + 1;
  }
}
```

inline, noinline, forceinline

Specifies inlining of all calls in a statement. This also describes pragmas forceinline and noinline.

Syntax

`#pragma inline [recursive]`

`#pragma forceinline [recursive]`

`#pragma noinline`

Arguments

`recursive`	Indicates that the pragma applies to all of the calls that are called by these calls, recursively, down the call chain.

Description

`inline`, `forceinline`, and `noinline` are statement-specific inlining pragmas. Each can be placed before a C/C++ statement, and it will then apply to all of the calls within a statement and all calls within statements nested within that statement.

The `forceinline` pragma indicates that the calls in question should be inlined whenever the compiler is capable of doing so.

The `inline` pragma is a hint to the compiler that the user prefers that the calls in question be inlined, but expects the compiler not to inline them if its heuristics determine that the inlining would be overly aggressive and might slow down the compilation of the source code excessively, create too large of an executable, or degrade performance.

The `noinline` pragma indicates that the calls in question should not be inlined.

These statement-specific pragmas take precedence over the corresponding function-specific pragmas.

The `inline`, `forceinline`, and `noinline` pragmas are supported in host code only.

Examples

Use the `forceinline` recursive pragma:

```
#include <stdio.h>

static void fun(float a[100][100], float b[100][100]) {
  inti , j;
  for (i = 0; i < 100; i++) {
    for (j = 0; j < 100; j++) {
      a[i][j] = 2 * i;
      b[i][j] = 4 * j;
    }
  }
}

static void sun(float a[100][100], float b[100][100]) {
  int i, j;

  for (i = 0; i < 100; i++) {
    for (j = 0; j < 100; j++) {
      a[i][j] = 2 * i;
      b[i][j] = 4 * j;
    }
    fun(a, b);
  }
}
static float a[100][100];
static float b[100][100];

extern int main() {
  int i, j;

  for (i = 0; i < 100; i++) {
    for (j = 0; j < 100; j++) {
      a[i][j] = i + j;
      b[i][j] = i - j;
    }
  }
  for (i = 0; i < 99; i++) {
    fun(a, b);
#pragma forceinline recursive
    for (j = 0; j < 99; j++) {
      sun(a, b);
    }
  }
  fprintf(stderr, "%d %d\n", a[99][9], b[99][99]);
}
```

The `forceinline recursive` pragma applies to the call 'sun(a,b)' as well as the call 'fun(a,b)' called inside 'sun(a,b)'.

ivdep

Instructs the compiler to ignore assumed vector dependencies.

Syntax

`#pragma ivdep`

Arguments

None

Description

The `ivdep` pragma instructs the compiler to ignore assumed vector dependencies. To ensure correct code, the compiler treats an assumed dependence as a proven dependence, which prevents vectorization. This pragma overrides that decision. Use this pragma only when you know that the assumed loop dependencies are safe to ignore.

The `ivdep` pragma is supported in host code only.

In addition to the `ivdep` pragma, the `vector` pragma can be used to override the efficiency heuristics of the vectorizer.

> **NOTE**
> The proven dependencies that prevent vectorization are not ignored, only assumed dependencies are ignored.

Examples

The loop in this example will not vectorize without the `ivdep` pragma, since the value of *k* is not known; vectorization would be illegal if `k < 0`:

```
void ignore_vec_dep(int *a, int k, int c, int m) {
  #pragma ivdep
  for (int i = 0; i < m; i++)
    a[i] = a[i + k] * c;
}
```

The pragma binds only the `for` loop contained in current function. This includes a `for` loop contained in a sub-function called by the current function:

```
#pragma ivdep
  for (i=1; i<n; i++) {
    e[ix[2][i]] = e[ix[2][i]]+1.0;
    e[ix[3][i]] = e[ix[3][i]]+2.0;
}
```

This loop requires the parallel option in addition to the `ivdep` pragma to indicate there is no loop-carried dependencies:

```
#pragma ivdep
  for (j=0; j<n; j++) { a[b[j]] = a[b[j]] + 1; }
```

This loop requires the parallel option in addition to the `ivdep` pragma to ensure there is no loop-carried dependency for the store into `a()`.

See Also

Function Annotations and the SIMD Directive for Vectorization
novector pragma
vector pragma

loop_count

Specifies the iterations for a for loop.

Syntax

`#pragma loop_count(n)`

`#pragma loop_count=n`

or

`#pragma loop_count(n1[, n2]...)`

`#pragma loop_count=n1[, n2]...`

or

`#pragma loop_count min(n),max(n),avg(n)`

`#pragma loop_count min=n, max=n, avg=n`

Arguments

(n) or =n	A non-negative integer value. The compiler will attempt to iterate the next loop the number of times specified in *n*; however, the number of iterations is not guaranteed.
(n1[,n2]...) or = n1[,n2]...	Non-negative integer values. The compiler will attempt to iterate the next loop the number of time specified by *n1* or *n2*, or some other unspecified number of times. This behavior allows the compiler some flexibility in attempting to unroll the loop. The number of iterations is not guaranteed.
min(n), max(n), avg(n) or min=n, max=n, avg=n	Non-negative integer values. Specify one or more in any order without duplication. The compiler insures the next loop iterates for the specified maximum, minimum, or average number (*n1*) of times. The specified number of iterations is guaranteed for min and max.

Description

The `loop_count` pragma specifies the minimum, maximum, or average number of iterations for a `for` loop. In addition, a list of commonly occurring values can be specified to help the compiler generate multiple versions and perform complete unrolling.

You can specify more than one pragma for a single loop; however, do not duplicate the pragma.

The `loop_count` pragma is supported in host code only.

Examples

Use the `loop_count` pragma to iterate through the loop a minimum of three, a maximum of ten, and average of five times:

```
#include <stdio.h>
int i;
int mysum(int start, int end, int a) {
  int iret=0;
  #pragma loop_count min(3), max(10), avg(5)
    for (i=start;i<=end;i++)
    iret += a;
    return iret;
```

```
}

int main() {
  int t;
  t = mysum(1, 10, 3);
  printf("t1=%d\r\n",t);
  t = mysum(2, 6, 2);
  printf("t2=%d\r\n",t);
  t = mysum(5, 12, 1);
  printf("t3=%d\r\n",t);
}
```

nofusion

Prevents a loop from fusing with adjacent loops.

Syntax

`#pragma nofusion`

Arguments

None

Description

The `nofusion` pragma lets you fine tune your program on a loop-by-loop basis. This pragma should be placed immediately before the loop that should not be fused.

The `nofusion` pragma is supported in host code only.

Examples

```
#define SIZE 1024

int sub () {
int B[SIZE], A[SIZE];
  int i, j, k=0;
  for(j=0; j<SIZE; j++)
    A[j] = A[j] + B[j];

#pragma nofusion
  for (i=0; i<SIZE; i++)
    k += A[i] + 1;
  return k;
}
```

novector

Specifies that a particular loop should never be vectorized.

Syntax

`#pragma novector`

Arguments

None

Description

The `novector` pragma specifies that a particular loop should never be vectorized, even if it is legal to do so. When avoiding vectorization of a loop is desirable (when vectorization results in a performance regression rather than improvement), the `novector` pragma can be used in the source text to disable vectorization of a loop. This behavior is in contrast to the `vector always` pragma.

The `novector` pragma is supported in host code only.

Examples

Use the `novector` pragma:

```
void foo(int lb, int ub) {
  #pragma novector
  for(j=lb; j<ub; j++) { a[j]=a[j]+b[j]; }
}
```

When the trip count (`ub - lb`) is too low to make vectorization worthwhile, you can use the `novector` pragma to tell the compiler not to vectorize, even if the loop is considered vectorizable.

See Also

Function Annotations and the SIMD Directive for Vectorization
vector pragma

omp target variant dispatch

Conditionally calls a procedure offload variant if the specified device is available; otherwise, executes the procedure on the host.

Syntax

```
#pragma omp target variant dispatch {device(integer-expression) | nowait |
subdevice([integer-constant ,] integer-expression [ : integer-expression [ : integer-
expression] ] ) | use_device_pointer (ptr-list)}
```

Arguments

`device`	Tells the compiler to call the variant only if device *n* is available.
`subdevice`	Tells the compiler to call the variant only if the specified tiles or compute slices are available.
`nowait`	Tells the compiler that calls to the procedure can occur asynchronously. If `nowait` is not specified, calls occur synchronously.
`use_device_ptr`	Tells the compiler to use the corresponding device pointer instead of the host pointer when the variant procedure is called.

If both `device` and `subdevice` are specified, the variant is called only if the specified tiles or compute slices are available on device *n*. Otherwise, the base version of the procedure is called on the host.

Description

The `omp target variant dispatch` pragma causes the compiler to emit conditional dispatch code around the associated procedure call that follows the pragma. If the specified device is available, the variant version is called.

The name of the procedure associated with the `omp target variant dispatch` pragma must have appeared in an `omp declare variant` pragma in the specification part of the calling scope. The interface of the variant procedure must be accessible in the base procedure where `omp target variant dispatch` appears.

The `omp target variant dispatch` pragma is supported in host code only.

> **NOTE**
> Use pragma `omp target variant dispatch` when calling Intel® oneAPI Math Kernel Library (oneMKL).
>
> In other cases, we recommend you use the OpenMP* pragma `omp dispatch`. For more information about pragma `omp dispatch`, see the OpenMP* documentation.

prefetch/noprefetch

Invites the compiler to issue or disable requests to prefetch data from memory. This pragma applies only to Intel® Advanced Vector Extensions 512 (Intel® AVX-512).

Syntax

```
#pragma prefetch

#pragma prefetch *:hint[:distance]

#pragma prefetch [var1 [: hint1 [: distance1]] [, var2 [: hint2 [: distance2]]]...]

#pragma noprefetch [var1 [, var2]...]
```

Arguments

var

An optional memory reference (data to be prefetched)

hint

An optional hint to the compiler to specify the type of prefetch. Possible values:

- `1`: For integer data that will be reused
- 2: For integer and floating point data that will be reused from L2 cache
- `3`: For data that will be reused from L3 cache
- `4`: For data that will not be reused

To use this argument, you must also specify *var*.

distance

An optional integer argument with a value greater than 0. It indicates the number of loop iterations ahead of which a prefetch is issued, before the corresponding load or store instruction. To use this argument, you must also specify *var* and *hint*.

Description

The `prefetch` pragma hints to the compiler to generate data prefetches for some memory references. These hints affect the heuristics used in the compiler. Prefetching data can minimize the effects of memory latency.

If you specify the `prefetch` pragma with no arguments, all arrays accessed in the immediately following loop are prefetched.

If the loop includes the expression `A(j)`, placing `#pragma prefetch A` in front of the loop instructs the compiler to insert prefetches for `A(j + d)` within the loop. Here, *d* is the number of iterations ahead of which to prefetch the data, and is determined by the compiler.

If you specify `#pragma prefetch *`, then *hint* and *distance* prefetches all array accesses in the loop.

To use these pragmas, the compiler general optimization level must be set at option `O2` or higher.

The `noprefetch` pragma hints to the compiler not to generate data prefetches for some memory references. This affects the heuristics used in the compiler.

The `prefetch` and `noprefetch` pragmas are supported in host code only.

Examples

Use the `prefetch` pragma:

```
#pragma prefetch htab_p:1:30
#pragma prefetch htab_p:0:6

// Issue vprefetch1 for htab_p with a distance of 30 vectorized iterations ahead
// Issue vprefetch0 for htab_p with a distance of 6 vectorized iterations ahead
// If pragmas are not present, compiler chooses both distance values

for (j=0; j<2*N; j++) { htab_p[i*m1 + j] = -1; }
```

Use `noprefetch` and `prefetch` pragmas together:

```
#pragma noprefetch b
#pragma prefetch a
for(i=0; i<m; i++) { a[i]=b[i]+1; }
```

Use `noprefetch` and `prefetch` pragmas together:

```
for (i=i0; i!=i1; i+=is) {

float sum = b[i];
int ip = srow[i];
int c = col[ip];

#pragma noprefetch col
#pragma prefetch value:1:80
#pragma prefetch x:1:40

for(; ip<srow[i+1]; c=col[++ip])
  sum -= value[ip] * x[c];
  y[i] = sum;
}
```

unroll/nounroll

Tells the compiler to unroll or not to unroll a counted loop.

Syntax

`#pragma unroll`

`#pragma unroll(`*n*`)`

`#pragma nounroll`

Arguments

n	The unrolling factor representing the number of times to unroll a loop; it must be an integer constant from 0 through 255.

Description

The `unroll`*[n]* pragma tells the compiler how many times to unroll a counted loop.

The `unroll` pragma must precede the `for` statement for each `for` loop it affects. If *n* is specified, the optimizer unrolls the loop *n* times. If *n* is omitted or if it is outside the allowed range, the optimizer assigns the number of times to unroll the loop.

This pragma is supported only when option `O3` is set. The `unroll` pragma overrides any setting of loop unrolling from the command line.

The pragma can be applied for the innermost loop nest as well as for the outer loop nest. If applied to outer loop nests, the current implementation supports complete outer loop unrolling. The loops inside the loop nest are either not unrolled at all or completely unrolled. The compiler generates correct code by comparing *n* and the loop count.

When unrolling a loop increases register pressure and code size it may be necessary to prevent unrolling of a loop. In such cases, use the `nounroll` pragma. The `nounroll` pragma instructs the compiler not to unroll a specified loop.

The `unroll` and `nounroll` pragmas are supported in both host and device code.

Target device support: CPU, GPU, FPGA.

Examples

Use the `unroll` pragma for innermost loop unrolling:

```
void unroll(int a[], int b[], int c[], int d[]) {
  #pragma unroll(4)
  for (int i = 1; i < 100; i++) {
    b[i] = a[i] + 1;
    d[i] = c[i] + 1;
  }
}
```

Use the `unroll` pragma for outer loop unrolling:

```
int m = 0;
int dir[4]= {1,2,3,4};
int data[10];
#pragma unroll (4)  // outer loop unrolling
  for (int i = 0; i < 4; i++) {
     for (int j = dir[i]; data[j]==N ; j+=dir[i])
        m++;
  }
```

When you place the `unroll` pragma before the first `for` loop, it causes the compiler to unroll the outer loop completely. If an `unroll` pragma is placed before the inner `for` loop as well as before the outer `for` loop, the compiler ignores the inner `for` loop `unroll` pragma. If the `unroll` pragma is placed only for the innermost loop, the compiler unrolls the innermost loop according to some factor.

unroll_and_jam/nounroll_and_jam

Enables or disables loop unrolling and jamming. These pragmas can only be applied to iterative for loops.

Syntax

`#pragma unroll_and_jam`

`#pragma unroll_and_jam (`*n*`)`

`#pragma nounroll_and_jam`

Arguments

n	The unrolling factor representing the number of times to unroll a loop; it must be an integer constant from 0 through 255

Description

The `unroll_and_jam` pragma partially unrolls one or more loops higher in the nest than the innermost loop and fuses/jams the resulting loops back together. This transformation allows more reuses in the loop.

This pragma is not effective on innermost loops. Ensure that the immediately following loop is not the innermost loop after compiler-initiated interchanges are completed.

Specifying this pragma is a hint to the compiler that the unroll and jam sequence is legal and profitable. The compiler enables this transformation whenever possible.

The `unroll_and_jam` pragma must precede the `for` statement for each `for` loop it affects. If *n* is specified, the optimizer unrolls the loop *n* times. If *n* is omitted or if it is outside the allowed range, the optimizer assigns the number of times to unroll the loop. The compiler generates correct code by comparing *n* and the loop count.

This pragma is supported only when compiler option `O3` is set. The `unroll_and_jam` pragma overrides any setting of loop unrolling from the command line.

When unrolling a loop increases register pressure and code size it may be necessary to prevent unrolling of a nested loop or an imperfect nested loop. In such cases, use the `nounroll_and_jam` pragma. The `nounroll_and_jam` pragma hints to the compiler not to unroll a specified loop.

The `unroll_and_jam` and `nounroll_and_jam` pragmas are supported in host code only.

Examples

Use the `unroll_and_jam` pragma:

```
int a[10][10];
int b[10][10];
int c[10][10];
int d[10][10];
void unroll(int n) {
    int i,j,k;
    #pragma unroll_and_jam (6)
    for (i = 1; i < n; i++) {
       #pragma unroll_and_jam (6)
```

```
        for (j = 1; j < n; j++) {
           for (k = 1; k < n; k++){
             a[i][j] += b[i][k]*c[k][j];
             }
        }
    }
}
```

vector

Tells the compiler that the loop should be vectorized according to the argument keywords.

Syntax

`#pragma vector {always[assert]|dynamic_align|nodynamic_align|temporal|nontemporal|[no]vecremainder|vectorlength(n1[, n2]...)}`

`#pragma vector nontemporal[(`*`var1`*`[, `*`var2`*`, ...])]`

Arguments

`always`	Instructs the compiler to override any efficiency heuristic during the decision to vectorize or not, and vectorize non-unit strides or very unaligned memory accesses; controls the vectorization of the subsequent loop in the program; optionally takes the keyword assert.
`dynamic_align`	Instructs the compiler to perform dynamic alignment optimization for the loop.
`nodynamic_align`	Disables dynamic alignment optimization for the loop.
`nontemporal`	Instructs the compiler to use non-temporal (that is, streaming) stores on systems based on all supported architectures, unless otherwise specified; optionally takes a comma-separated list of variables. When this pragma is specified, it is your responsibility to also insert any fences as required to ensure correct memory ordering within a thread or across threads. One typical way to do this is to insert a `_mm_sfence` intrinsic call just after the loops (such as the initialization loop) where the compiler may insert streaming store instructions.
`temporal`	Instructs the compiler to use temporal (that is, non-streaming) stores on systems based on all supported architectures, unless otherwise specified.
`vecremainder`	Instructs the compiler to vectorize the remainder loop when the original loop is vectorized.
`novecremainder`	Instructs the compiler not to vectorize the remainder loop when the original loop is vectorized.
`vectorlength` (*n1*[, *n2*]...)	Instructs the vectorizer which vector length/factor to use when generating the main vector loop.

Description

The `vector` pragma indicates that the loop should be vectorized, if it is legal to do so, ignoring normal heuristic decisions about profitability. The `vector` pragma takes several argument keywords to specify the kind of loop vectorization required. The compiler does not apply the `vector` pragma to nested loops, each nested loop needs a preceding pragma statement. Place the pragma before the loop control statement.

The `vector` pragma is supported in host code only.

Using the `always` keyword

When the `always` argument keyword is used, the pragma will ignore compiler efficiency heuristics for the subsequent loop. When `assert` is added, the compiler will generate a diagnostic message if the loop cannot be vectorized for any reason.

Using the `dynamic_align` and `nodynamic_align` keywords

Dynamic alignment is an optimization the compiler can perform to improve alignment of memory references inside the loop. It involves peeling iterations from the vector loop into a scalar loop (which may, in turn, also be vectorized) before the vector loop so that the vector loop aligns with a particular memory reference. Specifying `dynamic_align` enables the optimization to be performed, but the compiler will still use efficiency heuristics to determine whether the optimization will be applied to the loop. Specifying `nodynamic_align` disables the optimization. By default, the compiler does not perform optimization.

Using the `nontemporal` and `temporal` keywords

The `nontemporal` and `temporal` argument keywords are used to control how the "stores" of register contents to storage are performed (streaming versus non-streaming) on systems based on IA-32 and Intel® 64 architectures.

By default, the compiler automatically determines whether a streaming store should be used for each variable.

Streaming stores may cause significant performance improvements over non-streaming stores for large numbers on certain processors. However, the misuse of streaming stores can significantly degrade performance.

Using the `[no]vecremainder` keyword

If keyword `vecremainder` is specified, the compiler tries to vectorize the remainder loop when the main loop is vectorized. Even if the `always` keyword is specified, the remainder loop vectorization is still a subject of compiler efficiency heuristics.

If keyword `novecremainder` is specified, the compiler vectorizes the main loop, but it does not vectorize the remainder loop.

Using the `vectorlength` keyword

n is an integer power of 2; the value must be 2, 4, 6, 8, 16, 32, or 64. If more than one value is specified, the vectorizer will choose one of the specified vector lengths based on a cost model decision.

> **NOTE**
> The pragma `vector` should be used with care.
>
> Overriding the efficiency heuristics of the compiler should only be done if the programmer is absolutely sure that vectorization will improve performance.

See Also

Function Annotations and the SIMD Directive for Vectorization

Intel-supported Pragma Reference

The Intel® oneAPI DPC++/C++ Compiler supports the following pragmas to ensure compatibility with other compilers.

Pragmas Compatible with the Microsoft* Compiler

The following pragmas are compatible with the Microsoft Compiler. For more information about these pragmas, go to the Microsoft Developer Network (http://msdn.microsoft.com).

Pragma	Description
`alloc_text`	Names the code section where the specified function definitions are to reside.
`auto_inline`	Excludes any function defined within the range where **`off`** is specified from being considered as candidates for automatic inline expansion.
`bss_seg`	Indicates to the compiler the segment where uninitialized variables are stored in the `.obj` file.
`check_stack`	The **`on`** argument indicates that stack checking should be enabled for functions that follow and the **`off`** argument indicates that stack checking should be disabled for functions that follow.
`code_seg`	Specifies a code section where functions are to be allocated.
`comment`	Places a comment record into an object file or executable file.
`component`	Controls collecting of browse information or dependency information from within source files.
`conform`	Specifies the run-time behavior of the `/Zc:`*`forScope`* compiler option.
`const_seg`	Specifies the segment where functions are stored in the `.obj` file.
`data_seg`	Specifies the default section for initialized data.
`deprecated`	Indicates that a function, type, or any other identifier may not be supported in a future release or indicates that a function, type, or any other identifier should not be used any more.
`fenv_access`	Informs an implementation that a program may test status flags or run under a non-default control mode.
`float_control`	Specifies floating-point behavior for a function.
`fp_contract`	Allows or disallows the implementation to contract expressions.
`loop`	Controls how the loop code will be considered or excluded from consideration by the auto-vectorizer.
`init_seg`	Specifies the section to contain C++ initialization code for the translation unit.
`message`	Displays the specified string literal to the standard output device (`stdout`).

Pragma	Description
`optimize`	Specifies optimizations to be performed on functions below the pragma or until the next optimize pragma; implemented to partly support the Microsoft implementation of same pragma; for the Intel oneAPI DPC++/C++ Compiler implementation, see the `optimize` reference page.
`pointers_to_members`	Specifies whether a pointer to a class member can be declared before its associated class definition and is used to control the pointer size and the code required to interpret the pointer.
`pop_macro`	Sets the value of the specified macro to the value on the top of the stack.
`push_macro`	Saves the value of the specified macro on the top of the stack.
`region/endregion`	Specifies a code segment in the Microsoft Visual Studio* Code Editor that expands and contracts by using the outlining feature.
`section`	Creates a section in an `.obj` file. Once a section is defined, it remains valid for the remainder of the compilation.
`vtordisp`	The **on** argument enables the generation of hidden `vtordisp` members and the **off** disables them. **push** argument pushes the current `vtordisp` setting to the internal compiler stack. **pop** argument removes the top record from the compiler stack and restores the removed value of `vtordisp`.
`warning`	Allows selective modification of the behavior of compiler warning messages.
`weak`	Declares symbol you enter to be weak.

OpenMP* Standard Pragmas

The Intel oneAPI DPC++/C++ Compiler currently supports OpenMP* 5.0 Version TR4, and some OpenMP Version 5.1 pragmas. Supported pragmas are isted below. For more information about these pragmas, reference the OpenMP* Version 5.1 specification.

Intel-specific clauses are noted in the affected pragma description.

Pragma	Description
omp allocate	Specifies memory allocators to use for object allocation and deallocation
omp atomic	Specifies a computation that must be executed atomically.
omp barrier	Specifies a point in the code where each thread must wait until all threads in the team arrive.
omp cancel	Requests cancellation of the innermost enclosing region of the type specified, and causes the encountering task to proceed to the end of the cancelled construct.
omp cancellation point	Defines a point at which implicit or explicit tasks check to see if cancellation has been requested for the innermost enclosing region of the type specified. This construct does not implement a synchronization between threads or tasks.

Pragma	Description
omp critical	Specifies a code block that is restricted to access by only one thread at a time.
omp declare reduction	Declares User-Defined Reduction (UDR) functions (reduction identifiers) that can be used as reduction operators in a reduction clause.
omp declare simd	Creates a version of a function that can process multiple arguments using Single Instruction Multiple Data (SIMD) instructions from a single invocation from a SIMD loop.
omp declare target	Specifies functions and variables that are created or mapped to a device.
omp declare variant	Identifies a variant of a base procedure and specifies the context in which this variant is used.
omp dispatch	Determines if a procedure variant is called for a given procedure.
omp distribute	Specifies that the iterations of one or more loops should be distributed among the initial threads of all thread teams in a league.
omp distribute parallel for	Specifies a loop that can be executed in parallel by multiple threads that are members of multiple teams.
omp distribute parallel for simd	Specifies a loop that will be executed in parallel by multiple threads that are members of multiple teams. It will be executed concurrently using SIMD instructions.
omp distribute simd	Specifies a loop that will be distributed across the primary threads of the teams region. It will be executed concurrently using SIMD instructions.
omp flush	Identifies a point at which a thread's temporary view of memory becomes consistent with the memory.
omp for	Specifies a work-sharing loop. Iterations of the loop are executed in parallel by the threads in the team.
omp for simd	Specifies that the iterations of the loop will be distributed across threads in the team. Iterations executed by each thread can also be executed concurrently using SIMD instructions.
omp interop	Identifies a foreign runtime context and identifies runtime characteristics of that context, enabling interoperability with it.
omp loop	Specifies that the iterations of the associated loops can execute in any order or concurrently.
omp masked	Specifies a structured block that is executed by a subset of the threads of the current team.
omp master (deprecated; see omp masked)	Specifies a code block that must be executed only once by the primary thread of the team.
omp ordered	Specifies a block of code that the threads in a team must execute in the natural order of the loop iterations, or as a stand-alone directive, specifies cross-iteration dependences in a doacross loop-nest.
omp parallel	Specifies that a structured block should be run in parallel by a team of threads.

Pragma	Description
omp parallel for	Provides an abbreviated way to specify a parallel region containing only a FOR construct.
omp parallel for simd	Specifies a parallel construct that contains one for simd construct and no other statement.
omp parallel sections	Specifies a parallel construct that contains only a sections construct.
omp requires	Lists the features that an implementation must support so that the program compiles and runs correctly.
omp scan	Specifies a scan computation that updates each list item in each iteration of the loop.
omp scope	Defines a structured block that is executed by all threads in a team but where additional OpenMP* operations can be specified.
omp sections	Defines a set of structured blocks that will be distributed among the threads in the team.
omp simd	Transforms the loop into a loop that will be executed concurrently using SIMD instructions.
omp single	Specifies that a block of code is to be executed by only one thread in the team.
omp target	Creates a device data environment and executes the construct on that device.
omp target data	Specifies that variables are mapped to a device data environment for the extent of the region.
omp target enter data	Specifies that variables are mapped to a device data environment.
omp target exit data	Specifies that variables are unmapped from a device data environment.
omp target parallel loop	Provides an abbreviated way to specify a target region that contains only a parallel loop construct.
omp target teams	Creates a device data environment and executes the construct on the same device. It also creates a league of thread teams with the primary thread in each team executing the structured block.
omp target teams distribute	Creates a device data environment and then executes the construct on that device. It also specifies that loop iterations will be distributed among the primary threads of all thread teams in a league created by a teams construct.
omp target teams distribute parallel for	Creates a device data environment and then executes the construct on that device. It also specifies a loop that can be executed in parallel by multiple threads that are members of multiple teams created by a teams construct.
omp target teams distribute parallel for simd	Creates a device data environment and then executes the construct on that device. It also specifies a loop that can be executed in parallel by multiple threads that are members of multiple teams created by a teams construct. The loop will be distributed across the teams, which will be executed concurrently using SIMD instructions.

Pragma	Description
omp target teams distribute simd	Creates a device data environment and then executes the construct on that device. It also specifies that loop iterations will be distributed among the primary threads of all thread teams in a league created by a teams construct. It will be executed concurrently using SIMD instructions.
omp target teams loop	Provides an abbreviated way to specify a target region that contains only a teams loop construct.
omp target update	Makes the items listed in the device data environment consistent between the device and host, in accordance with the motion clauses on the pragma.
omp task	Specifies a code block whose execution may be deferred.
omp taskgroup	Causes the program to wait until the completion of all enclosed and descendant tasks.
omp taskwait	Specifies a wait on the completion of child tasks generated since the beginning of the current task.
omp taskyield	Specifies that the current task can be suspended at this point in favor of execution of a different task.
omp teams	Creates a league of thread teams inside a target region to execute a structured block in the initial thread of each team.
omp teams distribute	Creates a league of thread teams and specifies that loop iterations will be distributed among the primary threads of all thread teams in the league.
omp teams distribute parallel for	Creates a league of thread teams and specifies that the associated loop can be executed in parallel by multiple threads that are members of multiple teams.
omp teams distribute parallel for simd	Creates a league of thread teams and specifies that the associated loop can be executed concurrently using SIMD instructions in parallel by multiple threads that are members of multiple teams.
omp teams distribute simd	Creates a league of thread teams and specifies that the associated loop will be distributed across the primary threads of the teams and executed concurrently using SIMD instructions.
omp teams loop	Provides an abbreviated way to specify a teams construct that contains only a loop construct.
omp threadprivate	Specifies a list of globally-visible variables that will be allocated private to each thread.

Pragmas Compatible with Other Compilers

The following pragmas are compatible with other compilers. For more information about these pragmas, see the documentation for the specified compiler.

Pragma	Description
`include_directory`	HP-compatible pragma. It appends the string argument to the list of places to search for `#include` files.

Pragma	Description
`poison`	GCC-compatible pragma. It labels the identifiers you want removed from your program; an error results when compiling a `"poisoned" identifier; #pragma POISON` is also supported.
`options`	GCC-compatible pragma; It sets the alignment of fields in structures.
`weak`	GCC-compatible pragma, it declares the symbol you enter to be weak.

See Also

Intel-specific Pragmas
Zc compiler option

Error Handling

This topic describes compiler warnings and errors. The compiler sends these messages, along with the erroneous source line, to stderr.

Warnings

Warning messages report legal but questionable use of C or C++. The compiler displays warnings by default. You can suppress warning messages by specifying an appropriate compiler option. Warnings do not stop translation or linking. Warnings do not interfere with any output files.

The following is a representative warning message:

```
unknown pragma ignored [-Wunknown-pragmas]
```

Some warnings that start with `-W` can be disabled using the negative form of the option `-Wno-`. For example, option `-Wno-unknown-pragmas` disables option `-Wunknown-pragmas`.

Errors

Error messages report syntactic or semantic misuse of C or C++. The compiler always displays error messages. Errors suppress object code for the module containing the error and prevent linking, but they allow parsing to continue to detect other possible errors.

The following are some representative error messages:

```
expected ';' at end of declaration
```

```
unexpected type name 'b': expected expression
```

For a summary of warning and error options, see the Clang documentation.

Part IV

Compilation

This section contains information about features that can affect compilation, such as environment variables, and using configuration files.

Compilation Overview

Compilation Environment

You can customize the environment used during compilation using a combination of

- Configuration Files
- Environment variables
- Response Files

You can also modify the compilation by adding additional include directories for the compiler to search during compilation. See Specify Compiler Files for more information.

Default Compiler Behavior

The Intel® oneAPI DPC++/C++ Compiler processes C/C++ and SYCL source files. Compilation can be divided into these major phases: :

- Preprocessing
- Semantic parsing
- Optimization
- Code generation
- Linking

By default, the compiler performs the first four phases of compilation and then invokes the linker to perform the linking phase. The default linkers are `ld` for Linux and `link` for Windows.

Default settings for the compiler include:

- Optimization level O2 (`-O2`)
- Floating point model = fast (`-fp-model=fast`)
- `icpx` C++ language standard: C++14
- `dpcpp` C++ language standard: C++17
- C++ runtime:
  - Linux: libstdc++, using headers and libraries installed on the system
  - Windows: Microsoft Visual C++ (MSVC) provided headers and libraries
- SVML and specific interfaces enabled to call into the Intel libirc library

Customize the Compilation Process

The Intel® oneAPI DPC++/C++ Compiler provides multiple options to customize compilation.

Preprocessing

Several options are available to customize preprocessing. For example, you can:

- Specify the location of system and user header files
- Specify macros
- Stop the compilation process after preprocessing
- Send preprocessed output to stdout

You can optionally use your own preprocessor to generate a preprocessed file which can then be passed to the compiler.

For a detailed list of preprocessing options, see Preprocessor Options.

Compiling

Compiler options are not required to compile your program, but can be used to control different aspects of your application, such as:

- Code generation
- Optimization
- Output file (type, name, location)
- Linking properties
- Size of the executable
- Speed of the executable

For a detailed list of all compiler options, see Compiler Options.

Linking

You can perform the linking phase using the Intel compiler to invoke the linker (default) or by calling the linker directly.

> **NOTE** On Linux, calling the linker directly requires explicit understanding of which specific system and Intel libraries need to be linked in, as they will need to be passed directly to the linker.

To prevent default linking at compilation time, use the `-c` option. You must then explicitly pass along the generated object on the compilation command line and the compiler will create the final binary.

You can pass options to the linker for additional control of the linking phase. See Pass Options to the Linker for additional information.

See Also

Compiler Options
Specify Compiler Files
Preprocessor Options
Pass Options to the Linker

Supported Environment Variables

You can customize your system environment by specifying paths where the compiler searches for certain files such as libraries, include files, configuration files, and certain settings.

Compiler Compile-Time Environment Variables

The following table shows the compile-time environment variables that affect the compiler:

Compile-Time Environment Variable	Description
`CL` (Windows) `_CL_` (Windows)	Define the files and options you use most often with the CL variable. Note: You cannot set the CL environment variable to a string that contains an equal sign. You can use the pound sign instead. In the following example, the pound sign (#) is used as a substitute for an equal sign in the assigned string: `SET CL=/Dtest#100`
`IA32ROOT` (IA-32 architecture and Intel® 64 architecture)	Points to the directories containing the include and library files for a non-standard installation structure.
`ICXCFG`	Specifies the configuration file for customizing compilations when invoking the compiler using `icx`. Used instead of the default configuration file.
`ICPXCFG`	Specifies the configuration file for customizing compilations when invoking the compiler using `icpx`. Used instead of the default configuration file.
`__INTEL_PRE_CFLAGS` `__INTEL_POST_CFLAGS`	Specifies a set of compiler options to add to the compile line. This is an extension to the facility already provided in the compiler configuration file `icx.cfg`. **NOTE** By default, a configuration file named `icx.cfg` (Windows, Linux), or `icpx.cfg` (Linux) is used. This file is in the same directory as the compiler executable. To use another configuration file in another location, you can use the `ICXCFG` (Windows, Linux), `ICPXCFG` (Linux) environment variable to assign the directory and file name for the configuration file. You can insert command line options in the prefix position using `__INTEL_PRE_CFLAGS` , or in the suffix position using `__INTEL_POST_CFLAGS`. The command line is built as follows: **Syntax:**`icx` *<PRE flags> <flags from configuration file> <flags from the compiler invocation> <POST flags>* **NOTE** The driver issues a warning that the compiler is overriding an option because of an environment variable, but only when you include the option `/W5` (Windows) or `-w3` (Linux).
`PATH`	Specifies the directories the system searches for binary executable files. **NOTE** On Windows, this also affects the search for Dynamic Link Libraries (DLLs).

Compile-Time Environment Variable	Description
TMP TMPDIR TEMP	Specifies the location for temporary files. If none of these are specified, or writeable, or found, the compiler stores temporary files in /tmp (Linux) or the current directory (Windows). The compiler searches for these variables in the following order: TMP, TMPDIR, and TEMP. **NOTE** On Windows, these environment variables cannot be set from Visual Studio.
LD_LIBRARY_PATH (Linux)	Specifies the location for shared objects (.so files).
INCLUDE (Windows)	Specifies the directories for the source header files (include files).
LIB (Windows)	Specifies the directories for all libraries used by the compiler and linker.
GNU Environment Variables and Extensions	
CPATH (Linux)	Specifies the path to include directory for C/C++ compilations.
C_INCLUDE_PATH (Linux)	Specifies path to include directory for C compilations.
CPLUS_INCLUDE_PATH (Linux)	Specifies path to include directory for C++ compilations.
DEPENDENCIES_OUTPUT (Linux)	Specifies how to output dependencies for make based on the non-system header files processed by the compiler. System header files are ignored in the dependency output.
GCC_EXEC_PREFIX (Linux)	Specifies alternative names for the linker (ld) and assembler (as).
LIBRARY_PATH (Linux)	Specifies the path for libraries to be used during the link phase.
SUNPRO_DEPENDENCIES (Linux)	This variable is the same as DEPENDENCIES_OUTPUT, except that system header files are not ignored.

NOTE INTEL_ROOT is an environment variable that is reserved for the Intel® Compiler. Its use is not supported.

Compiler Run-Time Environment Variables

The following table summarizes compiler environment variables that are recognized at run time.

Run-Time Environment Variable	Description
GNU extensions (recognized by the Intel OpenMP* compatibility library)	

Run-Time Environment Variable	Description
`GOMP_CPU_AFFINITY` (Linux)	GNU extension recognized by the Intel OpenMP compatibility library. Specifies a list of OS processor IDs. You must set this environment variable before the first parallel region or before certain API calls including `omp_get_max_threads()`, `omp_get_num_procs()` and any affinity API calls. For detailed information on this environment variable, see *Thread Affinity Interface*. **Default:** Affinity is disabled
`GOMP_STACKSIZE` (Linux)	GNU extension recognized by the Intel OpenMP compatibility library. Same as `OMP_STACKSIZE.KMP_STACKSIZE` overrides `GOMP_STACKSIZE`, which overrides `OMP_STACKSIZE`. **Default:** See the description for `OMP_STACKSIZE`.
OpenMP Environment Variables (OMP_) and Extensions (KMP_)	
`OMP_CANCELLATION`	Activates cancellation of the innermost enclosing region of the type specified. If set to `TRUE`, the effects of the cancel construct and of cancellation points are enabled and cancellation is activated. If set to `FALSE`, cancellation is disabled and the cancel construct and cancellation points are effectively ignored. **NOTE** Internal barrier code will work differently depending on whether the cancellation is enabled. Barrier code should repeatedly check the global flag to figure out if the cancellation had been triggered. If a thread observes the cancellation it should leave the barrier prematurely with the return value 1 (may wake up other threads). Otherwise, it should leave the barrier with the return value 0. Enables (`TRUE`) or disables (`FALSE`) cancellation of the innermost enclosing region of the type specified. **Default:** FALSE **Example:** `OMP_CANCELLATION=TRUE`
`OMP_DISPLAY_ENV`	Enables (`TRUE`) or disables (`FALSE`) the printing to stderr of the OpenMP version number and the values associated with the OpenMP environment variable. Possible values are: TRUE, FALSE, or VERBOSE.

Run-Time Environment Variable	Description
	Default: FALSE **Example:** `OMP_DISPLAY_ENV=TRUE`
`OMP_DEFAULT_DEVICE`	Sets the device that will be used in a target region. The OpenMP routine `omp_set_default_device` or a `device` clause in a target pragma can override this variable. If no device with the specified device number exists, the code is executed on the host. If this environment variable is not set, device number 0 is used.
`OMP_DYNAMIC`	Enables (`TRUE`) or disables (`FALSE`) the dynamic adjustment of the number of threads. **Default:** FALSE **Example:** `OMP_DYNAMIC=TRUE`
`OMP_MAX_ACTIVE_LEVELS`	The maximum number of levels of parallel nesting for the program. Possible values: Non-negative integer. **Default:** 1
`OMP_NESTED`	Deprecated; use `OMP_MAX_ACTIVE_LEVELS` instead.
`OMP_NUM_THREADS`	Sets the maximum number of threads to use for OpenMP parallel regions if no other value is specified in the application. The value can be a single integer, in which case it specifies the number of threads for all parallel regions. The value can also be a comma-separated list of integers, in which case each integer specifies the number of threads for a parallel region at a nesting level. The first position in the list represents the outer-most parallel nesting level, the second position represents the next-inner parallel nesting level, and so on. At any level, the integer can be left out of the list. If the first integer in a list is left out, it implies the normal default value for threads is used at the outer-most level. If the integer is left out of any other level, the number of threads for that level is inherited from the previous level. This environment variable applies to the options `Qopenmp` (Windows) or `qopenmp` (Linux). **Default:** The number of processors visible to the operating system on which the program is executed. **Syntax:** `OMP_NUM_THREADS=value[,value]*`

Run-Time Environment Variable	Description
`OMP_PLACES`	Specifies an explicit ordered list of places, either as an abstract name describing a set of places or as an explicit list of places described by nonnegative numbers. An exclusion operator "!" can also be used to exclude the number or place immediately following the operator. For **explicit lists**, the meaning of the numbers and how the numbering is done for a list of nonnegative numbers are implementation defined. Generally, the numbers represent the smallest unit of execution exposed by the execution environment, typically a hardware thread. Intervals can be specified using the `<lower-bound>` : `<length>` : `<stride>` notation to represent the following list of numbers: `"<lower-bound>, <lower-bound> + <stride>, ..., <lower-bound> +(<length>-1)*<stride>."` When `<stride>` is omitted, a unit stride is assumed. Intervals can specify numbers within a place as well as sequences of places. `# EXPLICIT LIST EXAMPLE` `setenv OMP_PLACES "{0,1,2,3},{4,5,6,7},{8,9,10,11},{12,13,14,15}"` `setenv OMP_PLACES "{0:4},{4:4},{8:4},{12:4}"` `setenv OMP_PLACES "{0:4}:4:4"` The **abstract names** listed below should be understood by the execution and run-time environment: • `threads`: Each place corresponds to a single hardware thread on the target machine. • `cores`: Each place corresponds to a single core (having one or more hardware threads) on the target machine. • `ll_caches`: Each place corresponds to a set of cores that share the last level cache on the device. • `numa_domains`: Each place corresponds to a set of cores for which their closest memory on the device is 1) the same memory and 2) at a similar distance from the cores. • `sockets`: Each place corresponds to a single socket (consisting of one or more cores) on the target machine. When requesting fewer places or more resources than available on the system, the determination of which resources of type `abstract_name` are to be included in the place list is implementation-defined.

Run-Time Environment Variable	Description
	The precise definitions of the abstract names are implementation defined. An implementation may also add abstract names as appropriate for the target platform. The abstract name may be appended by a positive number in parentheses to denote the length of the place list to be created, that is `abstract_name(`*`num-places`*`)`. `# ABSTRACT NAMES EXAMPLE` `  setenv OMP_PLACES threads` `  setenv OMP_PLACES threads(4)` **NOTE** If any numerical values cannot be mapped to a processor on the target platform the behavior is implementation-defined. The behavior is also implementation-defined when the `OMP_PLACES` environment variable is defined using an abstract name.
`OMP_PROC_BIND` (Windows, Linux)	Sets the thread affinity policy to be used for parallel regions at the corresponding nested level. Enables (`TRUE`) or disables (`FALSE`) the binding of threads to processor contexts. If enabled, this is the same as specifying `KMP_AFFINITY=scatter`. If disabled, this is the same as specifying `KMP_AFFINITY=none`. **Acceptable values:** `TRUE`, `FALSE`, or a comma separated list, each element of which is one of the following values: `PRIMARY`, `MASTER` (deprecated), `CLOSE`, `SPREAD`. **Default:** `FALSE` If set to `FALSE`, the execution environment may move OpenMP threads between OpenMP places, thread affinity is disabled, and proc_bind clauses on parallel constructs are ignored. Otherwise, the execution environment should not move OpenMP threads between OpenMP places, thread affinity is enabled, and the initial thread is bound to the first place in the OpenMP place list. If set to `PRIMARY`, all threads are bound to the same place as the primary thread. If set to `CLOSE`, threads are bound to successive places, close to where the primary thread is bound. If set to `SPREAD`, the primary thread's partition is subdivided and threads are bound to single place successive sub-partitions.

Run-Time Environment Variable	Description
	NOTE `KMP_AFFINITY` takes precedence over `GOMP_CPU_AFFINITY` and `OMP_PROC_BIND`. `GOMP_CPU_AFFINITY` takes precedence over `OMP_PROC_BIND`.
`OMP_SCHEDULE`	Sets the run-time schedule type and an optional chunk size. **Default:** `static`, no chunk size specified **Example syntax:** `OMP_SCHEDULE="[modifier:]kind[,chunk_size]"` where • `modifier` is one of `monotonic` or `nonmonotonic` • `kind` is one of `static`, `dynamic`, `guided`, or `auto` • `chunk_size` is a positive integer **NOTE** Some environment variables are available for both Intel® microprocessors and non-Intel microprocessors, but may perform additional optimizations for Intel® microprocessors than for non-Intel microprocessors.
`OMP_STACKSIZE`	Sets the number of bytes to allocate for each OpenMP thread to use as the private stack for the thread. Recommended size is 16M. Use the optional suffixes to specify byte units: `B` (bytes), `K` (Kilobytes), `M` (Megabytes), `G` (Gigabytes), or `T` (Terabytes) to specify the units. If you specify a value without a suffix, the byte unit is assumed to be `K` (Kilobytes). This variable does not affect the native operating system threads created by the user program, or the thread executing the sequential part of an OpenMP program. The `kmp_{set,get}_stacksize_s()` routines set/retrieve the value. The `kmp_set_stacksize_s()` routine must be called from sequential part, before first parallel region is created. Otherwise, calling `kmp_set_stacksize_s()` has no effect. **Default (IA-32 architecture):** 2M **Default (Intel® 64 architecture):** 4M

Run-Time Environment Variable	Description
	Related environment variables: KMP_STACKSIZE (overrides OMP_STACKSIZE). **Syntax:** OMP_STACKSIZE=value
OMP_THREAD_LIMIT	Limits the number of simultaneously-executing threads in an OpenMP program. If this limit is reached and another native operating system thread encounters OpenMP API calls or constructs, the program can abort with an error message. If this limit is reached when an OpenMP parallel region begins, a one-time warning message might be generated indicating that the number of threads in the team was reduced, but the program will continue. This environment variable is only used for programs compiled with the following option: Qopenmp (Windows) or qopenmp (Linux). The omp_get_thread_limit() routine returns the value of the limit. **Default:** No enforced limit **Related environment variable:** KMP_ALL_THREADS (overrides OMP_THREAD_LIMIT). **Example syntax:** OMP_THREAD_LIMIT=value
OMP_WAIT_POLICY	Decides whether threads spin (active) or yield (passive) while they are waiting. OMP_WAIT_POLICY=ACTIVE is an alias for KMP_LIBRARY=turnaround, and OMP_WAIT_POLICY=PASSIVE is an alias for KMP_LIBRARY=throughput. **Default:** Passive **Syntax:** OMP_WAIT_POLICY=value
OMP_DISPLAY_AFFINITY	Instructs the runtime to display formatted affinity information for all OpenMP threads in the parallel region upon entering the first parallel region and when any change occurs in the information accessible by the format specifiers listed in the OMP_AFFINITY_FORMAT entry. Possible values: TRUE or FALSE **Default:** FALSE
OMP_AFFINITY_FORMAT	Defines the format when displaying OpenMP thread affinity information. Possible values are any string with the following format field available:

Run-Time Environment Variable	Description
	• %t or %{team_num}: Value returned by omp_get_team_num() • %T or %{num_teams}: Value returned by omp_get_num_teams() • %L or %{nesting_level}: Value returned by omp_get_level() • %n or %{thread_num}: Value returned by omp_get_thread_num() • %a or %{ancestor_tnum}: Value returned by omp_get_ancestor_thread_num(omp_get_level() - 1) • %H or %{host}: Name of host device • %P or %{process_id}: Process ID • %i or %{native_thread_id}: Native thread ID on the platform • %A or %{thread_affinity}: List of processor ID on which a thread may execute **Default:** 'OMP: pid %P tid %i thread %n bound to OS proc set {%A}'
OMP_MAX_TASK_PRIORITY	Controls the use of task priorities by setting the initial value. Possible values: Non-negative integer. **Default:** 0
OMP_TOOL	Controls whether the OpenMP runtime will try to register a first party tool that uses OMPT interface. Possible values: ENABLED or DISABLED. **Default:** ENABLED **NOTE** Only the host OpenMP runtime is supported.
OMP_TOOL_LIBRARIES	Sets a list of first-party tool locations that use the OMPT interface. The list enumerates names of dynamically-loadable libraries with OS-specific path separator. **Default:** Empty **NOTE** Only the host OpenMP runtime is supported.
OMP_TOOL_VERBOSE_INIT	Controls whether the OpenMP runtime will verbosely log the registration of a tool that uses the OMPT interface. Possible values: • DISABLED: Do not log the registration.

Run-Time Environment Variable	Description
	• STDOUT: Log the registration to stdout. • STDERR: Log the registration to stderr. • File_Name: Log the registration to the location specified by File_Name. **Default:**DISABLED **NOTE** Only the host OpenMP runtime is supported.
OMP_DEBUG	Controls whether the OpenMP runtime collects information that an OMPD library may need to support a tool. Possible values: ENABLED or DISABLED. **Default:** DISABLED **NOTE** Only the host OpenMP runtime is supported.
OMP_ALLOCATOR	Specifies the default allocator for allocation calls, directives, and clauses that do not specify an allocator. **Default:** omp_default_mem_alloc **Syntax:** <PredefinedMemAllocator> | <PredefinedMemSpace> | <PredefinedMemSpace>:<Traits> Currently supported values for <PredefinedMemAllocator> and <PredefinedMemSpace> : • omp_default_mem_alloc and omp_default_mem_space Additional values are supported if libmemkind is available and there is system support for it: • omp_high_bw_mem_alloc and omp_high_bw_mem_space • omp_large_cap_mem_alloc and omp_large_cap_mem_space Refer to the OpenMP specification for more information.
OMP_NUM_TEAMS	Sets the maximum number of teams created by a teams construct by setting nteams-var ICV. Possible values: Positive integer. **Default:** 1
OMP_TEAMS_THREAD_LIMIT	Sets the maximum number of OpenMP threads to use in each team created by a teams construct.

Run-Time Environment Variable	Description
	Possible values: Positive integer. **Default:** <NumberOfProcessors> / <nteams-var ICV>
`KMP_AFFINITY` (Linux, Windows)	Enables run-time library to bind threads to physical processing units. You must set this environment variable before the first parallel region, or certain API calls including `omp_get_max_threads()`, `omp_get_num_procs()` and any affinity API calls. For detailed information on this environment variable, see *Thread Affinity Interface*. **Default:** noverbose,warnings,noreset,respect,granularity=core,none Default (Windows with multiple processor groups): noverbose,warnings,noreset,norespect,granularity=group,compact,0,0 **NOTE** On Windows with multiple processor groups, the norespect affinity modifier is assumed when the process affinity mask equals a single processor group (which is default on Windows). Otherwise, the respect affinity modifier is used.
`KMP_ALL_THREADS`	Limits the number of simultaneously-executing threads in an OpenMP program. If this limit is reached and another native operating system thread encounters OpenMP API calls or constructs, then the program may abort with an error message. If this limit is reached at the time an OpenMP parallel region begins, a one-time warning message may be generated indicating that the number of threads in the team was reduced, but the program will continue execution. This environment variable is only used for programs compiled with the `Qopenmp`(Windows) or `qopenmp` (Linux) option. **Default:** No enforced limit.
`KMP_BLOCKTIME`	Sets the time, in milliseconds, that a thread should wait, after completing the execution of a parallel region, before sleeping. Use the optional character suffixes: `s` (seconds), `m` (minutes), `h` (hours), or `d` (days) to specify the units. Specify `infinite` for an unlimited wait time.

<table>
<tr><th>Run-Time Environment Variable</th><th>Description</th></tr>
<tr><td></td><td>Default: 200 milliseconds

Related Environment Variable: <code>KMP_LIBRARY</code> environment variable.</td></tr>
<tr><td><code>KMP_CPUINFO_FILE</code></td><td>Specifies an alternate file name for a file containing the machine topology description. The file must be in the same format as <code>/proc/cpuinfo</code>.

Default: None</td></tr>
<tr><td><code>KMP_DETERMINISTIC_REDUCTION</code></td><td>Enables (<code>TRUE</code>) or disables (<code>FALSE</code>) the use of a specific ordering of the reduction operations for implementing the reduction clause for an OpenMP parallel region. This has the effect that, for a given number of threads, in a given parallel region, for a given data set and reduction operation, a floating point reduction done for an OpenMP reduction clause has a consistent floating point result from run to run, since round-off errors are identical.

NOTE When compiling, you must set the following flag to ensure correct behavior:
<code>-fp-model precise</code> (Linux)
<code>-fp:precise</code> (Windows)
Default: <code>FALSE</code></td></tr>
<tr><td><code>KMP_DYNAMIC_MODE</code></td><td>Selects the method used to determine the number of threads to use for a parallel region when <code>OMP_DYNAMIC=TRUE</code>. Possible values: (<code>asat</code> | <code>load_balance</code> | <code>thread_limit</code>), where, <code>asat</code>: estimates number of threads based on parallel start time; NOTE
Support for <code>asat</code> (automatic self-allocating threads) is now deprecated and will be removed in a future release.
<code>load_balance</code>: tries to avoid using more threads than available execution units on the machine;
<code>thread_limit</code>: tries to avoid using more threads than total execution units on the machine.
Default (IA-32 architecture): <code>load_balance</code> (on all supported OSes)</td></tr>
</table>

Run-Time Environment Variable	Description
	Default (Intel® 64 architecture): `load_balance` (on all supported OSes)
`KMP_HOT_TEAMS_MAX_LEVEL`	Sets the maximum nested level to which teams of threads will be hot. **NOTE** A *hot* team is a team of threads optimized for faster reuse by subsequent parallel regions. In a hot team, threads are kept ready for execution of the next parallel region, in contrast to the cold team, which is freed after each parallel region, with its threads going into a common pool of threads. For values of 2 and above, nested parallelism should be enabled. **Default:** 1
`KMP_HOT_TEAMS_MODE`	Specifies the run-time behavior when the number of threads in a hot team is reduced. Possible values: • 0: Extra threads are freed and put into a common pool of threads. • 1: Extra threads are kept in the team in reserve, for faster reuse in subsequent parallel regions. **Default:** 0
`KMP_HW_SUBSET`	Specifies the subset of available hardware resources for the hardware topology hierarchy. The subset is specified in terms of number of units per upper layer unit starting from top layer downwards. For example, it can specify the number of sockets (top layer units), cores per socket, and the threads per core, to use with an OpenMP application. It is a convenient alternative to writing complicated explicit affinity settings or a limiting process affinity mask. You can also specify an offset value to set which resources to use. When available, you can specify attributes to select different subsets of resources. An extended syntax is available when `KMP_TOPOLOGY_METHOD=hwloc`. Depending on what resources are detected, you may be able to specify additional resources, such as NUMA nodes and groups of hardware resources that share certain cache levels. **Basic syntax:**

Run-Time Environment Variable	Description
	`[num_units]ID[@offset][:attribute] [, [num_units]ID[@offset][:attribute]...]` where • `num_units` is either a positive integer, which requests an exact number of resources, or an asterisk (*), which means using all available resources at that layer (for example, using all cores per socket). If `num_units` is not specified, the asterisk (*) semantics are assumed. • `ID` is a supported ID: *S - socket*: *num_units* specifies the requested number of sockets. *D - die*: *num_units* specifies the requested number of dies per socket. *C - core*: *num_units* specifies the requested number of cores per die - if any - otherwise, per socket. *T - thread*: *num_units* specifies the requested number of HW threads per core. Supported unit IDs are not case-sensitive. • `offset` is the number of units to skip (optional). • `attribute` is an attribute differentiating resources at a particular layer (optional). This is only available for the core layer on machines with Intel® Hybrid Technology. The attributes available to users are: • Core type: Either *intel_atom* or *intel_core* • Core efficiency: Specified as `effnum` where *num* is a number from 0 to the number of core efficiencies detected in the machine topology minus one. For example: `eff0`. The greater the efficiency number, the more performant the core. There may be more core efficiencies than core types, which can be viewed by setting `KMP_AFFINITY=verbose`. **NOTE** The hardware cache can be specified as a unit, for example L2 for L2 cache, or LL for last level cache. **Extended syntax when `KMP_TOPOLOGY_METHOD=hwloc`:**

Run-Time Environment Variable	Description
	Additional IDs can be specified if detected. For example: `N - numa` — *num_units* specifies the requested number of NUMA nodes per upper layer unit, e.g. per socket. `TI - tile` — *num_units* specifies the requested number of tiles to use per upper layer unit, e.g. per NUMA node. When any *numa* or *tile* units are specified in `KMP_HW_SUBSET`, the `KMP_TOPOLOGY_METHOD` will be automatically set to `hwloc`,so there is no need to set it explicitly. If you don't specify one or more types of resource, such as socket or thread, all available resources of that type are used. The run-time library prints a warning, and the setting of `KMP_HW_SUBSET` is ignored if: • a resource is specified, but detection of that resource is not supported by the chosen topology detection method and/or • a resource is specified twice. An exception to this condition is if attributes differentiate the resource. • attributes are used when unavailable, not detected in the machine topology, or conflict with each other. This variable does not work if the OpenMP affinity is set to `disabled`. **Default:** If omitted, the default value is to use all the available hardware resources. **Examples:** • `2s,4c,2t`: Use the first 2 sockets (s0 and s1), the first 4 cores on each socket (c0 - c3), and 2 threads per core. • `2s@2,4c@8,2t`: Skip the first 2 sockets (s0 and s1) and use 2 sockets (s2-s3), skip the first 8 cores (c0-c7) and use 4 cores on each socket (c8-c11), and use 2 threads per core. • `5C@1,3T`: Use all available sockets, skip the first core and use 5 cores, and use 3 threads per core. • `1T`: Use all cores on all sockets, 1 thread per core. • `1s, 1d, 1n, 1c, 1t`: Use 1 socket, 1 die, 1 NUMA node, 1 core, 1 thread - use HW thread as a result.

Run-Time Environment Variable	Description
	• 4c:intel_atom,5c:intel_core: Use all available sockets and use 4 Intel Atom® processor cores and 5 Intel® Core™ processor cores per socket. • 2c:eff0,3c:eff1: Use all available sockets and use 2 cores with efficiency 0 and 3 cores with efficiency 1 per socket. • 1s, 1c, 1t: Use 1 socket, 1 core, 1 thread. This may result in using single thread on a 3-layer topology architecture, or multiple threads on 4-layer or 5-layer architecture. Result may even be different on the same architecture, depending on KMP_TOPOLOGY_METHOD specified, as hwloc can often detect more topology layers than default method used by the OpenMP run-time library. To see the result of the setting, you can specify the verbose modifier in for the KMP_AFFINITY environment variable. The OpenMP run-time library will output to stderr stream the information about discovered HW topology before and after the KMP_HW_SUBSET setting was applied. For example, on Intel® Xeon Phi™ 7210 CPU in SNC-4 Clustering Mode, the setting KMP_AFFINITY=verbose KMP_HW_SUBSET=1N,1L2,1L1,1T outputs various verbose information to stderr, including the following lines about discovered HW topology before and after KMP_HW_SUBSET was applied: • Info #191: KMP_AFFINITY: 1 socket x 4 NUMA domains/socket x 8 tiles/NUMA domain x 2 cores/tile x 4 threads/core. (64 total cores) • Info #191: KMP_HW_SUBSET 1 socket x 1 NUMA domain/socket x 1 tile/NUMA domain x 1 core/tile x 1 thread/core (1 total cores)
KMP_INHERIT_FP_CONTROL	Enables (TRUE) or disables (FALSE) the copying of the floating-point control settings of the primary thread to the floating-point control settings of the OpenMP worker threads at the start of each parallel region. **Default:** TRUE
KMP_LIBRARY	Selects the OpenMP run-time library execution mode. The values for this variable are serial, turnaround, or throughput. **Default:** throughput
KMP_PLACE_THREADS	Deprecated; use KMP_HW_SUBSET instead.

Run-Time Environment Variable	Description
KMP_SETTINGS	Enables (TRUE) or disables (FALSE) the printing of OpenMP run-time library environment variables during program execution. Two lists of variables are printed: user-defined environment variables settings and effective values of variables used by OpenMP run-time library. **Default:** FALSE
KMP_STACKSIZE	Sets the number of bytes to allocate for each OpenMP thread to use as its private stack. Recommended size is 16m. Use the optional suffixes to specify byte units: B (bytes), K (Kilobytes), M (Megabytes), G (Gigabytes), or T (Terabytes) to specify the units. If you specify a value without a suffix, the byte unit is assumed to be K (Kilobytes). KMP_STACKSIZE overrides GOMP_STACKSIZE, which overrides OMP_STACKSIZE.**Default (IA-32 architecture):** 2m **Default (Intel® 64 architecture):** 4m
KMP_TOPOLOGY_METHOD	Forces OpenMP to use a particular machine topology modeling method. Possible values are: • all: Lets OpenMP choose which topology method is most appropriate based on the platform and possibly other environment variable settings. • cpuid_leaf11: Decodes the APIC identifiers as specified by leaf 11 of the *cpuid* instruction. • cpuid_leaf4: Decodes the APIC identifiers as specified in leaf 4 of the *cpuid* instruction. • cpuinfo: If KMP_CPUINFO_FILE is not specified, forces OpenMP to parse /proc/cpuinfo to determine the topology (Linux only). If KMP_CPUINFO_FILE is specified as described above, uses it (Windows or Linux). • group: Models the machine as a 2-level map, with level 0 specifying the different processors in a group, and level 1 specifying the different groups (Windows 64-bit only) . **NOTE** Support for group is now deprecated and will be removed in a future release. Use all instead.

Run-Time Environment Variable	Description
	• flat: Models the machine as a flat (linear) list of processors. • hwloc: Models the machine as the Portable Hardware Locality* (hwloc) library does. This model is the most detailed and includes, but is not limited to: numa nodes, packages, cores, hardware threads, caches, and Windows processor groups. **Default:** all
KMP_USER_LEVEL_MWAIT	Enables (TRUE) or disables (FALSE) the use of user-level mwait as alternative to putting waiting threads to sleep, if available, either from ring3 or WAITPKG. **Default:** FALSE
KMP_VERSION	Enables (TRUE) or disables (FALSE) the printing of OpenMP run-time library version information during program execution. **Default:** FALSE
KMP_WARNINGS	Enables (TRUE) or disables (FALSE) displaying warnings from the OpenMP run-time library during program execution. **Default:** TRUE
OpenMP Offload Environment Variables (OMP_, LIBOMPTARGET)	
OMP_TARGET_OFFLOAD	Controls the program behavior when offloading a target region. Possible values: • MANDATORY: Program execution is terminated if a device construct or device memory routine is encountered and the device is not available or is not supported. • DISABLED: Disables target offloading to devices and execution occurs on the host. • DEFAULT: Target offloading is enabled if the device is available and supported. **Default:** DEFAULT
LIBOMPTARGET_DEBUG	Controls whether debugging information will be displayed from the offload runtime. Possible values: • 0: Disabled. • 1: Displays basic debug information from the plugin actions such as device detection, kernel compilation, memory copy operations, kernel invocations, and other plugin-dependent actions.

Run-Time Environment Variable	Description
	• 2: Displays which GPU runtime API functions are invoked with which arguments and parameters in addition to the information displayed with value 1. **Default:** 0
LIBOMPTARGET_INFO	Controls whether basic offloading information will be displayed from the offload runtime. Possible values: • 0: Disabled. • 1: Prints all data arguments upon entering an OpenMP device kernel. • 2: Indicates when a mapped address already exists in the device mapping table. • 4: Dump the contents of the device pointer map if target offloading fails. • 8: Indicates when an entry is changed in the device mapping table. • 32: Indicates when data is copied to and from the device. **Default:** 0
LIBOMPTARGET_PLUGIN	Specifies which offload plugin is used when offloading a target region. Possible values: • LEVEL_ZERO | LEVEL0 | level_zero | level0: Uses Intel® oneAPI Level Zero (Level Zero) offload plugin. • OPENCL | opencl: Uses OpenCL offload plugin. • X86_64 | x86_64: Uses X86_64 plugin. **Default:** LEVEL_ZERO
LIBOMPTARGET_DEVICETYPE	Selects device type to which a target region is offloaded. Possible values: • GPU | gpu: GPU device is used. • CPU | cpu: CPU device is used. Offload plugin support for device type: • Level Zero offload plugin only supports GPU type. • OpenCL offload plugin supports both GPU and CPU types. • X86_64 offload plugin ignores this variable. **Default:** GPU
LIBOMPTARGET_PLUGIN_PROFILE	Enables basic plugin profiling and displays the result when program finishes.

Run-Time Environment Variable	Description
	Default: Disabled **Syntax:** <Value>[,usec], where <Value>=1 | T | t The unit of reported time is microsecond if ",usec" is appended, millisecond otherwise.
LIBOMPTARGET_DYNAMIC_MEMORY_SIZE	Sets the size of preallocated memory in MB to service malloc calls on the device. Currently, calls to free memory on the device do not release memory. All the allocated memory is released only when the program exits. Possible values: Non-negative integer. **Default:** 0
OpenMP Offload Environment Variables for Level Zero Offload Plugin	
LIBOMPTARGET_LEVEL0_COMPILATION_OPTIONS	Passes extra build options when building native target program binaries. Possible values: Valid Level Zero build options.
LIBOMPTARGET_DEVICES	Controls how subdevices or sub-subdevices are exposed to users if device supports subdevices. Possible values: • DEVICE | device: Only top-level devices are reported as OpenMP devices and subdevice clause is supported. • SUBDEVICE | subdevice: Only first-level subdevices are reported as OpenMP devices and subdevice clause is ignored. • SUBSUBDEVICE | subsubdevice: Only second-level subdevices are reported as OpenMP devices and subdevice clause is ignored. • ALL | all: All devices and subdevices are reported as OpenMP devices and subdevice clause is ignored. **Default:** DEVICE
LIBOMPTARGET_LEVEL0_MEMORY_POOL	Controls memory pool configuration. Possible values: • 0 : Disables using memory pool. • <PoolInfoList>=<PoolInfo>[,<PoolInfoList>] <PoolInfo>=<MemType>[,<AllocMax>[,<Capacity>[,<PoolSize>]]] <MemType>=all | device | host | shared <AllocMax> is a positive integer or empty

Run-Time Environment Variable	Description
	`<Capacity>` is a positive integer or empty `<PoolSize>` is a positive integer or empty `<AllocMax>` means maximum allocation size in MB supported by the pool, `<Capacity>` means number of allocations supported by a single memory block allocated from the Level Zero runtime, and `<PoolSize>` means maximum size of the total pool size in MB. **Examples:** • `all,2,8,1024`: Enables memory pool for all memory types which can allocate up to eight 2MB blocks from a single block allocated from Level Zero with 1GB total pool size allowed. • `device,1,4,512`: Enables memory pool only for `device` memory type which can allocate up to four 1MB blocks from a single block allocated from Level Zero with 512MB total pool size allowed. **Default:** `all,1,4,256`
`LIBOMPTARGET_LEVEL0_USE_COPY_ENGINE`	Disables use of copy engine for memory transfers. Possible values: • `0` | `F` | `f`: Disables use of copy engine. **Default:** Enabled
`LIBOMPTARGET_LEVEL0_DEFAULT_TARGET_MEM`	Selects memory type returned by the `omp_target_alloc` routine. Possible values: • `DEVICE` | `device`: Returned memory type is `device` type. Device owns the memory and data movement is explicit. • `SHARED` | `shared`: Returned memory type is `shared` type. Ownership of the memory is shared between host and device, and data movement is implicit. • `HOST` | `host`: Returned memory type is `host` type. Host owns the memory and data movement is implicit. **Default:** `DEVICE`
`LIBOMPTARGET_LEVEL0_STAGING_BUFFER_SIZE`	Sets the staging buffer size in KB that is used in data transfer between host and device. Possible values: Non-negative integers where 0 disables use of staging buffer. **Default:** `4`
OpenMP Offload Environment Variables for OpenCL Offload Plugin	

Run-Time Environment Variable	Description
LIBOMPTARGET_OPENCL_COMPILATION_OPTIONS	Passes extra compilation options when compiling target programs from SPIRV target images. Possible values: Valid OpenCL compilation options.
LIBOMPTARGET_OPENCL_LINKING_OPTIONS	Passes extra linking options when linking target programs. Possible values: Valid OpenCL linking options.
LIBOMPTARGET_OPENCL_USE_SVM	Enables or disables using OpenCL SVM memory for default target memory type. Possible values: • 1 | T | t: Enables using OpenCL SVM memory. • 0 | F | f: Disables using OpenCL SVM memory. **Default:** Disabled
LIBOMPTARGET_OPENCL_DATA_TRANSFER_METHOD	Selects memory transfer method to use. This is only effective when OpenCL SVM memory is enabled by setting LIBOMPTARGET_OPENCL_USE_SVM. Possible values: • 0: Uses clEnqueueRead and WriteBuffer API on a temporary OpenCL buffer. • 1: Uses clEnqueueSVMMap and Unmap API. • 2: Uses clEnqueueSVMMemcpy API. **Default:** 1
DPC++ Environment Variables	
DPCPP_CPU_CU_AFFINITY	Set thread affinity to CPU. The value and meaning is the following: • close - threads are pinned to CPU cores successively through available cores. • spread - threads are spread to available cores. • master - threads are put in the same cores as master. If DPCPP_CPU_CU_AFFINITY is set, master thread is pinned as well, otherwise master thread is not pinned This environment variable is similar to the OMP_PROC_BIND variable used by OpenMP. **Default:** Not set
DPCPP_CPU_NUM_CUS	Set the numbers threads used for kernel execution. To avoid over subscription, maximum value of DPCPP_CPU_NUM_CUS should be the number of hardware threads. If DPCPP_CPU_NUM_CUS is 1, all the workgroups are executed sequentially by a single thread and this is useful for debugging.

Run-Time Environment Variable	Description
	This environment variable is similar to OMP_NUM_THREADS variable used by OpenMP. **Default:** Not set. Determined by Intel® oneAPI Threading Building Blocks (oneTBB).
DPCPP_CPU_PLACES	Specify the places that affinities are set. The value is { sockets | numa_domains | cores | threads }. This environment variable is similar to the OMP_PLACES variable used by OpenMP. If value is numa_domains, oneTBB NUMA API will be used. This is analogous to OMP_PLACES=numa_domains in the OpenMP 5.1 Specification. oneTBB task arena is bound to numa node and SYCL nd range is uniformly distributed to task arenas. DPCPP_CPU_PLACES is suggested to be used together with DPCPP_CPU_CU_AFFINITY. **Default:** cores
DPCPP_CPU_SCHEDULE	Specify the algorithm for scheduling work-groups by the scheduler. Currently, DPC++ uses oneTBB for scheduling when using the OpenCL CPU driver. The value selects the petitioner used by the oneTBB scheduler. The value and meaning is the following: • dynamic - oneTBB auto_partitioner. It performs sufficient splitting to balance load. • affinity - oneTBB affinity_partitioner. It improves auto_partitioner's cache affinity by its choice of mapping subranges to worker threads compared to • static - oneTBB static_partitioner. It distributes range iterations among worker threads as uniformly as possible. oneTBB partitioner relies grain-size to control chunking. Grain-size is 1 by default, indicating every work-group can be executed independently. **Default:** dynamic

The following table summarizes CPU environment variables that are recognized at run time.

Runtime configuration	Default value	Description
CL_CONFIG_CPU_FORCE_PRIVATE_MEM_SIZE	32KB	Forces CL_DEVICE_PRIVATE_MEM_SIZE for the CPU device to be the given value. The value must include the unit; for example: 8MB, 8192KB, 8388608B.

Runtime configuration	Default value	Description
		NOTE You must compile your host application with sufficient stack size.
CL_CONFIG_CPU_FORCE_LOCAL_MEM_SIZE	32KB	Forces CL_DEVICE_LOCAL_MEM_SIZE for CPU device to be the given value. The value needs to be set with size including units, examples: 8MB, 8192KB, 8388608B. **NOTE** You must compile your host application with sufficient stack size. Our recommendation is to set the stack size equal to twice the local memory size to cover possible application and OpenCL Runtime overheads.
CL_CONFIG_CPU_EXPENSIVE_MEM_OPT	0	A bitmap indicating enabled expensive memory optimizations. These optimizations may lead to more JIT compilation time, but give some performance benefit. **NOTE** Currently, only the least significant bit is available. Available bits: • 0: OpenCL address space alias analysis
CL_CONFIG_CPU_STREAMING_ALWAYS	False	Controls whether non-temporal instructions are used.

See Also

Qopenmp compiler option
Thread Affinity Interface

Pass Options to the Linker

Specify Linker Options

This topic describes the options that let you control and customize linking with tools and libraries and define the output of the linker.

Linux

This section describes options specified at compile-time that take effect at link-time to define the output of the `ld` linker. See the `ld` man page for more information on the linker.

Option	Description
`-L`*directory*	Instruct the linker to search *directory* for libraries.
`-Qoption`,*tool*,*list*	Passes an argument list to another program in the compilation sequence, such as the assembler or linker.
`-shared`	Instructs the compiler to build a Dynamic Shared Object (DSO) instead of an executable.
`-shared-libgcc`	`-shared-libgcc` has the opposite effect of `-static-libgcc` . When it is used, the GNU standard libraries are linked in dynamically, allowing the user to override the static linking behavior when the `-static` option is used. **NOTE** Note: By default, all C++ standard and support libraries are linked dynamically.
`-shared-intel`	Specifies that all Intel-provided libraries should be linked dynamically.
`-static`	Causes the executable to link all libraries statically, as opposed to dynamically. When `-static` is not used: • `/lib/ld-linux.so.2` is linked in • all other libs are linked dynamically When `-static` is used: • `/lib/ld-linux.so.2` is not linked in • all other libs are linked statically
`-static-libgcc`	This option causes the GNU standard libraries to be linked in statically.
`-Bstatic` `-Bdynamic`	Either option is placed in the linker command line corresponding to its location on the user command line to control the linking behavior of any library being passed in via the command line.
`-static-intel`	This option causes Intel-provided libraries to be linked in statically. It is the opposite of `-shared-intel`.

Option	Description
`-Wl,`*optlist*	This option passes a comma-separated list (*optlist*) of options to the linker.
`-Xlinker` *val*	This option passes a value (*val*), such as a linker option, an object, or a library, directly to the linker.

Windows

This section describes options specified at compile-time that take effect at link-time.

You can use the `link` option to pass options specifically to the linker at compile time. For example:

```
icx a.cpp libfoo.lib /link -delayload:comct132.dll
```

In this example, the compiler recognizes that `libfoo.lib` is a library that should be linked with `a.cpp`, so it does not need to follow the `link` option on the command line. The compiler does not recognize `-delayload:comct132.dll`, so the `link` option is used to direct the option to the linking phase. On C++, you can use the `Qoption` option to pass options to various tools, including the linker. You can also use `#pragma comment` on C++ to pass options to the linker. Linker options do not work for SYCL kernels, but they do work for host code (including pragmas for linker options). For example:

```
#pragma comment(linker, "/defaultlib:mylib.lib")
```

OR

```
#pragma comment(lib, "mylib.lib")
```

Both examples instruct the compiler to link `mylib.lib` at link time.

Specify Alternate Tools and Paths

This content does not apply to SYCL.

Use the `Qlocation` option to specify an alternate path for a tool. This option accepts two arguments using the following syntax:

Linux

```
-Qlocation,tool,path
```

Windows

```
/Qlocation,tool,path
```

where *tool* designates which compilation tool is associated with the alternate *path*.

tool	Description
`cpp`	Specifies the preprocessor for the compiler.
`c`	Specifies the Intel® oneAPI DPC++/C++ Compiler .
`asm`	Specifies the assembler.
`link`	Specifies the linker.

Use the `Qoption` option to pass an option specified by *optlist* to a *tool*, where *optlist* is a comma-separated list of options. The syntax for this command is:

Linux

```
-Qoption,tool,optlist
```

Windows

```
/Qoption,tool,optlist
```

where

- *tool* designates which compilation tool receives the *optlist*
- *optlist* indicates one or more valid argument strings for the designated program. If the argument is a command-line option, you must include the hyphen. If the argument contains a space or tab character, the entire argument must be enclosed in quotation characters (""). Separate multiple arguments with commas.

Use Configuration Files

You can decrease the time you spend entering command-line options by using the configuration file to automate command-line entries. Configuration files are automatically processed every time you run the Intel® oneAPI DPC++/C++ Compiler. You can insert any valid command-line options into the configuration file. The compiler processes options in the configuration file in the order in which they appear, followed by the specified command-line options when the compiler is invoked.

> **NOTE**
> Options in the configuration file are executed every time you run the compiler. If you have varying option requirements for different projects, use Using Response Files .

Sample Configuration Files

> **NOTE**
> Anytime you instruct the compiler to use a different configuration file, the default configuration file(s) are ignored.

The following examples illustrate basic configuration files.

Linux

```
## Sample icpx.cfg file
  -I/my_headers
```

Windows

```
## Sample icx.cfg file
  /Ic:\my_headers
```

In the Windows examples, the compiler reads the configuration file and invokes the `I` option every time you run the compiler, along with any options specified on the command line.

See Also

Supported Environment Variables
Using Response Files

Use Response Files

You can use response files to:

- Specify options used during particular compilations or projects.
- Save this information in individual files.

Response files are invoked as options on the command line. Options in response files are inserted in the command line at the point where the response file is invoked. Unlike configuration files, which are automatically processed every time you run the compiler, response files must be invoked as an option on the command line. If you create a response file without specifying it on the command line, it will not be invoked.

Sample Response Files

Linux

```
# response file: response1.txt
# compile with these options
  -w0
# end of response1 file
```

```
# response file: response2.txt
# compile with these options
  -O0
# end of response2 file
```

Windows

```
# response file: response1.txt
# compile with these options
  /W0
# end of response1 file
```

```
# response file: response2.txt
# compile with these options
  /Od
# end of response2 file
```

Use response files to decrease the time spent entering command-line options and to ensure consistency by automating command-line entries. Use individual response files to maintain options for specific projects.

Any number of options or file names can be placed on a line in a response file. Several response files can be referenced in the same command line. The following example shows how to specify a response file on the command line:

Linux

```
  icpx @response1.txt prog1.cpp @response2.txt prog2.cpp
```

Windows

```
  icx @response1.txt prog1.cpp @response2.txt prog2.cpp
```

NOTE
An "at" sign (`@`) must precede the name of the response file on the command line.

See Also
Using Configuration Files

*Global Symbols and Visibility Attributes for Linux**

A global symbol is one that is visible outside the compilation unit (single source file and its include files) in which it is declared. In C/C++, this means anything declared at file level without the `static` keyword. For example:

```
int x = 5;         // global data definition
extern int y;      // global data reference
int five()         // global function definition
  { return 5; }
extern int four(); // global function reference
```

A complete program consists of a main program file and possibly one or more shareable object (`.so`) files that contain the definitions for data or functions referenced by the main program. Similarly, shareable objects might reference data or functions defined in other shareable objects. Shareable objects are so called because if more than one simultaneously executing process has the shareable object mapped into its virtual memory, there is only one copy of the read-only portion of the object resident in physical memory. The main program file and any shareable objects that it references are collectively called the components of the program.

Each global symbol definition or reference in a compilation unit has a `visibility` attribute that controls how (or if) it may be referenced from outside the component in which it is defined. The `visibility` attribute accepts one of five keywords values:

- **`external`:** The compiler must treat the symbol as though it is defined in another component. For a definition, this means that the compiler must assume that the symbol will be overridden (preempted) by a definition of the same name in another component. See Symbol Preemption. If a function symbol has external visibility, the compiler knows that it must be called indirectly and can inline the indirect call stub.
- **`default`:** Other components can reference the symbol. Furthermore, the symbol definition may be overridden (preempted) by a definition of the same name in another component.
- **`protected`:** Other components can reference the symbol, but it cannot be preempted by a definition of the same name in another component.
- **`hidden`:** Other components cannot directly reference the symbol. However, its address might be passed to other components indirectly (for example, as an argument to a call to a function in another component, or by having its address stored in a data item reference by a function in another component).
- **`internal`:** The symbol cannot be referenced outside its defining component, either directly or indirectly.

Static local symbols (in C/C++, declared at file scope or elsewhere with the keyword static) usually have `hidden` visibility— they cannot be referenced directly by other components (or, for that matter, other compilation units within the same component), but they might be referenced indirectly.

NOTE
Visibility applies to references as well as definitions. A symbol reference's visibility attribute is an assertion that the corresponding definition will have that visibility.

Specify Symbol Visibility Explicitly

You can explicitly set the visibility of an individual symbol using the `visibility` attribute on a data or function declaration. For example:

```
int i __attribute__ ((visibility("default")));
void __attribute__ ((visibility("hidden"))) x () {...}
extern void y() __attribute__ ((visibility("protected")));
```

The value of the `visibility` declaration attribute overrides the default set by the options `-fpic` or `-fno-common` .

Save Compiler Information in Your Executable

Linux

To view the information stored in the object file, use the `objdump` command. For example:

```
objdump -sj comment a.out
strings -a a.out | grep comment:
```

Windows

To view the linker directives stored in string format in the object file, use the `link` command. For example:

```
link /dump /directives filename.obj
```

In the output, the ?-comment linker directive displays the compiler version information. To search your executable for compiler information, use the `findstr` command. For example, to search for any strings that contain the substring "Compiler":

```
findstr "Compiler" filename.exe
```

Link Debug Information

Linux*

Use option `g` at compile time to tell the compiler to generate symbolic debugging information in the object file.

Use option `gsplit-dwarf` to create a separate object file containing DWARF debug information. Because the DWARF object file is not used by the linker, this reduces the amount of debug information the linker must process and it results in a smaller executable file. See `gsplit-dwarf` for detailed information.

Windows*

Use option `Z7` at compile time or option `debug` at link time to tell the compiler to generate symbolic debugging information in the object file. Alternately, use option `Zi` at link time to generate executables with debug information in the `.pdb` file.

Ahead of Time Compilation

Ahead of Time (AOT) Compilation is a helpful feature for your development lifecycle or distribution time. It benefits you when you know beforehand what your target device is going to be at application execution time. The AOT feature provides the following benefits:

- No additional compilation time is done when running your application.
- No just-in-time (JIT) bugs encountered due to compilation for the target device, because this step is skipped with AOT compilation.
- Your final code, executing on the target device, can be tested as-is before you deliver it to end-users.

A program built with AOT compilation for a specific target device will not run on a non-specific device. You must detect the proper target device at runtime and report an error if the targeted device is not present. The use of exception handling with an asynchronous exception handler is recommended.

SYCL supports AOT compilation for the following targets: Intel® CPUs, Intel® Processor Graphics (Gen9 or above), and Intel® FPGA.

Prerequisites

To target a GPU with the AOT feature, you must have the OpenCL™ Offline Compiler (OCLOC) tool installed. OCLOC can generate binaries that use OpenCL™ or the Intel® oneAPI Level Zero (Level Zero) backend.

Linux

OCLOC is not packaged with the Linux version of Intel® oneAPI DPC++/C++ Compiler and must be installed separately. Refer to Install OpenCL™ Offline Compiler (OCLOC) for details.

Windows

OCLOC is packaged with the Windows version of Intel® oneAPI DPC++/C++ Compiler.

AOT Compilation Supported Options

Use the following options to target a specific device for AOT compilation:

- `-fsycl-targets`, to specify the device target
- `-Xs`, to pass options to the backend tool

`-Xs` is a general device target option. If there are multiple targets desired (example: `-fsycl-targets=spir64_gen,spir64_x86_64`) the options specified with `-Xs` apply to all targets. This is not desired for multiple targets. You can use `-Xsycl-target-backend=spir64_gen <option>` and `-Xsycl-target-backend=spir64_x86_64 <option>` to add specificity.

When using Ahead of Time (AOT) compilation, the options passed with `-Xs` are not compiler options.

To see a list of the options you can pass with `-Xs` when using AOT, specify `-fsycl-help=gen`, `-fsycl-help=x86_64`, or `-fsycl-help=fpga` on the command line.

Use AOT for the Target Device (Intel® CPUs)

Use the following option arguments to specify Intel® CPUs as the target device for AOT compilation:

- `-fsycl-targets=spir64_x86_64`
- `-Xs "-march=<arch>"`, where `<arch>` is one of the following:

Switch	Display Name
`avx`	Intel® Advanced Vector Extensions (Intel® AVX)

Switch	Display Name
avx2	Intel® Advanced Vector Extensions 2 (Intel® AVX2)
avx512	Intel® Advanced Vector Extensions 512 (Intel® AVX-512)
sse4.2	Intel® Streaming SIMD Extensions 4.2 (Intel® SSE4.2)

The following examples tell the compiler to generate code that uses Intel® AVX2 instructions:

Linux

```
dpcpp -fsycl-targets=spir64_x86_64 -Xs "-march=avx2" main.cpp
```

Windows

```
dpcpp-cl /EHsc -fsycl-targets=spir64_x86_64 -Xs "-march=avx2" main.cpp
```

Build an Application with Multiple Source Files for CPU Targeting

Method 1: Compile your normal files (with no SYCL kernels) to create host objects. Then compile the file with the kernel code and link it with the rest of the application.

Linux

```
dpcpp -c main.cpp
dpcpp -fsycl-targets=spir64_x86_64 -Xs "-march=avx2" mandel.cpp main.o
```

Windows

```
dpcpp-cl -c /EHsc main.cpp
dpcpp-cl /EHsc -fsycl-targets=spir64_x86_64 -Xs "-march=avx2" mandel.cpp main.obj
```

Method 2: Compile the file with the kernel code and create a fat object. Then compile the rest of the files and linking to create a fat executable:

NOTE Currently, Method 2 only works on a HOST selector.

Linux

```
dpcpp -c -fsycl-targets=spir64_x86_64 -Xs "-march=avx2" mandel.cpp
dpcpp main.cpp mandel.o -fsycl-targets=spir64_x86_64 -Xs "-march=avx2"
```

Windows

```
dpcpp-cl -c /EHsc -fsycl-targets=spir64_x86_64 -Xs "-march=avx2" mandel.cpp
dpcpp-cl /EHsc main.cpp mandel.obj -fsycl-targets=spir64_x86_64 -Xs "-march=avx2"
```

Use AOT for Integrated Graphics (Intel® GPU)

Use the following option arguments to specify Intel® GPU as the target device for AOT compilation:

- `-fsycl-targets=spir64_gen`
- `-Xs "-device <arch>"` option, where `<arch>` is the target device. Possible values:

Switch	Display Name
skl	6th generation Intel® Core™ processor (Skylake with Intel® Processor Graphics Gen9)

Switch	Display Name
`kbl`	7th generation Intel® Core™ processor (Kaby Lake with Intel® Processor Graphics Gen9)
`cfl`	8th generation Intel® Core™ processor (Coffee Lake with Intel® Processor Graphics Gen9)
`glk`	Gemini Lake with Intel® Processor Graphics Gen9
`icllp`	10th generation Intel® Core™ processor (Ice Lake with Intel® Processor Graphics Gen11)
`tgllp`	11th generation Intel® Core™ processor (Tiger Lake with Intel® Processor Graphics Gen12)
`dg1`	Intel® Iris® X^e MAX graphics
`Gen9`	Intel® Processor Graphics Gen9
`Gen11`	Intel® Processor Graphics Gen11
`Gen12LP`	Intel® Processor Graphics Gen12 (Lower Power)
`adls`	12th generation Intel® Core™ processor (Alder Lake S with Intel® Processor Graphics Gen12.2)
`aldp`	12th generation Intel® Core™ processor (Alder Lake P with Intel® Processor Graphics Gen12.2)

To see the complete list of supported target device types for your installed version of OCLOC, run:

```
ocloc compile --help
```

If multiple target devices are listed in the compile command, the Intel® oneAPI DPC++/C++ Compiler compiles for each of these targets and creates a fat-binary that contains all the device binaries produced this way.

Examples of supported `-device` patterns:

Linux

- To compile for a single target, using `skl` as an example, use:

```
dpcpp -fsycl-targets=spir64_gen -Xs "-device skl" vector-add.cpp
```

- To compile for two targets, using `skl` and `icllp` as examples, use:

```
dpcpp -fsycl-targets=spir64_gen -Xs "-device skl,icllp" vector-add.cpp
```

- To compile for all the targets known to OCLOC, use:

```
dpcpp -fsycl-targets=spir64_gen -Xs "-device *" vector-add.cpp
```

Windows

- To compile for a single target, using `skl` as an example, use:

```
dpcpp-cl /EHsc -fsycl-targets=spir64_gen -Xs "-device skl" vector-add.cpp
```

- To compile for two targets, using `skl` and `icllp` as examples, use:

```
dpcpp-cl /EHsc -fsycl-targets=spir64_gen -Xs "-device skl,icllp" vector-add.cpp
```

- To compile for all the targets known to OCLOC, use:

```
dpcpp-cl /EHsc -fsycl-targets=spir64_gen -Xs "-device *" vector-add.cpp
```

Build an Application with Multiple Source Files for GPU Targeting

Method 1: Compile your normal files (with no SYCL kernels) to create host objects. Then compile the file with the kernel code and link it with the rest of the application.

Linux

```
dpcpp -c main.cpp
dpcpp -fsycl-targets=spir64_gen -Xs "-device *" mandel.cpp main.o
```

Windows

```
dpcpp-cl -c /EHsc main.cpp
dpcpp-cl /EHsc -fsycl-targets=spir64_gen -Xs "-device *" mandel.cpp main.obj
```

Method 2: Compile the file with the kernel code and create a fat object. Then compile the rest of the files and linking to create a fat executable:

> **NOTE** Currently, Method 2 only works on a HOST selector.

Linux

```
dpcpp -c -fsycl-targets=spir64_gen mandel.cpp
dpcpp main.cpp mandel.o -fsycl-targets=spir64_gen -Xs "-device *"
```

Windows

```
dpcpp-cl -c /EHsc -fsycl-targets=spir64_gen mandel.cpp
dpcpp-cl /EHsc main.cpp mandel.obj -fsycl-targets=spir64_gen -Xs "-device *"
```

Use AOT in Microsoft Visual Studio

You can use Microsoft Visual Studio for compiling and linking. Set the following flags to use AOT compilation for CPU or GPU:

CPU:

- To compile, in the dialog box, select: **Configuration Properties** > **DPC++** > **General** > **Specify SYCL offloading targets for AOT compilation**
- To link, in the dialog box, select: **Configuration Properties** > **Linker** > **General** > **Specify CPU Target Device for AOT compilation**

GPU:

- To compile, in the dialog box, select: **Configuration Properties** > **DPC++** > **General** > **Specify SYCL offloading targets for AOT compilation**
- To link, in the dialog box, select: **Configuration Properties** > **Linker** > **General** > **Specify GPU Target Device for AOT compilation**

See Also

`-fsycl-targets` compiler option

`-Xs` compiler option

Device Offload Compilation Considerations

SYCL compilation performs a compile that generates both host and target binaries for a single source file. The SYCL compilation flow generates file dependencies from the device compilation to the host compilation. These dependent files are considered to be integration files that are included in the host side compilation.

A file, called an integration footer, is added to the end of the original source file before being compiled. To accomplish this process, a new temporary source file is generated and is considered the host source file for the compilation. The file is a new source dependency and could break your build environments that track the generated files during a compilation. These build environments need to be configured in the SYCL space for the additional intermediate file to be part of the compilation flow.

The location of the additional file is generated in the common temporary file location, specified by the `TMP` then `TEMP` environment variables.

Use a Third-Party Compiler as a Host Compiler for SYCL Code

This content describes the steps needed to use an external host compiler (G++) along with the Intel® oneAPI DPC++/C++ Compiler.

In this example, you will use a host compiler to generate the host objects and perform the final link. The host compiler needs to know where to find the required headers and libraries. Follow the example instructions to build a SYCL program using a G++ Compiler (`g++`) for host code and an Intel® oneAPI DPC++/C++ Compiler (`dpcpp`) for SYCL code.

For Linux

This example includes the following:

- `a.cpp`: SYCL code
- `b.cpp`: SYCL code
- `main.cpp`: C++ code

1. Follow the Get Started with the Intel® oneAPI Base Toolkit for Linux guide to set up the build environment:

   NOTE The build environment requires GCC version 5.1 or above to be installed and accessible.

```
source /opt/intel/oneapi/setvars.sh
```

2. Set up the headers and library locations:

```
export LIBDIR=<Location of libsycl.so>
export INCLUDEDIR=<Location of SYCL headers>
```

3. Build the objects for your device:

```
dpcpp -c a.cpp -fPIC -o a.o
dpcpp -c b.cpp -fPIC -o b.o
```

4. Create the integration header files (used by the host compiler):

```
dpcpp -fsycl-device-only -Xclang -fsycl-int-header=a_host.h a.cpp
dpcpp -fsycl-device-only -Xclang -fsycl-int-header=b_host.h b.cpp
```

5. Create the host objects:

```
g++ -std=c++17 -c a.cpp -o a_host.o -include a_host.h -fPIC -I$INCLUDEDIR
g++ -std=c++17 -c b.cpp -o b_host.o -include b_host.h -fPIC -I$INCLUDEDIR
```

6. Compile other C++ code (or non-SYCL code) using G++:

```
g++ -std=c++17 main.cpp -c -fPIC -I$INCLUDEDIR
```

7. Create a device object:

```
dpcpp -fPIC -fsycl -fsycl-link a.o b.o -o device.o
```

8. Create an archive `libuser.a` that contains the necessary host and device objects:

   NOTE This step is optional.

```
ar -rcs libuser.a a_host.o b_host.o device.o
```

9. Perform the final link to create a `final.exe` executable:

```
g++ main.o a_host.o b_host.o device.o -L$LIBDIR -lOpenCL -lsycl -o finalexe.exe
```

10. Build the `final.exe` with an archive:

    NOTE This step is optional.

```
g++ main.o -Wl,--whole-archive libuser.a -Wl,--no-whole-archive -L$LIBDIR -lOpenCL -lsycl -o
finalexe.exe
```

For Windows

Windows is not supported in this release.

Options

The compiler has two options that let you use an external compiler to perform host side compilation. The options are:

- `fsycl-host-compiler`: Tells the compiler to use the specified compiler for host compilation of the performed offloading compilation.
- `fsycl-host-compiler-options`: Passes options to the compiler specified by the option `fsycl-host-compiler`.

See Also

fsycl-host-compiler
fsycl-host-compiler-options

Part V

Optimization and Programming

This section contains information about features related to code optimization and program performance improvement.

Extensions

For the latest information about extensions, see the oneAPI Specification and the SYCL Reference.

OpenMP* Support

The Intel® oneAPI DPC++/C++ Compiler supports most of the OpenMP* Application Programming Interface versions 5.0 and 5.1. For the complete OpenMP specification, read the specifications available from the OpenMP web site (http://www.openmp.org; see OpenMP Specifications on that site). The descriptions of OpenMP language characteristics in this documentation often use terms defined in that specification.

The OpenMP API provides symmetric multiprocessing (SMP) with the following major features:

- Relieves you from implementing the low-level details of iteration space partitioning, data sharing, thread creation, scheduling, or synchronization.
- Provides the benefit of performance available from shared memory multiprocessor and multi-core processor systems on all supported Intel architectures, including those processors with Intel® Hyper-Threading Technology (Intel® HT Technology).

The compiler performs transformations to generate multithreaded code based on your placement of OpenMP pragmas in the source program, making it simple to add threading to existing software. The compiler compiles parallel programs and supports the industry-standard OpenMP pragmas.

The compiler provides Intel®-specific extensions to the OpenMP specification including run-time library routines and environment variables. A summary of the compiler options appear in the OpenMP Options Quick Reference.

Parallel Processing with OpenMP

To compile with the OpenMP API, add the pragmas to your code. The compiler processes the code and internally produces a multithreaded version which is then compiled into an executable with the parallelism implemented by threads that execute parallel regions or constructs.

Using Other Compilers

The OpenMP specification does not define interoperability of multiple implementations, so the OpenMP implementation supported by other compilers and OpenMP support in the Intel® oneAPI DPC++/C++ Compiler might not be interoperable. Even if you compile and build the entire application with one compiler, be aware that different compilers might not provide OpenMP source compatibility that enable you to compile and link the same set of application sources with a different compiler and get the expected parallel execution results.

Add OpenMP* Support

To add OpenMP* support to your application, do the following:

1. Add the appropriate OpenMP pragmas to your source code.
2. Compile the application with the `/Qopenmp` (Windows*) or `-qopenmp` (Linux*) option.
3. For applications with large local or temporary arrays, you may need to increase the stack space available at runtime. In addition, you may need to increase the stack allocated to individual threads by using the `OMP_STACKSIZE` environment variable or by setting the corresponding library routines.

You can set other environment variables to control multi-threaded code execution.

OpenMP Pragma Syntax

To add OpenMP support to your application, first declare the OpenMP header and then add appropriate OpenMP pragmas to your source code.

To declare the OpenMP header, add the following in your code:

```
#include <omp.h>
```

OpenMP pragmas use a specific format and syntax. Intel Extension Routines to OpenMP describes the OpenMP extensions to the specification that have been added to the Intel® oneAPI DPC++/C++ Compiler.

To use pragmas in your source, use this syntax:

```
<prefix> <pragma> [<clause>, ...] <newline>
```

where:

- *<prefix>* - Required for all OpenMP pragmas. The prefix must be `#pragma omp`.
- *<pragma>* - A valid OpenMP pragma. Must immediately follow the prefix.
- [*<clause>*] - Optional. Clauses can be in any order and repeated as necessary, unless otherwise restricted.
- *<newline>* - A required component of pragma syntax. It precedes the structured block that is enclosed by this pragma.

The pragmas are interpreted as comments if you omit the `/Qopenmp` (Windows) or `-qopenmp` (Linux) option.

The following example demonstrates one way of using an OpenMP pragma to parallelize a loop:

```
#include <omp.h>
void simple_omp(int *a){
  int i;
  #pragma omp parallel for
  for (i=0; i<1024; i++)
    a[i] = i*2;
}
```

Compile the Application

The `/Qopenmp` (Windows) or `-qopenmp` (Linux) option enables the parallelizer to generate multi-threaded code based on the OpenMP pragmas in the source. The code can be executed in parallel on single processor, multi-processor, or multi-core processor systems.

The `/Qopenmp` (Windows) or `-qopenmp` (Linux) option works with both `-O0` (Linux) and `/Od` (Windows*) and with any optimization level of `O1`, `O2` and `O3`.

Specifying `-O0` (Linux) or `/Od` (Windows) with the `/Qopenmp` (Windows) or `-qopenmp` (Linux) option helps to debug OpenMP applications.

Compile your application using a command similar to one of the following:

Linux

```
icpx -qopenmp source_file
```

Windows

```
icx /Qopenmp source_file
```

For example, to compile the previous code example without generating an executable, use the `c` option:

Linux

```
icpx -qopenmp -c parallel.cpp
```

Windows

```
icx /Qopenmp /c parallel.c
```

To build your application with target offload support (introduced since OpenMP 4.0) use compiler options to specify the target for which the regions marked with OpenMP "target" pragmas must be compiled. For example:

Linux

```
icpx -qopenmp -fopenmp-targets=spir64 offload.cpp
```

Windows

```
icx /Qopenmp /Qopenmp-targets=spir64 offload.c
```

Refer to Get Started with OpenMP* Offload to GPU for the Intel® oneAPI DPC/C++ Compiler and Intel® Fortran Compiler for more information.

Configure the OpenMP Environment

Before you run the multi-threaded code, you can set the number of desired threads using the OpenMP environment variable, `OMP_NUM_THREADS`.

See Also

c compiler option
O compiler option
OpenMP* Examples
qopenmp, Qopenmp compiler option
Supported Environment Variables

Parallel Processing Model

A program containing OpenMP* pragmas begins execution as a single thread, called the initial thread of execution. The initial thread executes sequentially until the first parallel construct is encountered.

The `omp parallel` pragma defines the extent of the parallel construct. When the initial thread encounters a parallel construct, it creates a team of threads, with the initial thread becoming the primary thread of the team. All program statements enclosed by the parallel construct are executed in parallel by each thread in the team, including all routines called from within the enclosed statements.

The statements enclosed lexically within a construct define the static extent of the construct. The dynamic extent includes all statements encountered during the execution of a construct by a thread, including all called routines.

When a thread encounters the end of a structured block enclosed by a parallel construct, the thread waits until all threads in the team have arrived. When that happens the team is dissolved, and only the primary thread continues execution of the code following the parallel construct. The other threads in the team enter a wait state until they are needed to form another team. You can specify any number of parallel constructs in a single program. As a result, thread teams can be created and dissolved many times during program execution.

The following example illustrates, from a high level, the execution model for the OpenMP constructs. The comments in the code explain the structure of each construct or section.

```
main() {                     // Begin serial execution.
  ...                        // Only the initial thread executes
  #pragma omp parallel       // Begin a parallel construct and form a team.
  {
    #pragma omp sections     // Begin a worksharing construct.
    {
       #pragma omp section   // One unit of work.
       {...}
       #pragma omp section   // Another unit of work.
       {...}
    }                        // Wait until both units of work complete.
    ...                      // This code is executed by each team member.
    #pragma omp for nowait   // Begin a worksharing Construct
    for(...) {               // Each iteration chunk is unit of work.
      ...                    // Work is distributed among the team members.
    }                        // End of worksharing construct.
                             // nowait was specified so threads proceed.
    #pragma omp critical     // Begin a critical section.
    {...}                    // Only one thread executes at a time.
    ...                      // This code is executed by each team member.
    #pragma omp barrier      // Wait for all team members to arrive.
    ...                      // This code is executed by each team member.
  }                          // End of Parallel Construct
                             // Disband team and continue serial execution.
  ...                        // Possibly more parallel constructs.
}                            // End serial execution.
```

Use Orphaned Pragmas

In routines called from within parallel constructs, you can also use pragmas. Pragmas that are not in the static extent of the parallel construct, but are in the dynamic extent, are called orphaned pragmas. Orphaned pragmas allow you to execute portions of your program in parallel with only minimal changes to the sequential version of the program. Using this functionality, you can code parallel constructs at the top levels of your program call tree and use directives to control execution in any of the called routines. For example:

```
int main(void) {
  #pragma omp parallel {
     phase1();
  }
}

void phase1(void) {
  #pragma omp for // This is an orphaned pragma.
  for(i=0; i < n; i++) { some_work(i); }
}
```

This is an orphaned `omp for` loop pragma since the parallel region is not lexically present in routine `phase1`.

Data Environment

You can control the data environment of OpenMP constructs by using data environment clauses supported by the construct. You can also privatize named global-lifetime objects by using the `threadprivate` pragma.

Refer to the OpenMP specification for the full list of data environment clauses. Some commonly used ones include:

- default
- shared
- private
- firstprivate
- lastprivate
- reduction
- linear
- map

You can use several pragma clauses to control the data scope attributes of variables for the duration of the construct in which you specify them; however, if you do not specify a data scope attribute clause on a pragma, the behavior for the variable is determined by the default scoping rules, which are described in the OpenMP specification, for the variables affected by the directive.

Determine How Many Threads to Use

For applications where the workload depends on application input that can vary widely, delay the decision about the number of threads to employ until runtime when the input sizes can be examined. Examples of workload input parameters that affect the thread count include things like matrix size, database size, image/video size and resolution, depth/breadth/bushiness of tree-based structures, and size of list-based structures. Similarly, for applications designed to run on systems where the processor count can vary widely, defer choosing the number of threads to employ until application runtime when the machine size can be examined.

For applications where the amount of work is unpredictable from the input data, consider using a calibration step to understand the workload and system characteristics to aid in choosing an appropriate number of threads. If the calibration step is expensive, the calibration results can be made persistent by storing the results in a permanent place like the file system.

Avoid simultaneously using more threads than the number of processing units on the system. This situation causes the operating system to multiplex threads on the processors and typically yields sub-optimal performance.

When developing a library as opposed to an entire application, provide a mechanism whereby the user of the library can conveniently select the number of threads used by the library, because it is possible that the user has outer-level parallelism that renders the parallelism in the library unnecessary or even disruptive.

Use the `num_threads` clause on parallel regions to control the number of threads employed and use the `if` clause on parallel regions to decide whether to employ multiple threads at all. The `omp_set_num_threads()` routine can also be used, but it also affects parallel regions created by the calling thread. The `num_threads` clause is local in its effect, so it does not impact other parallel regions. The disadvantages of explicitly setting the number of threads are:

1. In a system with a large number of processors, your application will use some but not all of the processors.
2. In a system with a small number of processors, your application may force over subscription that results in poor performance.

The Intel OpenMP runtime will create the same number of threads as the available number of logical processors unless you use the omp_set_num_threads() routine. To determine the actual limits, use omp_get_thread_limit() and omp_get_max_active_levels(). Developers should carefully consider their thread usage and nesting of parallelism to avoid overloading the system. The `OMP_THREAD_LIMIT` environment variable limits the number of OpenMP threads to use for the whole OpenMP program. The `OMP_MAX_ACTIVE_LEVELS` environment variable limits the number of active nested parallel regions.

Binding Sets and Binding Regions

The binding task set for an OpenMP construct is the set of tasks that are affected by, or provide the context for, the execution of its region. It can be all tasks, the current team tasks, all tasks of the current team that are generated in the region, the binding implicit task, or the generating task.

The binding thread set for an OpenMP construct is the set of threads that are affected by, or provide the context for, the execution of its region. It can be all threads on a device, all threads in a contention group, all primary threads executing an enclosing teams region, the current team, or the encountering thread.

The binding region for an OpenMP construct is the enclosing region that determines the execution context and the scope of the effects of the directive:

- The binding region for an omp ordered construct is the innermost enclosing omp for loop region.
- The binding region for a omp taskwait construct is the innermost enclosing omp task region.
- For all other constructs for which the binding thread set is the current team or the binding task set is the current team tasks, the binding region is the innermost enclosing region.
- For constructs for which the binding task set is the generating task, the binding region is the region of the generating task.
- A omp parallel construct need not be active to be a binding region.
- A construct need not be explicit to be a binding region.
- A region never binds to any region outside of the innermost enclosing parallel region.

Worksharing Using OpenMP*

To get the maximum performance benefit from a processor with multi-core and Intel® Hyper-Threading Technology (Intel® HT Technology), an application needs to be executed in parallel. Parallel execution requires threads, and threading an application is not a simple thing to do; using OpenMP* can make the process a lot easier. Using the OpenMP pragmas, most loops with no loop-carried dependencies can be threaded with one simple statement. This topic explains how to start using OpenMP to parallelize loops, which is also called worksharing.

Options that use OpenMP are available for both Intel® and non-Intel microprocessors, but these options may perform additional optimizations on Intel® microprocessors than they perform on non-Intel microprocessors. The list of major, user-visible OpenMP constructs and features that may perform differently on Intel® microprocessors than on non-Intel microprocessors includes: locks (internal and user visible), the SINGLE construct, barriers (explicit and implicit), parallel loop scheduling, reductions, memory allocation, and thread affinity and binding.

Most loops can be threaded by inserting one pragma immediately prior to the loop. Further, by leaving the details to the Intel® oneAPI DPC++/C++ Compiler and OpenMP, you can spend more time determining which loops should be threaded and how to best restructure the algorithms for maximum performance. The maximum performance of OpenMP is realized when it is used to thread hotspots, the most time-consuming loops in your application.

The power and simplicity of OpenMP is demonstrated by looking at an example. The following loop converts a 32-bit RGB (red, green, blue) pixel to an 8-bit gray-scale pixel. One pragma, which has been inserted immediately before the loop, is all that is needed for parallel execution.

```
#pragma omp parallel for
for (i=0; i < numPixels; i++) {
  pGrayScaleBitmap[i] = (unsigned BYTE)
    (pRGBBitmap[i].red * 0.299 +
     pRGBBitmap[i].green * 0.587 +
     pRGBBitmap[i].blue * 0.114);
}
```

First, the example uses worksharing, which is the general term used in OpenMP to describe distribution of work across threads. When worksharing is used with the `for` construct, as shown in the example, the iterations of the loop are distributed among multiple threads so that each loop iteration is executed exactly once with different iterations executing if there is more than one available threads. The `for` construct on its own only distributes the loop iterations among existing threads. The example uses a `parallel for` construct, which combines `parallel` and `for` constructs to first create a team of threads and then distribute the loop iterations among the threads. Since there is no explicit `num_threads` clause, OpenMP determines the number of threads to create and how to best create, synchronize, and destroy them. OpenMP places the following five restrictions on which loops can be threaded:

- The loop variable must be of type signed or unsigned integer, random access iterator, or pointer.
- The comparison operation must be in the form `loop_variable <, <=, >, >=, or != loop_invariant_expression` of a compatible type.
- The third expression or increment portion of the `for` loop must be either addition or subtraction by a loop invariant value.
- If the comparison operation is < or <=, the loop variable must increment on every iteration; conversely, if the comparison operation is > or >=, the loop variable must decrement on every iteration.
- The loop body must be single-entry-single-exit, meaning no jumps are permitted from inside to outside the loop, with the exception of the `exit` statement that terminates the whole application. If the statements `goto` or `break` are used, the statements must jump within the loop, not outside it. Similarly, for exception handling, exceptions must be caught within the loop.

Although these restrictions might sound somewhat limiting, non-conforming loops can frequently be rewritten to follow these restrictions.

Basics of Compilation

Using the OpenMP pragmas requires an OpenMP-compatible compiler and thread-safe libraries. Adding the `/Qopenmp` (Windows*) or `-qopenmp` (Linux*) option to the compiler instructs the compiler to pay attention to the OpenMP pragmas and to generate multi-threaded code. If you omit the `/Qopenmp` (Windows) or `-qopenmp` (Linux) option, the compiler will ignore OpenMP pragmas, which provides a very simple way to generate a single-threaded version without changing any source code. To compile programs containing target and related constructs for offloading to a GPU, the `-fopenmp-targets=spir64` and `/Qopenmp-targets:spir64` flags are needed on Linux and Windows respectively.

For conditional compilation, the compiler defines the `_OPENMP` macro. If needed, the macro can be tested as shown in the following example.

```
#ifdef _OPENMP
   fn();
#endif
```

A Few Simple Examples

The following examples illustrate how simple OpenMP is to use. In common practice, additional issues need to be addressed, but these examples illustrate a good starting point.

In the first example, the loop clips an array to the range from 0 to 255.

```
// clip an array to 0 <= x <= 255
for (i=0; i < numElements; i++) {
  if (array[i] < 0)
  array[i] = 0;
  else if (array[i] > 255)
    array[i] = 255;
}
```

You can thread it using a single OpenMP pragma; insert the pragma immediately prior to the loop:

```
#pragma omp parallel for
for (i=0; i < numElements; i++) {
  if (array[i] < 0)
  array[i] = 0;
  else if (array[i] > 255)
    array[i] = 255;
}
```

In the second example, the loop generates a table of square roots for the numbers from 0 to 100.

```
double value;
double roots[100];
for (value = 0.0; value < 100.0; value ++) { roots[(int)value] = sqrt(value); }
```

Thread the loop by changing the loop variable to a signed integer or unsigned integer and inserting a `#pragma omp parallel for` pragma.

```
int value;
double roots[100];
#pragma omp parallel for
for (value = 0; value < 100; value ++) { roots[value] = sqrt((double)value); }
```

Avoid Data Dependencies and Race Conditions

When a loop meets all five loop restrictions (listed above) and the compiler threads the loop, the loop still might not work correctly due to the existence of data dependencies.

Data dependencies exist when different iterations of a loop (more specifically a loop iteration that is executed on a different thread) read or write the same location in shared memory. Consider the following example that calculates factorials.

```
// Each loop iteration writes a value that a different iteration reads.
#pragma omp parallel for
for (i=2; i < 10; i++) { factorial[i] = i * factorial[i-1]; }
```

The compiler will thread this loop, but the threading will fail because at least one of the loop iterations is data-dependent upon a different iteration. This situation is referred to as a race condition. Race conditions can only occur when using shared resources (like memory) and parallel execution. To address this problem either rewrite the loop or pick a different algorithm, one that does not contain the race condition.

Race conditions are difficult to detect because, for a given case or system, the threads might win the race in the order that happens to make the program function correctly. Because a program works once does not mean that the program will work under all conditions. Testing your program on various machines, some with Intel® Hyper-Threading Technology and some with multiple physical processors, is a good starting point to help identify race conditions.

Traditional debuggers are useless for detecting race conditions because they cause one thread to stop the race while the other threads continue to significantly change the runtime behavior; however, thread checking tools can help.

Manage Shared and Private Data

Nearly every loop (in real applications) reads from or writes to memory; it's your responsibility, as the developer, to instruct the compiler what memory should be shared among the threads and what memory should be kept private. When memory is identified as shared, all threads access the same memory location. When memory is identified as private, however, a separate copy of the variable is made for each thread to access in private. When the loop ends, the private copies are destroyed. By default, all variables are shared except for the loop variable, which is private.

Memory can be declared as private in two ways:

- Declare the variable inside the loop-really inside the parallel OpenMP pragma-without the static keyword.
- Specify the private clause on an OpenMP pragma.

The following loop fails to function correctly because the variable *temp* is shared. It should be private.

```
// Variable temp is shared among all threads, so while one thread
// is reading variable temp another thread might be writing to it
#pragma omp parallel for
for (i=0; i < 100; i++) {
```

```
  temp = array[i];
  array[i] = do_something(temp);
}
```

The following two examples both declare the variable *temp* as private memory, which solves the problem.

```
#pragma omp parallel for
for (i=0; i < 100; i++) {
  int temp; // variables declared within a parallel construct
            // are, by definition, private
  temp = array[i];
  array[i] = do_something(temp);
}
```

The *temp* variable can also be made private in the following way:

```
#pragma omp parallel for private(temp)
for (i=0; i < 100; i++) {
  temp = array[i];
  array[i] = do_something(temp);
}
```

Every time you use OpenMP to parallelize a loop, you should carefully examine all memory references, including the references made by called functions. Variables declared within a parallel construct are defined as private except when they are declared with the `static` declarator, because static variables are not allocated on the stack.

Reductions

Loops that accumulate a value are fairly common, and OpenMP has a specific clause to accommodate them. Consider the following loop that calculates the sum of an array of integers.

```
sum = 0;
for (i=0; i < 100; i++) {
  sum += array[i]; // this variable needs to be shared to generate
                   // the correct results, but private to avoid
                   // race conditions from parallel execution
}
```

The variable sum in the previous loop must be shared to generate the correct result, but it also must be private to permit access by multiple threads. OpenMP provides the `reduction` clause that is used to efficiently combine the mathematical reduction of one or more variables in a loop. The following example demonstrates how the loop can use the `reduction` clause to generate the correct results.

```
sum = 0;
#pragma omp parallel for reduction(+:sum)
for (i=0; i < 100; i++) { sum += array[i]; }
```

In the case of the example listed above, the reduction provides private copies of the variable *sum* for each thread, and when the threads exit, it adds the values together and places the result in the one global copy of the variable.

The following table lists the possible reduction operations, along with their initial values (mathematical identity values).

Operation	***private* Variable Initialization Value**
+ (addition)	0
- (subtraction)	0

Operation	*private* Variable Initialization Value
* (multiplication)	1
& (bitwise and)	~0
| (bitwise or)	0
^ (bitwise exclusive or)	0
&& (conditional and)	1
|| (conditional or)	0

Multiple reductions in a loop are possible by specifying comma-separated variables and operations on a given `parallel` construct. Reduction variables must meet the following requirements:

- can be listed in just one reduction.
- cannot be declared constant.
- cannot be declared private in the `parallel` construct.

Load Balancing and Loop Scheduling

Load balancing, the equal division of work among threads, is among the most important attributes for parallel application performance. Load balancing is extremely important, because it ensures that the processors are busy most, if not all, of the time. Without a balanced load, some threads may finish significantly before others, leaving processor resources idle and wasting performance opportunities.

Within loop constructs, poor load balancing is often caused by variations in compute time among loop iterations. It is usually easy to determine the variability of loop iteration compute time by examining the source code. In most cases, you will see that loop iterations consume a uniform amount of time. When that is not true, it may be possible to find a set of iterations that consume similar amounts of time. For example, sometimes the set of all even iterations consumes about as much time as the set of all odd iterations. Similarly, it might be the case that the set of the first half of the loop consumes about as much time as the second half. In contrast, it might be impossible to find sets of loop iterations that have a uniform execution time. Regardless of the case, you should provide this extra loop scheduling information to OpenMP so it can better distribute the iterations of the loop across the threads (and therefore processors) for optimum load balancing.

If you know that all loop iterations consume roughly the same amount of time, the OpenMP `schedule` clause should be used to distribute the iterations of the loop among the threads in roughly equal amounts via the scheduling policy. In addition, you need to minimize the chances of memory conflicts that may arise because of false sharing due to using large chunks. This behavior is possible because loops generally touch memory sequentially, so splitting up the loop in large chunks— like the first half and second half when using two threads— will result in the least chance for overlapping memory. While this may be the best choice for memory issues, it may be bad for load balancing. Unfortunately, the reverse is also true; what might be best for load balancing may be bad for memory performance. You must strike a balance between optimal memory usage and optimal load balancing by measuring the performance to see what method produces the best results.

Use the following general form on the `parallel` construct to schedule an OpenMP loop:

```
#pragma omp parallel for schedule(kind [, chunk size])
```

Four different loop scheduling types (kinds) can be provided to OpenMP, as shown in the following table. The optional parameter (chunk), when specified, must be a positive integer.

Kind	Description
static	Divide the loop into equal-sized chunks or as equal as possible in the case where the number of loop iterations is not evenly divisible by the number of threads multiplied by the chunk size. By default, chunk size is *loop_count*/*number_of_threads*. Set chunk to 1 to interleave the iterations.
dynamic	Use the internal work queue to give a chunk-sized block of loop iterations to each thread. When a thread is finished, it retrieves the next block of loop iterations from the top of the work queue. By default, the chunk size is 1. Be careful when using this scheduling type because of the extra overhead involved.
guided	Similar to dynamic scheduling, but the chunk size starts off large and decreases to better handle load imbalance between iterations. The optional chunk parameter specifies them minimum size chunk to use. By default the chunk size is approximately *loop_count*/*number_of_threads*.
auto	When schedule (auto) is specified, the decision regarding scheduling is delegated to the compiler. The programmer gives the compiler the freedom to choose any possible mapping of iterations to threads in the team.
runtime	Uses the `OMP_SCHEDULE` environment variable to specify which one of the three loop-scheduling types should be used. `OMP_SCHEDULE` is a string formatted exactly the same as would appear on the parallel construct.

Assume that you want to parallelize the following loop.

```
for (i=0; i < NumElements; i++) {
   array[i] = StartVal;
   StartVal++;
}
```

As written, the loop contains a data dependency, making it impossible to parallelize without a change. The new loop, shown below, fills the array in the same manner, but without data dependencies. The new loop benefits from using the SIMD instructions generated by the compiler.

```
#pragma omp parallel for
for (i=0; i < NumElements; i++)
{
     array[i] = StartVal + i;
}
```

Observe that the code is not 100% identical because the value of variable *StartVal* is not incremented. As a result, when the parallel loop is finished, the variable will have a value different from the one produced by the serial version. If the value of `StartVal` is needed after the loop, the additional statement, shown below, is needed.

```
// This works and is identical to the serial version.
#pragma omp parallel for
for (i=0; i < NumElements; i++)
{
     array[i] = StartVal + i;
}
StartVal += NumElements;
```

OpenMP Tasking Model

The OpenMP tasking model enables parallelization of a large range of applications. A task is an instance of executable code and its data environment that can be scheduled for execution by threads.

The `task` Construct

The `task` construct defines an explicit `task` region as shown in the following example:

```
void test1(LIST *head) {
  #pragma omp parallel shared(head)
  {
    #pragma omp single
     {
      LIST *p = head;
      while (p != NULL) {
        #pragma omp task firstprivate(p)
        {
          do_work1(p);
        }
        p = p->next;
     }
    }
  }
```

The binding thread set of the `task` region is the current parallel team. A `task` region binds to the innermost enclosing `parallel` region. When a thread encounters a `task` construct, a task is generated from the structured block enclosed in the construct. The encountering thread may immediately execute the task, or defer its execution. A `task` construct may be nested inside an outer task, but the `task` region of the inner task is not a part of the `task` region of the outer task.

Use Clauses with the `task` Construct

The `task` construct can take optional clauses. The data environment of the task is created according to the data-sharing attribute clauses on the `task` construct and any defaults that apply. The example below shows a way to generate `N` tasks with one thread and execute the generated tasks with the threads in the parallel team:

```
double data[N];
int i;
#pragma omp parallel shared(data)
{
  #pragma omp single private(i)
  {
    for (i=0, i<N; i++)
    {
      #pragma omp task firstprivate(i) shared(data))
      {
        do_work(data, i);
      }
    }
  }
}
```

Task Scheduling

When a thread reaches a task scheduling point, it may perform a task switch, suspending the current task and beginning or resuming execution of a different task bound to the current team. Refer to the OpenMP 5.1 specifications for the full list of task scheduling point locations. Some examples include:

- the point where a task is explicitly generated.

- the point immediately following the generation of an explicit task.
- after the last instruction of a `task` region.
- in a `taskwait` region.
- in a `taskyield` region.
- in implicit and explicit barrier regions.

NOTE
Task scheduling points dynamically divide `task` regions into parts. Each part is executed from start to finish without interruption. Different parts of the same `task` region are executed in the order in which they are encountered. In the absence of task synchronization constructs, the order in which a thread executes parts of different schedulable tasks is unspecified. A correct program must behave correctly and consistently with all conceivable scheduling sequences.

The `taskwait` Construct

The `taskwait` construct specifies a wait on the completion of child tasks generated since the beginning of the current task. A `taskwait` region binds to the current `task` region. The binding thread set of the `taskwait` region is the current team.

The `taskwait` region includes an implicit task scheduling point in the current `task` region. The current `task` region is suspended at the task scheduling point until execution of all its child tasks generated before the `taskwait` region is completed.

```
#pragma omp task // TASK1
{
  ...
  #pragma omp task // TASK 2 (child of TASK1)
  {
    do_work1();
  }
  #pragma omp task // TASK3 (child of TASK 1)
  {
    ...
    #pragma omp task // TASK4 (child of TASK3, not TASK1)
    {
      do_work2();
    }
    ...
  }
  #pragma omp taskwait // suspend TASK1; wait for TASK2 and TASK3 to complete
  ...
}
```

The `taskyield` Construct

The `taskyield` construct specifies that the current task can be suspended at that point and the thread may switch to the execution of a different task. You can use this construct to provide an explicit task scheduling point at a particular point in the task.

See Also

OMP_SCHEDULE
qopenmp, Qopenmp
Supported Environment Variables

Control Thread Allocation

The KMP_HW_SUBSET and KMP_AFFINITY environment variables allow you to control how the OpenMP* runtime uses the hardware threads on the processors. These environment variables allow you to try different thread distributions on the cores of the processors and determine how these threads are bound to the cores. You can use the environment variables to work out what is optimal for your application.

The KMP_HW_SUBSET variable controls the allocation of hardware resources and the KMP_AFFINITY variable controls how the OpenMP threads are bound to those resources.

Control Thread Distribution

The KMP_HW_SUBSET variable controls the hardware resources that will be used by the program. This variable often specifies three layers of machine topology: the number of sockets to use, how many cores to use per socket, and how many threads to use per core. For example, KMP_HW_SUBSET=2s,12c,2t means to use two sockets, 12 cores per socket, and two threads per core, giving a total of 48 available hardware threads.

When more layers exist (NUMA domain, tile, etc.) in the machine topology, you can specify those layers as well. For example, KMP_HW_SUBSET=2s,2n,8c,2t means to use two sockets, two NUMA domains per socket, eight cores per NUMA domain, and two threads per core, giving a total of 64 available hardware threads. For historical reasons, when a layer is not explicitly specified in KMP_HW_SUBSET, it is assumed you want all the resources in that unspecified layer. You can use KMP_AFFINITY=verbose to see all the different detected layers in the machine. For example, KMP_HW_SUBSET=2s,2t is interpreted to mean use two sockets, all cores per socket (and possibly all resources of other detected layers as well), and two threads per layer.

When available, you can specify core attributes to choose different sets of cores. The core attributes are appended to the regular core layer specification with a colon (:) and attribute. There are two attributes to help filter types of cores:

1. Core type, specified as intel_core, or intel_atom.
2. Core efficiency, specified as eff*num* where *num* is a non-negative integer from zero to the number of core efficiencies detected minus one. The larger the efficiency the more performant the core. For example, KMP_HW_SUBSET=4c:eff0,5c:eff1 will select all sockets, four cores of efficiency 0, five cores of efficiency 1, and all threads per those cores.

There is also a special syntax to explicitly request all resources at a specific layer. Instead of specifying a positive integer, you can use an optional asterisk (*). For example, KMP_HW_SUBSET='*c:eff0' or KMP_HW_SUBSET=c:eff0 will request all the cores of efficiency 0.

Consider a system with 24 cores and four hardware threads per core. While specifying two threads per core often yields better performance than one thread per core, specifying three or four threads per core may or may not improve the performance. This variable enables you to conveniently measure the performance of up to four threads per core.

For example, you can determine the effects of assigning 24, 48, 72, or the maximum 96 OpenMP threads in a system with 24 cores by specifying the following variable settings:

To Assign This Number of Threads ...	... Use This Setting
24	KMP_HW_SUBSET=24c,1t
48	KMP_HW_SUBSET=24c,2t
72	KMP_HW_SUBSET=24c,3t
96	KMP_HW_SUBSET=24c,4t

Caution
Take care when using the OMP_NUM_THREADS variable along with this variable. Using the OMP_NUM_THREADS variable can result in over or under subscription.

NOTE
If you use KMP_HW_SUBSET to specify more resources than the system has, the runtime will issue a warning and ignore the setting. For example, setting KMP_HW_SUBSET=24c,5t will be ignored on a system where each core has four hardware threads.

Control Thread Bindings

The KMP_AFFINITY variable controls how the OpenMP threads are bound to the hardware resources allocated by the KMP_HW_SUBSET variable. While this variable can be set to several binding or affinity types, the following are the recommended affinity types to use to run your OpenMP threads on the processor:

- *compact*: Distribute the threads sequentially among the cores.
- *scatter*: Distribute the threads among the cores in a round robin manner. Distribution is one thread per core initially, followed by repeat distribution among the cores.

The following table shows how the threads are bound to the cores when you want to use three threads per core on two cores by specifying KMP_HW_SUBSET=2c,3t:

Affinity	**OpenMP Threads on Core 0**	**OpenMP Threads on Core 1**
KMP_AFFINITY=compact	0, 1, 2	3, 4, 5
KMP_AFFINITY=scatter	0, 2, 4	1, 3, 5

Determine the Best Setting

To determine the best thread distribution and bindings using these variables, use the following:

1. Ensure that your OpenMP code is working properly before using these environment variables.
2. Establish a baseline with your current OpenMP code to compare to the performance when you allocate the threads to a processor.
3. Measure the performance of distributing one, two, three, or four threads per core by use the KMP_HW_SUBSET variable.
4. Measure the performance of binding the threads to the cores by using the KMP_AFFINITY variable.

See Also

Thread Affinity Interface
Supported Environment Variables

OpenMP* Pragmas

This is a summary of the OpenMP* pragmas supported in the Intel® oneAPI DPC++/C++ Compiler. For detailed information about the OpenMP API, see the *OpenMP Application Program Interface* Version 5.1 specification, which is available from the OpenMP web site.

PARALLEL Pragma

Use this pragma to form a team of threads and execute those threads in parallel.

Pragma	Description
omp parallel	Specifies that a structured block should be run in parallel by a team of threads.

TASKING Pragma

Use these pragmas for deferring execution.

Pragma	Description
omp task	Specifies a code block whose execution may be deferred.
omp taskloop	Specifies that the iterations of one or more associated for loops should be executed using OpenMP tasks. The iterations are distributed across tasks that are created by the construct and scheduled to be executed in parallel by the current team.

WORKSHARING Pragmas

Use these pragmas to share work among a team of threads.

Pragma	Description
omp for	Specifies a work-sharing loop. Iterations of the loop are executed in parallel by the threads in the team.
omp loop	Specifies that the iterations of the associated loops can execute in any order or concurrently.
omp sections	Defines a set of structured blocks that will be distributed among the threads in the team.
omp single	Specifies that a block of code is to be executed by only one thread in the team.

SYNCHRONIZATION Pragmas

Use these pragmas to synchronize between threads.

Pragma	Description
omp atomic	Specifies a computation that must be executed atomically.
omp barrier	Specifies a point in the code where each thread must wait until all threads in the team arrive.
omp critical	Specifies a code block that is restricted to access by only one thread at a time.
omp flush	Identifies a point at which a thread's temporary view of memory becomes consistent with the memory.
omp masked	Specifies a structured block that is executed by a subset of the threads of the current team.
omp master (deprecated, see omp masked)	Specifies a code block that must be executed only once by the primary thread of the team.

Pragma	Description
omp ordered	Specifies a block of code that the threads in a team must execute in the natural order of the loop iterations, or as a stand-alone directive, specifies cross-iteration dependences in a doacross loop-nest.
omp taskgroup	Causes the program to wait until the completion of all enclosed and descendant tasks.
omp taskwait	Specifies a wait on the completion of child tasks generated since the beginning of the current task.
omp taskyield	Specifies that the current task can be suspended at this point in favor of execution of a different task.

Data Environment Pragmas

Use these pragmas to affect the data environment.

Pragma	Description
omp threadprivate	Specifies a list of globally-visible variables that will be allocated private to each thread.

Offload Target Control Pragmas

Use these pragmas to control execution on one or more offload targets.

Pragma	Description
omp declare target	Specifies functions and variables that are created or mapped to a device.
omp declare variant	Identifies a variant of a base procedure and specifies the context in which this variant is used.
omp dispatch	Determines if a procedure variant is called for a given procedure.
omp distribute	Specifies that the iterations of one or more loops should be distributed among the initial threads of all thread teams in a league.
omp interop	Identifies a foreign runtime context and identifies runtime characteristics of that context, enabling interoperability with it.
omp requires	Lists the features that an implementation must support so that the program compiles and runs correctly.
omp target enter data	Specifies that variables are mapped to a device data environment.
omp target exit data	Specifies that variables are unmapped from a device data environment.
omp teams	Creates a league of thread teams inside a target region to execute a structured block in the initial thread of each team.

Vectorization Pragmas

Use these pragmas to control execution on vector hardware.

Pragma	Description
omp simd	Transforms the loop into a loop that will be executed concurrently using SIMD instructions.
omp declare simd	Creates a version of a function that can process multiple arguments using Single Instruction Multiple Data (SIMD) instructions from a single invocation from a SIMD loop.

Cancellation Constructs

Pragma	Description
omp cancel	Requests cancellation of the innermost enclosing region of the type specified, and causes the encountering task to proceed to the end of the cancelled construct.
omp cancellation point	Defines a point at which implicit or explicit tasks check to see if cancellation has been requested for the innermost enclosing region of the type specified. This construct does not implement a synchronization between threads or tasks.

User-Defined Reduction Pragma

Use this pragma to define reduction identifiers that can be used as reduction operators in a reduction clause.

Pragma	Description
omp declare reduction	Declares User-Defined Reduction (UDR) functions (reduction identifiers) that can be used as reduction operators in a reduction clause.

Memory Space Allocation Pragma

Use this declarative directive to allocate memory space.

Pragma	Description
omp allocate	Specifies memory allocators to use for object allocation and deallocation

Combined and Composite Pragmas

Use these pragmas as shortcuts for multiple pragmas in sequence. A combined construct is a shortcut for specifying one construct immediately nested inside another construct. A combined construct is semantically identical to that of explicitly specifying the first construct containing one instance of the second construct and no other statements.

A composite construct is composed of two constructs but does not have identical semantics to specifying one of the constructs immediately nested inside the other. A composite construct either adds semantics not included in the constructs from which it is composed or the nesting of the one construct inside the other is not conforming.

Pragma	Description
omp distribute parallel for [1]	Specifies a loop that can be executed in parallel by multiple threads that are members of multiple teams.

Pragma	Description
omp distribute parallel for simd[1]	Specifies a loop that will be executed in parallel by multiple threads that are members of multiple teams. It will be executed concurrently using SIMD instructions.
omp distribute simd [1]	Specifies a loop that will be distributed across the primary threads of the teams region. It will be executed concurrently using SIMD instructions.
omp for simd[1]	Specifies that the iterations of the loop will be distributed across threads in the team. Iterations executed by each thread can also be executed concurrently using SIMD instructions.
omp parallel for	Provides an abbreviated way to specify a parallel region containing only a FOR construct.
omp parallel for simd	Specifies a parallel construct that contains one for simd construct and no other statement.
omp parallel sections	Specifies a parallel construct that contains only a sections construct.
omp target parallel	Creates a device data environment and executes the parallel region on that device.
omp target parallel for	Provides an abbreviated way to specify a target construct that contains an omp target parallel for construct and no other statement between them.
omp target parallel for simd	Specifies a target construct that contains an omp target parallel for simd construct and no other statement between them.
omp target parallel loop	Provides an abbreviated way to specify a target region that contains only a parallel loop construct.
omp target simd	Specifies a target construct that contains an omp simd construct and no other statement between them.
omp target teams	Creates a device data environment and executes the construct on the same device. It also creates a league of thread teams with the primary thread in each team executing the structured block.
omp target teams distribute	Creates a device data environment and then executes the construct on that device. It also specifies that loop iterations will be distributed among the primary threads of all thread teams in a league created by a teams construct.
omp target teams distribute parallel for	Creates a device data environment and then executes the construct on that device. It also specifies a loop that can be executed in parallel by multiple threads that are members of multiple teams created by a teams construct.
omp target teams distribute parallel for simd	Creates a device data environment and then executes the construct on that device. It also specifies a loop that can be executed in parallel by multiple threads that are members of multiple teams created by a teams construct. The loop will be distributed across the teams, which will be executed concurrently using SIMD instructions.

Pragma	Description
omp target teams distribute simd	Creates a device data environment and then executes the construct on that device. It also specifies that loop iterations will be distributed among the primary threads of all thread teams in a league created by a teams construct. It will be executed concurrently using SIMD instructions.
omp target teams loop	Provides an abbreviated way to specify a target region that contains only a teams loop construct.
omp taskloop simd [1]	Specifies a loop that can be executed concurrently using SIMD instructions and that those iterations will also be executed in parallel using OpenMP* tasks.
omp teams distribute	Creates a league of thread teams and specifies that loop iterations will be distributed among the primary threads of all thread teams in the league.
omp teams distribute parallel for	Creates a league of thread teams and specifies that the associated loop can be executed in parallel by multiple threads that are members of multiple teams.
omp teams distribute parallel for simd	Creates a league of thread teams and specifies that the associated loop can be executed concurrently using SIMD instructions in parallel by multiple threads that are members of multiple teams.
omp teams distribute simd	Creates a league of thread teams and specifies that the associated loop will be distributed across the primary threads of the teams and executed concurrently using SIMD instructions.
omp teams loop	Provides an abbreviated way to specify a teams construct that contains only a loop construct.

Footnotes:

[1] This directive specifies a composite construct.

OpenMP* Library Support

This section provides information about OpenMP* run-time library routines, Intel® compiler extension routines to OpenMP, OpenMP support libraries and how to use them, and the thread affinity interface.

OpenMP* Run-time Library Routines

OpenMP* provides run-time library routines to help you manage your program in parallel mode. Many of these run-time library routines have corresponding environment variables that can be set as defaults. The run-time library routines let you dynamically change these factors to assist in controlling your program. In all cases, a call to a run-time library routine overrides any corresponding environment variable.

Caution
Running OpenMP runtime library routines may initialize the OpenMP runtime environment, which might cause a situation where subsequent programmatic setting of OpenMP environment variables has no effect. To avoid this situation, you can use the Intel extension routine `kmp_set_defaults()` to set OpenMP environment variables.

The compiler supports all the OpenMP run-time library routines. Refer to the OpenMP API specification for detailed information about using these routines.

Include the appropriate declarations of the routines in your source code by adding a statement similar to the following:

```
#include <omp.h>
```

The header files are provided in the `../include` (Linux*) or `..\include` (Windows*) directory of your compiler installation.

Thread Team Routines

Routines that affect and monitor thread teams in the current contention group.

Routine	Description
`void omp_set_num_threads(int nthreads)`	Sets the number of threads to use for subsequent parallel regions created by the calling thread.
`int omp_get_num_threads(void)`	Returns the number of threads that are being used in the current parallel region. This function does not necessarily return the value inherited by the calling thread from the `omp_set_num_threads()` function.
`int omp_get_max_threads(void)`	Returns the number of threads available to subsequent parallel regions created by the calling thread.
`int omp_get_thread_num(void)`	Returns the thread number of the calling thread, within the context of the current parallel region.
`int omp_in_parallel(void)`	Returns `TRUE` if called within the dynamic extent of a parallel region executing in parallel; otherwise returns `FALSE`.
`void omp_set_dynamic(int dynamic_threads)`	Enables or disables dynamic adjustment of the number of threads used to execute a parallel region. If `dynamic_threads` is `TRUE`, dynamic threads are enabled. If `dynamic_threads` is `FALSE`, dynamic threads are disabled. Dynamic threads are disabled by default.
`int omp_get_dynamic(void)`	Returns `TRUE` if dynamic thread adjustment is enabled, otherwise returns `FALSE`.
`int omp_get_cancellation(void)`	Returns `TRUE` if cancellation is enabled, otherwise returns `FALSE`. This routine can be affected by the setting for environment variable `OMP_CANCELLATION`.

Routine	Description
`void omp_set_nested(int nested)` **NOTE** This has been deprecated.	Enables or disables nested parallelism. If `nested` is `TRUE`, nested parallelism is enabled. If `nested` is `FALSE`, nested parallelism is disabled. Nested parallelism is disabled by default.
`int omp_get_nested(void)` **NOTE** This has been deprecated.	Returns `TRUE` if nested parallelism is enabled, otherwise returns `FALSE`.
`void omp_set_schedule(omp_sched_t kind,int chunk_size)`	Determines the schedule of a worksharing loop that is applied when '`runtime`' is used as the schedule kind.
`void omp_get_schedule(omp_sched_kind *kind,int *chunk_size)`	Returns the schedule of a worksharing loop that is applied when the '`runtime`' schedule is used.
`int omp_get_thread_limit(void)`	Returns the maximum number of simultaneously executing threads in an OpenMP program.
`int omp_get_supported_active_levels(void)`	Returns the number of active levels of parallelism supported by the implementation.
`void omp_set_max_active_levels(int max_active_levels)`	Limits the number of nested active parallel regions. The value of `max_active_levels` must evaluate to a non-negative integer.
`int omp_get_max_active_levels(void)`	Returns the maximum number of nested active parallel regions.
`int omp_get_level(void)`	Returns the number of nested parallel regions (whether active or inactive) enclosing the task that contains the call, not including the implicit parallel region.
`int omp_get_ancestor_thread_num(int level)`	Returns the thread number of the ancestor at a given nest level of the current thread.
`int omp_get_team_size(int level)`	Returns the size of the thread team to which the ancestor or the current thread belongs for a given nested level.
`int omp_get_active_level(void)`	Returns the number of nested, active parallel regions enclosing the task that contains the call.

Thread Affinity Routines

Routines that affect and access thread affinity policies that are in effect.

Function	Description
omp_proc_bind_t omp_get_proc_bind(void)	Returns the currently active thread affinity policy, which can be initialized by the environment variable OMP_PROC_BIND. This policy is used for subsequent nested parallel regions.
int omp_get_num_places(void)	Returns the number of places available to the execution environment in the place list of the initial task, usually threads, cores, or sockets.
int omp_get_place_num_procs(int place_num)	Returns the number of processors associated with the place numbered place_num. The routine returns zero when place_num is negative or is greater than or equal to omp_get_num_places().
void omp_get_place_proc_ids(int place_num, int *ids)	Returns the numerical identifiers of each processor associated with the place numbered place_num. The numerical identifiers are non-negative and their meaning is implementation defined. The numerical identifiers are returned in the array ids and their order in the array is implementation defined.The array ids must be sufficiently large to contain omp_get_place_num_procs(place_num) elements. The routine has no effect when place_num is negative or greater than or equal to omp_get_num_places().
int omp_get_place_num(void)	Returns the place number of the place to which the encountering thread is bound. The returned value is between 0 and omp_get_num_places() - 1, inclusive. When the encountering thread is not bound to a place, the routine returns -1.
int omp_get_partition_num_places(void)	Returns the number of places in the place partition of the innermost implicit task.
void omp_get_partition_place_nums(int *place_nums)	Returns the list of place numbers corresponding to the places in the place-partition-var ICV of the innermost implicit task. The array place_nums must be sufficiently large to contain omp_get_partition_num_places() elements.
void omp_set_affinity_format(const char *format)	Sets the affinity format to be used on the device by setting the value of the affinity-format-var ICV.
size_t omp_get_affinity_format(char *buffer, size_t size)	Returns the value of the affinity-format-var ICV on the device.
void omp_display_affinity(const char *format)	Prints the OpenMP thread affinity information using the format specification provided.
size_t omp_capture_affinity(char *buffer, size_t size, const char *format)	Prints the OpenMP thread affinity information into a buffer using the format specification provided.

Teams Region Routines

Routines that affect and monitor the league of teams that may execute a teams region.

Function	Description
`int omp_get_num_teams(void)`	Returns the number of initial teams in the current teams region.
`int omp_get_team_num(void)`	Returns the initial team number of the calling thread.
`void omp_set_num_teams(int num_teams)`	Affects the number of threads to be used for subsequent teams regions that do not specify a `num_teams` clause.
`int omp_get_max_teams(void)`	Returns an upper bound on the number of teams that could be created by a teams construct without a `num_teams` clause that is encountered after execution returns from this routine.
`void omp_set_teams_thread_limit(int thread_limit)`	Defines the maximum number of OpenMP threads that can participate in each contention group created by a teams construct.
`int omp_get_teams_thread_limit(void)`	Returns the maximum number of OpenMP threads available to participate in each contention group created by a teams construct.

Tasking Routines

Routines that pertain to OpenMP explicit tasks.

Function	Description
`int omp_get_max_task_priority(void)`	Returns the maximum value that can be specified in the `priority` clause.
`int omp_in_explicit_task(void)`	Returns `TRUE` if called within an explicit task region; otherwise returns `FALSE`.
`int omp_in_final(void)`	Returns `TRUE` if called within a final task region; otherwise returns `FALSE`.

Resource Relinquishing Routines

Routines that relinquish resources used by the OpenMP runtime. These routines are only effective on the host device.

Function	Description
`int omp_pause_resource(omp_pause_resource_t kind, int device_num)`	Allows the runtime to relinquish resources used by OpenMP on the specified device. The routine returns zero in case of success, and non-zero otherwise.
`int omp_pause_resource_all(omp_pause_resource_t kind)`	Allows the runtime to relinquish resources used by OpenMP on all devices. The routine returns zero in case of success, and non-zero otherwise.

Device Information Routines

Routines that pertain to the set of devices that are accessible to an OpenMP program.

Function	Description
`int omp_get_num_procs(void)`	Returns the number of processors available to the program.
`void omp_set_default_device(int device_number)`	Sets the default device number.
`int omp_get_default_device(void)`	Returns the default device number.
`int omp_get_num_devices(void)`	Returns the number of target devices.
`int omp_get_device_num(void)`	Returns the device number of the device on which the calling thread is executing.
`int omp_is_initial_device(void)`	Returns `TRUE` if the current task is running on the host device; otherwise, `FALSE`.
`int omp_get_initial_device(void)`	Returns the device number of the host device. The value of the device number is implementation defined. If it is between 0 and `omp_get_num_devices()`-1, then it is valid in all device constructs and routines; if it is outside that range, then it is only valid in the device memory routines and not in the `device` clause.

Device Memory Routines

Routines that support allocation of memory and management of pointers in the data environments of target devices.

Routine	Description
`void *omp_target_alloc(size_t size, int device_num)`	Allocates memory in a device data environment and returns a device pointer to that memory.
`void omp_target_free(void *device_ptr, int device_num)`	Frees device memory that was allocated by the `omp_target_alloc`.
`int omp_target_is_present(const void *ptr, int device_num)`	Returns TRUE if `device_num` refers to the host device or if `ptr` refers to storage that has corresponding storage in the device data environment of `device_num`. Otherwise, it returns FALSE.
`int omp_target_is_accessible(const void *ptr, size_t size, int device_num)`	Returns TRUE if the storage of size bytes starting at the address given by `ptr` is accessible from device `device_num`. Otherwise, it returns FALSE.
`int omp_target_memcpy(void *dst, const void *src, size_t length, size_t dst_offset, size_t src_offset, int dst_device_num, int src_device_num)`	This routine copies `length` bytes of memory at offset `src _offset` from `src` in the device data environment of device `src_device_num` to `dst`, starting at offset `dst_offset` in the device data environment of the device specified by

Routine	Description
	dst_device_num. Returns zero on success and a non-zero value on failure. Use omp_get_initial_device to return the device number you can use to reference the host device and host device data environment. This routine includes a task scheduling point. The effect of this routine is unspecified when it is called from within a target region.
int omp_target_memcpy_rect(void *dst, const void *src, size_t element_size, int num_dims, const size_t *volume, const size_t *dst_offsets, const size_t *src_offsets, const size_t *dst_dimensions, const size_t *src_dimensions, int dst_device_num, int src_device_num)	This routine copies a rectangular subvolume of src, in the device data environment of the device specified by src_device_num, to dst, in the device data environment of the device specified by dst_device_num. Specify the volume in terms of the size of an element, the number of its dimensions, and constant arrays of length num_dims. The maximum number of dimensions supported is three or more. The volume array specifies the length, in number of elements, to copy in each dimension from src to dst. The dst_offsets and src_offsets parameters specify the number of elements from the origin of dst and src, in elements. The dst_dimensions and src_dimensions parameters specify the length of each dimension of dst and src. The routine returns zero if successful. Otherwise, it returns a non-zero value. If both dst and src are NULL pointers, the routine returns the number of dimensions supported by the implementation for the specified device numbers. You can use the device number returned by omp_get_initial_device to reference the host device and host device data environment. This routine contains a task scheduling point. The effect of this routine is unspecified when called from within a target region.
int omp_target_associate_ptr(const void *host_ptr, const void *device_ptr, size_t size, size_t device_offset, int device_num)	Maps a device pointer, which might be returned by omp_target_alloc, to a host pointer.
int omp_target_disassociate_ptr(const void *ptr, int device_num)	Removes the associated pointer for a given device from a host pointer.
void *omp_get_mapped_ptr(const void *ptr, int device_num)	Returns the device pointer that is associated with a host pointer for a given device.

Lock Routines

Use these routines to affect OpenMP locks.

Function	Description
`void omp_init_lock(omp_lock_t *lock)`	Initializes the lock to the unlocked state.
`void omp_init_nest_lock(omp_nest_lock_t *lock)`	Initializes the nested lock to the unlocked state. The nesting count for the nested lock is set to zero.
`void omp_init_lock_with_hint(omp_lock_t *lock, omp_sync_hint_t hint)`	Initializes the lock to the unlocked state, optionally choosing a specific lock implementation based on `hint`. See the OpenMP specification for the available hints.
`void omp_init_nest_lock_with_hint(omp_nest_lock_t *lock, omp_sync_hint_t hint)`	Initializes the nested lock to the unlocked state, optionally choosing a specific lock implementation based on `hint`. The nesting count for the nested lock is set to zero. See the OpenMP specification for the available hints.
`void omp_destroy_lock(omp_lock_t *lock)`	Changes the state of the lock to uninitialized.
`void omp_destroy_nest_lock(omp_nest_lock_t *lock)`	Changes the state of the nested lock to uninitialized.
`void omp_set_lock(omp_lock_t *lock)`	Forces the executing thread to wait until the lock is available. The thread is granted ownership of the lock when it becomes available.
`void omp_set_nest_lock(omp_nest_lock_t *lock)`	Forces the executing thread to wait until the nested lock is available. If the thread already owns the lock, then the lock nesting count is incremented.
`void omp_unset_lock(omp_lock_t *lock)`	Releases the executing thread from ownership of the lock. The behavior is undefined if the executing thread does not own the lock.
`void omp_unset_nest_lock(omp_nest_lock_t *lock)`	Decrements the nesting count for the nested lock and releases the executing thread from ownership of the nested lock if the resulting nesting count is zero. Behavior is undefined if the executing thread does not own the nested lock.
`int omp_test_lock(omp_lock_t *lock)`	Attempts to set the lock. If successful, returns `TRUE`, otherwise returns `FALSE`.
`int omp_test_nest_lock(omp_nest_lock_t *lock)`	Attempts to set the nested lock. If successful, returns the nesting count, otherwise returns zero.

Timing Routines

Function	Description
`double omp_get_wtime(void)`	Returns a double precision value equal to the elapsed wall clock time (in seconds) relative to an arbitrary reference time. The reference time does not change during program execution.

Function	Description
`double omp_get_wtick(void)`	Returns a double precision value equal to the number of seconds between successive clock ticks.

Event Routines

Function	Description
`void omp_fulfill_event(omp_event_handle_t event)`	Fulfills the event associated with the event handle `event` and destroys the event.

Interoperability Routines

Function	Description
`int omp_get_num_interop_properties(const omp_interop_t interop)`	Returns the number of implementation-defined properties available for interop. The total number of properties available for interop is the returned value minus `omp_ipr_first`.
`omp_intptr_t omp_get_interop_int(const omp_interop_t interop, omp_interop_property_t property_id, int *ret_code)`	Returns the requested integer property, if available, and zero if an error occurs or no value is available.
`void *omp_get_interop_ptr(const omp_interop_t interop, omp_interop_property_t property_id, int *ret_code)`	Returns the requested pointer property, if available, and NULL if an error occurs or no value is available.
`const char *omp_get_interop_str(const omp_interop_t interop, omp_interop_property_t property_id, int *ret_code)`	Returns the requested string property as a C string, if available, and NULL if an error occurs or no value is available.
`const char *omp_get_interop_name(const omp_interop_t interop, omp_interop_property_t property_id)`	Returns the name of the property identified by `property_id` as a C string.
`const char *omp_get_interop_type_desc(const omp_interop_t interop, omp_interop_property_t property_id)`	Returns a C string that describes the type of the property identified by `property_id` in human-readable form.
`const char *omp_get_interop_rc_desc(const omp_interop_t interop, omp_interop_rc_t ret_code)`	Returns a C string that describes the return code `ret_code` in human-readable form.

Memory Management Routines

Function	Description
`omp_allocator_handle_t omp_init_allocator(omp_memspace_handle_t memspace, int ntraits, const omp_alloctrait_t traits[])`	Creates a new allocator that is associated with the `memspace` memory space and returns a handle to it.
`void omp_destroy_allocator(omp_allocator_handle_t allocator)`	Releases all resources used to implement the allocator handle.
`void omp_set_default_allocator(omp_allocator_handle_t allocator)`	Sets the default memory allocator to be used by allocation calls, allocate directives and allocate clauses that do not specify an allocator.
`omp_allocator_handle_t omp_get_default_allocator(void)`	Returns a handle to the memory allocator to be used by allocation calls, allocate directives and allocate clauses that do not specify an allocator.
`void *omp_alloc(size_t size, omp_allocator_handle_t allocator)`	Requests a memory allocation of size bytes from the specified memory allocator.
`void *omp_aligned_alloc(size_t alignment, size_t size, omp_allocator_handle_t allocator)`	Requests a memory allocation of `size` bytes from the specified memory allocator. Memory allocated by `omp_aligned_alloc` will be byte-aligned to at least the maximum of the alignment required by malloc, the alignment trait of the allocator and the alignment argument value.
`void omp_free(void *ptr, omp_allocator_handle_t allocator)`	Deallocates the memory to which `ptr` points. The `ptr` argument must have been returned by an OpenMP allocation routine.
`void *omp_calloc(size_t nmemb, size_t size, omp_allocator_handle_t allocator)`	Requests a memory allocation from the specified memory allocator for an array of `nmemb` elements each of which has a size of `size` bytes.
`void *omp_aligned_calloc(size_t alignment, size_t nmemb, size_t size, omp_allocator_handle_t allocator)`	Requests a memory allocation from the specified memory allocator for an array of `nmemb` elements each of which has a size of `size` bytes. Memory allocated by `omp_aligned_calloc` will be byte-aligned to at least the maximum of the alignment required by malloc, the alignment trait of the allocator and the alignment argument value.
`void *omp_realloc(void *ptr, size_t size, omp_allocator_handle_t allocator, omp_allocator_handle_t free_allocator)`	Deallocates the memory to which `ptr` points and requests a new memory allocation of `size` bytes from the specified memory allocator. Upon success it returns a pointer to the allocated memory and the contents of the new object shall be the same as that of the old object prior to deallocation up to the minimum size of old allocated size and `size` argument.

Tool Control Routines

Function	Description
`int omp_control_tool(int command, int modifier, void *arg)`	Enables a program to pass commands to an active tool.

Environment Display Routines

Function	Description
`void omp_display_env(int verbose)`	Displays the OpenMP version number and the initial values of ICVs associated with the environment variables.

See Also

Intel Extension Routines to OpenMP*

Intel® Compiler Extension Routines to OpenMP*

The Intel® compiler implements the following group of routines as extensions to the OpenMP* run-time library:

- Get and set the execution environment
- Get and set the stack size for parallel threads
- Memory allocation
- Get and set the thread sleep time for the throughput execution mode
- Target memory allocation

The Intel® extension routines described in this section can be used for low-level tuning to verify that the library code and application are functioning as intended. These routines are generally not recognized by other OpenMP-compliant compilers, which may cause the link stage to fail in the other compiler. To execute these OpenMP routines, use the `/Qopenmp-stubs` (Windows*) or `-qopenmp-stubs` (Linux*) option.

In most cases, environment variables can be used in place of the extension library routines. For example, the stack size of the parallel threads may be set using the `OMP_STACKSIZE` environment variable rather than the `kmp_set_stacksize_s()` library routine.

NOTE
A run-time call to an Intel extension routine takes precedence over the corresponding environment variable setting.

Execution Environment

Function	Description
`void kmp_set_defaults(char const *)`	Sets OpenMP environment variables defined as a list of variables separated by "|" in the argument.
`void kmp_set_library_throughput(void)`	Sets execution mode to throughput, which is the default. Allows the application to determine the runtime environment. Use in multi-user environments.
`void kmp_set_library_turnaround(void)`	Sets execution mode to turnaround. Use in dedicated parallel (single user) environments.

Function	Description
`void kmp_set_library_serial(void)`	Sets execution mode to serial.
`void kmp_set_library(int)`	Sets execution mode indicated by the value passed to the function. Valid values are: • **1** - serial mode • **2** - turnaround mode • **3** - throughput mode Call this routine before the first parallel region is executed.
`int kmp_get_library(void)`	Returns a value corresponding to the current execution mode: • **1** - serial • **2** - turnaround • **3** - throughput

Stack Size

Function	Description
`size_t kmp_get_stacksize_s(void)`	Returns the number of bytes that will be allocated for each parallel thread to use as its private stack. This value can be changed with `kmp_set_stacksize_s()` routine, prior to the first parallel region or via the `KMP_STACKSIZE` environment variable.
`int kmp_get_stacksize(void)`	Provided for backwards compatibility only. Use `kmp_get_stacksize_s()` routine for compatibility across different families of Intel processors.
`void kmp_set_stacksize_s(size_t`*`size`*`)`	Sets to *size* the number of bytes that will be allocated for each parallel thread to use as its private stack. This value can also be set via the `KMP_STACKSIZE` environment variable. In order for `kmp_set_stacksize_s()` to have an effect, it must be called before the beginning of the first (dynamically executed) parallel region in the program.
`void kmp_set_stacksize(int `*`size`*`)`	Provided for backward compatibility only. Use `kmp_set_stacksize_s()` for compatibility across different families of Intel® processors.

Memory Allocation

The Intel® compiler implements a group of memory allocation routines as an extension to the OpenMP run-time library to enable threads to allocate memory from a heap local to each thread. These routines are: `kmp_malloc()`, `kmp_calloc()`, and `kmp_realloc()`.

The memory allocated by these routines must also be freed by the `kmp_free()` routine. While you can allocate memory in one thread and then free that memory in a different thread, this mode of operation incurs a slight performance penalty.

Function	Description
`void* kmp_malloc(size_t size)`	Allocate memory block of *size* bytes from thread-local heap.
`void* kmp_calloc(size_t nelem, size_t elsize)`	Allocate array of *nelem* elements of size *elsize* from thread-local heap.
`void* kmp_realloc(void* ptr, size_t size)`	Reallocate memory block at address *ptr* and *size* bytes from thread-local heap.
`void* kmp_free(void* ptr)`	Free memory block at address *ptr* from thread-local heap. Memory must have been previously allocated with `kmp_malloc()`, `kmp_calloc()`, or `kmp_realloc()`.

Thread Sleep Time

In the throughput OpenMP* Support Libraries, threads wait for new parallel work at the ends of parallel regions, and then sleep, after a specified period of time. This time interval can be set by the `KMP_BLOCKTIME` environment variable or by the `kmp_set_blocktime()` function.

Function	Description
`int kmp_get_blocktime(void)`	Returns the number of milliseconds that a thread should wait, after completing the execution of a parallel region, before sleeping, as set either by the `KMP_BLOCKTIME` environment variable or by `kmp_set_blocktime()`.
`void kmp_set_blocktime(int msec)`	Sets the number of milliseconds that a thread should wait, after completing the execution of a parallel region, before sleeping. This routine affects the block time setting for the calling thread and any OpenMP team threads formed by the calling thread. The routine does not affect the block time for any other threads.

Target Memory Allocation

Function	Description
`void *omp_target_alloc_host(size_t size, int device_num)`	Returns the address of a storage location that is `size` bytes in length allocated in host memory. The same pointer may be used to access the memory on the host and all supported devices. If the allocation request fails, a null pointer is returned.
`void *omp_target_alloc_device(size_t size, int device_num)`	Returns the address of a storage allocation that is `size` bytes in length. Device allocations are owned by the device specified by `device_num` in device

	memory if present. Generally, the allocation can be accessed only by the device, but may be copied to other device or host allocated memory. A null pointer return value indicates the allocation was not successful.
`void *omp_target_alloc_shared(size_t size, int device_num)`	Returns the address of a storage allocation that is `size` bytes in length. The same pointer may be used to access the memory on the host and the specified device. Shared allocations are shared by the host and the specified device, and are intended to migrate between the host and the device. A null pointer is returned if the allocation is unsuccessful.
`void *ompx_target_realloc(void *ptr, size_t size, int device_num)`	Deallocates the device memory specified with `ptr` and allocates a new device memory with the specified size in bytes for the given device `device_num`. The returned memory can be accessed only by the specified device. The contents of the new memory object are the same as that of the old object prior to deallocation up to the minimum size of old allocated size and `size` argument.
`void *ompx_target_realloc_host(void *ptr, size_t size, int device_num)`	Deallocates the device memory specified with `ptr` and allocates a new device memory with the specified size in bytes for the given device `device_num`. The returned memory can be accessed by the host and all supported devices. The contents of the new memory object are the same as that of the old object prior to deallocation up to the minimum size of old allocated size and `size` argument.
`void *ompx_target_realloc_device(void *ptr, size_t size, int device_num)`	Deallocates the device memory specified with `ptr` and allocates a new device memory with the specified size in bytes for the given device `device_num`. The returned memory can be accessed only by the specified device. The contents of the new memory object are the same as that of the old object prior to deallocation up to the minimum size of old allocated size and `size` argument.
`void *ompx_target_realloc_shared(void *ptr, size_t size, int device_num)`	Deallocates the device memory specified with `ptr` and allocates a new device memory with the specified size in bytes for the given device `device_num`. The returned memory can be accessed by the host and the specified device. The contents of the new memory object are the same as that of the old object prior to deallocation up to the minimum size of old allocated size and `size` argument.

`void *ompx_target_aligned_alloc(size_t alignment, size_t size, int device_num)`	Allocates device memory that is aligned to the specified alignment argument `align` for the specified device `device_num`. The returned memory can be accessed only by the specified device.
`void *ompx_target_aligned_alloc_host(size_t alignment, size_t size, int device_num)`	Allocates device memory that is aligned to the specified alignment argument `align` for the specified device `device_num`. The returned memory can be accessed by the host and all supported devices.
`void *ompx_target_aligned_alloc_device(size_t alignment, size_t size, int device_num)`	Allocates device memory that is aligned to the specified alignment argument `align` for the specified device `device_num`. The returned memory can be accessed only by the specified device.
`void *ompx_target_aligned_alloc_shared(size_t alignment, size_t size, int device_num)`	Allocates device memory that is aligned to the specified alignment argument `align` for the specified device `device_num`. The returned memory can be accessed by the host and the specified device.

See Also

openmp-stubs, Qopenmp-stubs compiler option
OpenMP* Run-time Library Routines
OpenMP* Support Libraries

OpenMP* Support Libraries

The Intel® oneAPI DPC++/C++ Compiler provides support libraries for OpenMP*. There are several kinds of libraries:

- **Performance:** supports parallel OpenMP execution.
- **Stubs:** supports serial execution of OpenMP applications.

Each kind of library is available for both dynamic and static linking on Linux* operating systems. Only dynamic linking is supported on Windows* operating systems.

Performance Libraries

To use these libraries, specify the `/Qopenmp` (Windows*) or `-qopenmp` (Linux*) option.

Options that use OpenMP are available for both Intel® and non-Intel microprocessors, but these options may perform additional optimizations on Intel® microprocessors than they perform on non-Intel microprocessors. The list of major, user-visible OpenMP constructs and features that may perform differently on Intel® microprocessors than on non-Intel microprocessors includes: locks (internal and user visible), the SINGLE construct, barriers (explicit and implicit), parallel loop scheduling, reductions, memory allocation, and thread affinity and binding.

Operating System	Dynamic Link	Static Link
Linux	`libiomp5.so`	`libiomp5.a`
Windows	`libiomp5md.lib`	None

Operating System	Dynamic Link	Static Link
	`libiomp5md.dll`	

Many routines in the OpenMP support libraries are more optimized for Intel® microprocessors than for non-Intel microprocessors.

Stubs Libraries

To use these libraries, specify `/Qopenmp-stubs` (Windows*) or `-qopenmp-stubs` (Linux*) option. These allow you to compile OpenMP applications in serial mode and provide stubs for OpenMP routines and extended Intel-specific routines.

Operating System	Dynamic Link	Static Link
Linux	`libiompstubs5.so`	`libiompstubs5.a`
Windows	`libiompstubs5md.lib` `libiompstubs5md.dll`	None

Execution Modes

The compiler enables you to run an application under different execution modes specified at run time; the libraries support the turnaround, throughput, and serial modes. Use the `KMP_LIBRARY` environment variable to select the modes at run time.

Mode	Description
throughput (default)	The throughput mode allows the program to yield to other running programs and adjust resource usage to produce efficient execution in a dynamic environment. In a multi-user environment where the load on the parallel machine is not constant or where the job stream is not predictable, it may be better to design and tune for throughput. This minimizes the total time to run multiple jobs simultaneously. In this mode, the worker threads yield to other threads while waiting for more parallel work. After completing the execution of a parallel region, threads wait for new parallel work to become available. After a certain period of time has elapsed, they stop waiting and sleep. Until more parallel work becomes available, sleeping allows processor and resources to be used for other work by non-OpenMP threaded code that may execute between parallel regions, or by other applications. The amount of time to wait before sleeping is set either by the `KMP_BLOCKTIME` environment variable or by the `kmp_set_blocktime()` function. A small blocktime value may offer better overall performance if your application contains non-OpenMP threaded code that executes between parallel regions. A larger blocktime value may be more appropriate if threads are to be reserved solely for use for OpenMP execution, but may penalize other concurrently-running OpenMP or threaded applications.
turnaround	The turnaround mode is designed to keep active all processors involved in the parallel computation, which minimizes execution time of a single job. In this mode, the worker threads actively wait for more parallel work, without yielding to other threads (although they are still subject to `KMP_BLOCKTIME` control). In

Mode	Description
	a dedicated (batch or single user) parallel environment where all processors are exclusively allocated to the program for its entire run, it is most important to effectively use all processors all of the time. **NOTE** Avoid over-allocating system resources. The condition can occur if either too many threads have been specified, or if too few processors are available at run time. If system resources are over-allocated, this mode will cause poor performance. The throughput mode should be used instead if this occurs.
serial	The serial mode forces parallel applications to run as a single thread.

See Also

qopenmp, Qopenmp compiler option
qopenmp-stubs, Qopenmp-stubs compiler option

Use the OpenMP Libraries

This section describes the steps needed to set up and use the OpenMP Libraries from the command line. On Windows systems, you can also build applications compiled with the OpenMP libraries in the Microsoft Visual Studio development environment.

For a list of the options and libraries used by the OpenMP libraries, see OpenMP Support Libraries.

Set Up Environment

Set up your environment for access to the compiler to ensure that the appropriate OpenMP library is available during linking.

Linux

On Linux systems you can source the appropriate script file (`setvars` file).

Windows

On Windows systems you can either execute the appropriate batch (`.bat`) file or use the command-line window supplied in the compiler program folder that already has the environment set up.

During compilation, ensure that the version of `omp.h` used when compiling is the version provided by that compiler. For example, use the `omp.h` provided with GCC when you compile with GCC.

Caution
Be aware that when using the GCC or Microsoft Compiler, you may inadvertently use inappropriate header or module files. To avoid this, copy the header or module file(s) to a separate directory and put it in the appropriate `include` path using the `-I` option.

If a program uses data structures or classes that contain members with data types defined in the `omp.h` file, then source files that use those data structures should all be compiled with the same `omp.h` file.

Linux Examples

This section shows several examples of using OpenMP with the Intel® oneAPI DPC++/C++ Compiler from the command line on Linux.

Compile and Link OpenMP Libraries

You can compile an application and link the Intel OpenMP libraries with a single command using the `-qopenmp` option. For example:

```
icpx -qopenmp hello.cpp
```

By default, the Intel® oneAPI DPC++/C++ Compiler performs a dynamic link of the OpenMP libraries. To perform a static link (not recommended), add the option `-qopenmp-link=static`. The option `-qopenmp-link` controls whether the linker uses static or dynamic OpenMP libraries on Linux systems (default is `-qopenmp-link=dynamic`). See OpenMP Support Libraries for more information about dynamic and static OpenMP libraries.

Link OpenMP Object Files Compiled with GCC or Intel® oneAPI DPC++/C++ Compiler

You can use the `icx`/`icpx` compilers with the `gcc`/`g++` compilers to compile parts of an application and create object files that can then be linked (object-level interoperability).

When using `gcc` or the `g++` compiler to link the application with the Intel oneAPI DPC++/C++ Compiler OpenMP compatibility library, you need to specify the following:

- The Intel OpenMP library name using the `-l` option
- The Linux `pthread` library using the `-l` option
- The path to the Intel libraries where the Intel oneAPI DPC++/C++ Compiler is installed using the `-L` option

For example:

1. Compile `foo.c` and `bar.c` with `gcc`, using the `-fopenmp` option to enable OpenMP support:

```
gcc -fopenmp -c foo.c bar.c
```

   The `-c` prevents linking at this step.
2. Use the `gcc` compiler to link the application object code with the Intel OpenMP library:

```
gcc foo.o bar.o -liomp5 -lpthread -L<install_dir>/lib
```

   where <install_dir> is the location of the installed Intel OpenMP library.

Alternately, you can use the Intel oneAPI DPC++/C++ Compiler to link the application so that you don't need to specify the `gcc-l` option, `-L` option, and the `-lpthread` options.

For example:

1. Compile `foo.c` with `gcc`, using the `gcc -fopenmp` option to enable OpenMP:

```
gcc -fopenmp -c foo.c
```

2. Compile `bar.c` with icx, using the `-qopenmp` option to enable OpenMP:

```
icx -qopenmp -c bar.c
```

3. Use the icx compiler to link the resulting application object code with the Intel OpenMP library:

```
icx -qopenmp foo.o bar.o
```

Link Mixed C/C++ and Fortran Object Files

You can mix C/C++ and Fortran object files and link the Intel OpenMP libraries using GNU, GCC, or Intel oneAPI DPC++/C++ Compiler compilers.

This example shows mixed C and Fortran sources, linked using the Intel oneAPI DPC++/C++ Compiler. Consider the mixed source files `ibar.c`, `gbar.c`, and `foo.f`, where the main program is contained in ibar.c:

1. Compile `ibar.c` using the icx compiler:

```
icx -qopenmp -c ibar.c
```

2. Compile `gbar.c` using the `gcc` compiler:

```
gcc -fopenmp -c gbar.c
```

3. Compile `foo.f` using the `ifort` compiler:

```
ifort -qopenmp -c foo.f
```

4. Use the icx compiler to link the resulting object files:

```
icx -qopenmp foo.o ibar.o gbar.o
```

If the main program were contained in the Fortran file foo.f, the linking step must be performed by the ifort compiler.

> **NOTE**
> Do not mix objects created by the Intel Fortran Compiler Classic and Intel Fortran Compiler with the GNU Fortran Compiler (`gfortran`); instead, recompile all Fortran sources with either `ifort` or `ifx`, or recompile all Fortran sources with the GNU Fortran Compiler . The GNU Fortran Compiler is only available on Linux operating systems.

When using the GNU `gfortran` Compiler to link the application with the Intel oneAPI DPC++/C++ Compiler OpenMP compatibility library, you need to specify the following:

- The Intel® OpenMP compatibility library name and the Intel®`irc` libraries using the `-l` option
- The Linux `pthread` library using the `-l` option
- The path to the Intel® libraries where the Intel oneAPI DPC++/C++ Compiler is installed using the `-L` option

You do not need to specify the `-fopenmp` option on the link line.

For example, consider the mixed source files `ibar.c`, `gbar.c`, and `foo.f`:

1. Compile `ibar.c` using the icx compiler:

```
icx -qopenmp -c ibar.c
```

2. Compile `gbar.c` using the GCC compiler:

```
gcc -fopenmp -c gbar.c
```

3. Compile `foo.f` using the `gfortran` compiler:

```
gfortran -fopenmp -c foo.f
```

4. Use the `gfortran` compiler to link the application object code with the Intel OpenMP library. You do not need to specify the `-fopenmp` option in the link command:

```
gfortran foo.o ibar.o gbar.o -lirc -liomp5 -lpthread -lc -L<install_dir>/lib
```

where <install_dir> is the location of the installed Intel OpenMP library.

Alternately, you can use the Intel oneAPI DPC++/C++ Compiler. to link the application object code but need to pass multiple `gfortran` libraries using the `-l` options at the link step.

This example shows mixed C and GNU Fortran sources linked using the icx compiler. Consider the mixed source files `ibar.c` and `foo.f`:

1. Compile the C source with the icx compiler:

```
icx -qopenmp -c ibar.c
```

2. Compile the GNU Fortran source with `gfortran`:

```
gfortran -fopenmp -c foo.f
```

3. Use icx to link the resulting object files with the `-l` option to pass the needed `gfortran` libraries:

```
icx -qopenmp foo.o ibar.o -lgfortran
```

Windows Examples

This section shows several examples of using OpenMP with the Intel® C++ Compiler from the command line on Windows.

Compile and Link OpenMP Libraries

You can compile an application and link the Compatibility libraries with a single command using the `/Qopenmp` option. By default, the Intel oneAPI DPC++/C++ Compiler performs a dynamic link of the OpenMP libraries.

For example, to compile source file `hello.cpp` and link Compatibility libraries using the Intel® C++ Compiler:

```
icx /MD /Qopenmp hello.cpp
```

When using the Microsoft Visual C++ Compiler, you should link with the Intel® OpenMP compatibility library. You need to avoid linking the Microsoft OpenMP runtime library (`vcomp`) and explicitly pass the name of the Intel® OpenMP compatibility library as linker options using the `/link` option. For example:

```
cl /MD /openmp hello.cpp /link /nodefaultlib:vcomp libiomp5md.lib
```

Mix OpenMP Object Files Compiled with Visual C++ Compiler or Intel oneAPI DPC++/C++ Compiler

You can use the Intel oneAPI DPC++/C++ Compiler with the Visual C++ Compiler to compile parts of an application and create object files that can then be linked (object-level interoperability).

For example:

1. Compile `f1.c` and `f2.c` with the Visual C++ Compiler, using the `/openmp` option to enable OpenMP support:

```
cl /MD /openmp /c f1.c f2.c
```

   The `/c` prevents linking at this step.

2. Compile `f3.c` and `f4.c` with the icx compiler, using the `/Qopenmp` option to enable OpenMP support:

```
icx /MD /Qopenmp /c f3.c f4.c
```

3. Use the icx compiler to link the resulting application object code with the Intel C++ Compiler OpenMP library:

```
icx /MD /Qopenmp f1.obj f2.obj f3.obj f4.obj /Feapp /link /nodefaultlib:vcomp
```

   The `/Fe` specifies the generated executable file name.

   Alternatively, use the Visual C++ linker to link the application object code with the Compatibility library `libiomp5md.lib`:

```
link f1.obj f2.obj f3.obj f4.obj /out:app.exe /nodefaultlib:vcomp libiomp5md.lib
```

Use Intel OpenMP Libraries from Visual Studio

When running Windows, you can make certain changes in the Visual C++ Visual Studio development environment to use the Intel oneAPI DPC++/C++ Compiler and Visual C++ to create applications that use the Intel OpenMP libraries.

Set the project **Property Pages** to indicate the Intel OpenMP runtime library location:

1. Open the project's property pages in from the main menu: **Project** > **Properties** (or right-click the Project name and select **Properties**) .

2. Select **Configuration Properties** > **Linker** > **General** > **Additional Library Directories**.
3. Enter the path to the Intel®-provided compiler libraries. For example, for an IA-32 architecture system (C/C++ only), enter:

```
<Intel_compiler_installation_path>\windows\compiler\lib\ia32_win
```

Make the Intel OpenMP dynamic runtime library accessible at runtime; you must specify the corresponding path:

1. Open the project's property pages in from the main menu: **Project** > **Properties** (or right-click the Project name and select **Properties**).
2. Select **Configuration Properties** > **Debugging** > **Environment**.
3. Enter the path to the Intel®-provided compiler libraries. For example, for an IA-32 architecture system (C/C++ only), enter:

```
PATH=%PATH%;<Intel_compiler_installation_path>\windows\redist\ia32_win\compiler
```

Add the Intel OpenMP runtime library name to the linker options and exclude the default Microsoft OpenMP runtime library:

1. Open the project's property pages in from the main menu: **Project** > **Properties** (or right-click the Project name and select **Properties**).
2. Select **Configuration Properties** > **Linker** > **Command Line** > **Additional Options**.
3. Enter the OpenMP library name and the Visual C++ linker option, `/nodefaultlib:vcomp libiomp5md.lib`.

See Also

qopenmp, Qopenmp compiler option
Using IPO
OpenMP Support Libraries
qopenmp-link, Qopenmp-link compiler option

Thread Affinity Interface

The Intel® runtime library has the ability to bind OpenMP* threads to physical processing units. The interface is controlled using the `KMP_AFFINITY` environment variable. Depending on the system (machine) topology, application, and operating system, thread affinity can have a dramatic effect on the application speed.

Thread affinity restricts execution of certain threads (virtual execution units) to a subset of the physical processing units in a multiprocessor computer. Depending upon the topology of the machine, thread affinity can have a dramatic effect on the execution speed of a program.

Thread affinity is supported on Windows* systems and versions of Linux* systems that have kernel support for thread affinity.

The Intel OpenMP runtime library has the ability to bind OpenMP threads to physical processing units. There are three types of interfaces you can use to specify this binding, which are collectively referred to as the Intel OpenMP Thread Affinity Interface:

- The high-level affinity interface uses an environment variable to determine the machine topology and assigns OpenMP threads to the processors based upon their physical location in the machine. This interface is controlled entirely by the `KMP_AFFINITY` environment variable.
- The mid-level affinity interface uses an environment variable to explicitly specifies which processors (labeled with integer IDs) are bound to OpenMP threads. This interface provides compatibility with the GCC* `GOMP_AFFINITY` environment variable, but you can also invoke it by using the `KMP_AFFINITY` environment variable. The `GOMP_AFFINITY` environment variable is supported on Linux systems only, but users on Windows or Linux systems can use the similar functionality provided by the `KMP_AFFINITY` environment variable.
- The low-level affinity interface uses APIs to enable OpenMP threads to make calls into the OpenMP runtime library to explicitly specify the set of processors on which they are to be run. This interface is similar in nature to `sched_setaffinity` and related functions on Linux systems or to `SetThreadAffinityMask` and related functions on Windows systems. In addition, you can specify certain

options of the `KMP_AFFINITY` environment variable to affect the behavior of the low-level API interface. For example, you can set the affinity type `KMP_AFFINITY` to disabled, which disables the low-level affinity interface, or you could use the `KMP_AFFINITY` or `GOMP_AFFINITY` environment variables to set the initial affinity mask, and then retrieve the mask with the low-level API interface.

The following terms are used in this section:

- The total number of processing elements on the machine is referred to as the number of *OS thread contexts.*
- Each processing element is referred to as an Operating System processor, or *OS proc*.
- Each OS processor has a unique integer identifier associated with it, called an *OS proc ID*.
- The term *package* refers to a single or multi-core processor chip.
- The term *OpenMP Global Thread ID* (GTID) refers to an integer which uniquely identifies all threads known to the Intel OpenMP runtime library. The thread that first initializes the library is given GTID 0. In the normal case where all other threads are created by the library and when there is no nested parallelism, then *n-threads-var* - 1 new threads are created with GTIDs ranging from 1 to *ntheads-var* - 1, and each thread's GTID is equal to the OpenMP thread number returned by function `omp_get_thread_num`(). The high-level and mid-level interfaces rely heavily on this concept. Hence, their usefulness is limited in programs containing nested parallelism. The low-level interface does not make use of the concept of a GTID and can be used by programs containing arbitrarily many levels of parallelism.

Some environment variables are available for both Intel® microprocessors and non-Intel microprocessors, but may perform additional optimizations for Intel® microprocessors than for non-Intel microprocessors.

The KMP_AFFINITY Environment Variable

> **NOTE**
> You must set the `KMP_AFFINITY` environment variable before the first parallel region, or certain API calls including `omp_get_max_threads()`, `omp_get_num_procs()` and any affinity API calls, as described in Low Level Affinity API, below.

The `KMP_AFFINITY` environment variable uses the following general syntax:

Syntax
KMP_AFFINITY=[<*modifier*>,...]<*type*>[,<*permute*>][,<*offset*>]

For example, to list a machine topology map, specify `KMP_AFFINITY=verbose,none` to use a *modifier* of `verbose` and a *type* of `none`.

The following table describes the supported specific arguments.

Argument	**Default**	**Description**
`modifier`	`noverbose` `respect` `granularity=core`	Optional. String consisting of keyword and specifier. • `granularity=`<*specifier*> takes the following specifiers: fine, thread, core, tile, die, node, group, and socket • `norespect` • `noverbose` • `nowarnings` • `noreset` • `proclist={`<*proc-list*>`}` • `respect`

Argument	Default	Description
		• verbose • warnings • reset The syntax for *<proc-list>* is explained in mid-level affinity interface. **NOTE** On Windows with multiple processor groups, the norespect affinity modifier is assumed when the process affinity mask equals a single processor group (which is default on Windows). Otherwise, the respect affinity modifier is used.
type	none	Required string. Indicates the thread affinity to use. • balanced • compact • disabled • explicit • none • scatter • logical (deprecated; instead use compact, but omit any permute value) • physical (deprecated; instead use scatter, possibly with an offset value) The logical and physical types are deprecated but supported for backward compatibility.
permute	0	Optional. Positive integer value. Not valid with type values of explicit, none, or disabled.
offset	0	Optional. Positive integer value. Not valid with type values of explicit, none, or disabled.

Affinity Types

Type is the only required argument.

type = none (default)

Does not bind OpenMP threads to particular thread contexts; however, if the operating system supports affinity, the compiler still uses the OpenMP thread affinity interface to determine machine topology. Specify `KMP_AFFINITY=verbose,none` to list a machine topology map.

type = balanced

Places threads on separate cores until all cores have at least one thread, similar to the `scatter` type. However, when the runtime must use multiple hardware thread contexts on the same core, the `balanced` type ensures that the OpenMP thread numbers are close to each other, which `scatter` does not do. This affinity type is supported on the CPU only for single socket systems.

> **NOTE**
> The OpenMP* environment variable `OMP_PROC_BIND=spread` is similar to `KMP_AFFINITY=balanced` and is available on all platforms, including multi-socket CPU systems.

type = compact

Specifying `compact` assigns the OpenMP thread <*n*>+1 to a free thread context as close as possible to the thread context where the <*n*> OpenMP thread was placed. For example, in a topology map, the nearer a node is to the root, the more significance the node has when sorting the threads.

type = disabled

Specifying `disabled` completely disables the thread affinity interfaces. This forces the OpenMP run-time library to behave as if the affinity interface was not supported by the operating system. This includes the low-level API interfaces such as `kmp_set_affinity` and `kmp_get_affinity`, which have no effect and will return a nonzero error code.

type = explicit

Specifying `explicit` assigns OpenMP threads to a list of OS proc IDs that have been explicitly specified by using the proclist= modifier, which is required for this affinity type. See Explicitly Specifying OS Proc IDs (GOMP_CPU_AFFINITY).

type = scatter

Specifying `scatter` distributes the threads as evenly as possible across the entire system. `scatter` is the opposite of `compact`; so the leaves of the node are most significant when sorting through the machine topology map.

Deprecated Types: logical and physical

Types `logical` and `physical` are deprecated and may become unsupported in a future release. Both are supported for backward compatibility.

For `logical` and `physical` affinity types, a single trailing integer is interpreted as an `offset` specifier instead of a `permute` specifier. In contrast, with `compact` and `scatter` types, a single trailing integer is interpreted as a `permute` specifier.

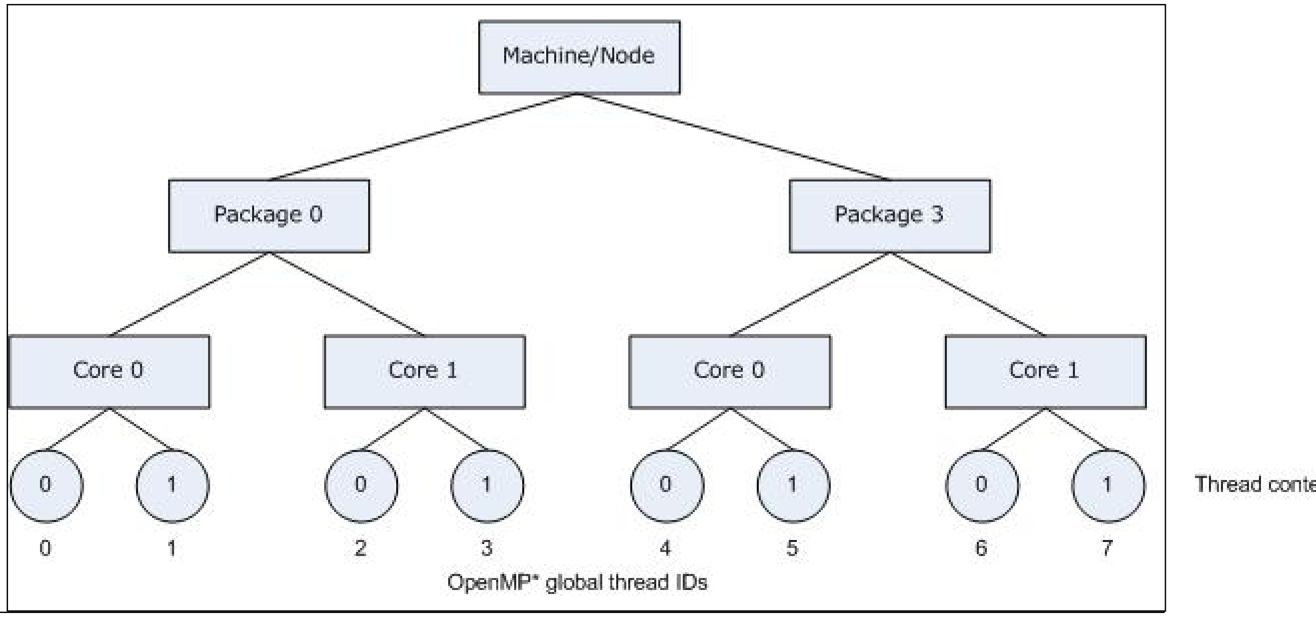
- Specifying `logical` assigns OpenMP threads to consecutive logical processors, which are also called hardware thread contexts. The type is equivalent to `compact`, except that the `permute` specifier is not allowed. Thus, `KMP_AFFINITY=logical,n` is equivalent to `KMP_AFFINITY=compact,0,n` (this equivalence is true regardless of the whether or not a `granularity=fine` modifier is present).
- Specifying `physical` assigns threads to consecutive physical processors (cores). For systems where there is only a single thread context per core, the type is equivalent to `logical`. For systems where multiple thread contexts exist per core, `physical` is equivalent to `compact` with a `permute` specifier of 1; that is, `KMP_AFFINITY=physical,n` is equivalent to `KMP_AFFINITY=compact,1,n` (regardless of the whether or

not a `granularity=fine` modifier is present). This equivalence means that when the compiler sorts the map it should permute the innermost level of the machine topology map to the outermost, presumably the thread context level. This type does not support the `permute` specifier.

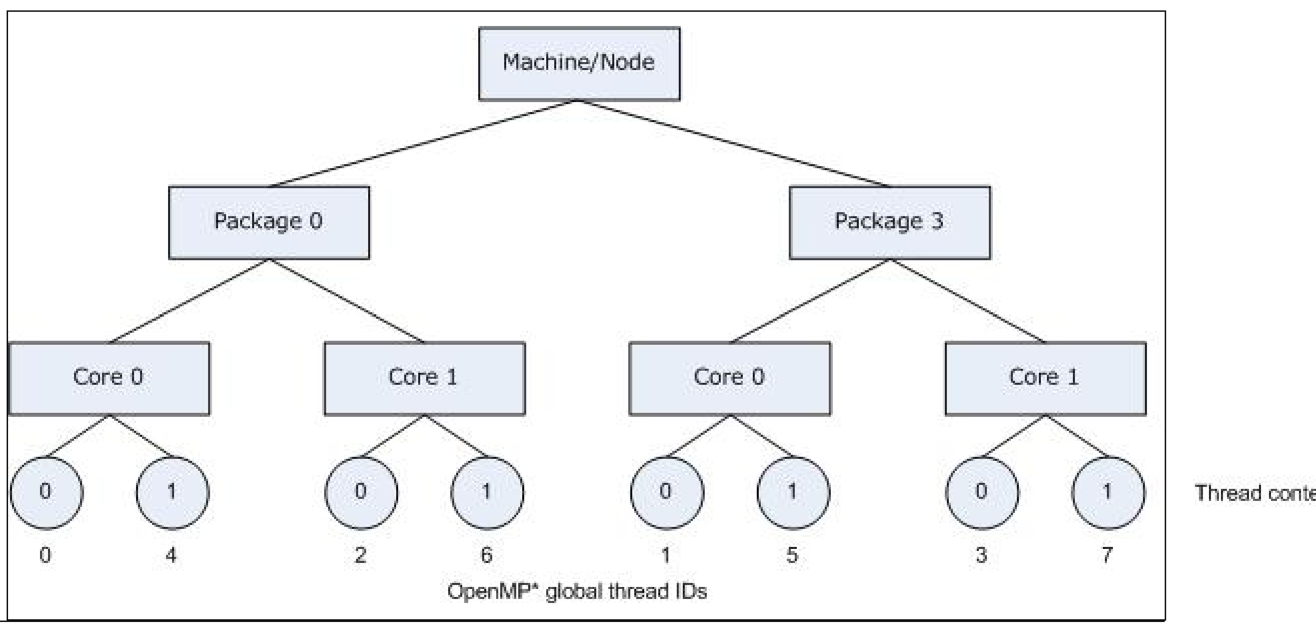
Examples of Types `compact` and `scatter`

The following figure illustrates the topology for a machine with two processors, and each processor has two cores; further, each core has Intel® Hyper-Threading Technology (Intel® HT Technology) enabled.

The following figure also illustrates the binding of OpenMP thread to hardware thread contexts when specifying `KMP_AFFINITY=granularity=fine,compact`.



Specifying `scatter` on the same system as shown in the figure above, the OpenMP threads would be assigned the thread contexts as shown in the following figure, which shows the result of specifying `KMP_AFFINITY=granularity=fine,scatter`.



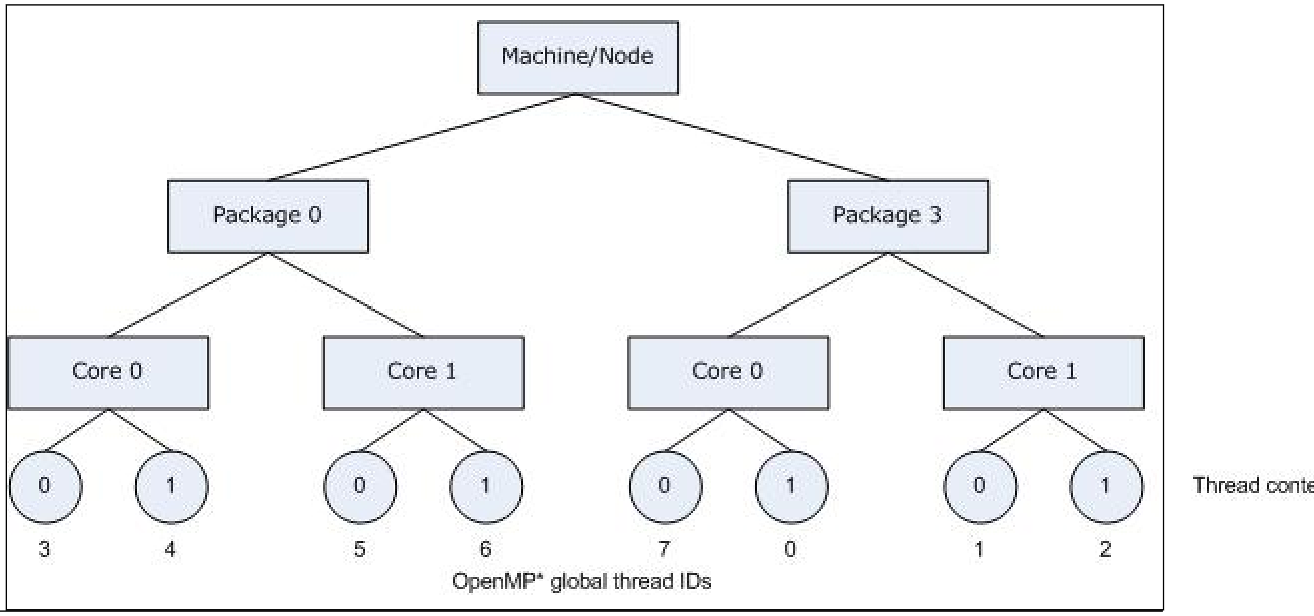
permute and offset combinations

For both `compact` and `scatter`, `permute` and `offset` are allowed; however, if you specify only one integer, the compiler interprets the value as a permute specifier. Both `permute` and `offset` default to 0.

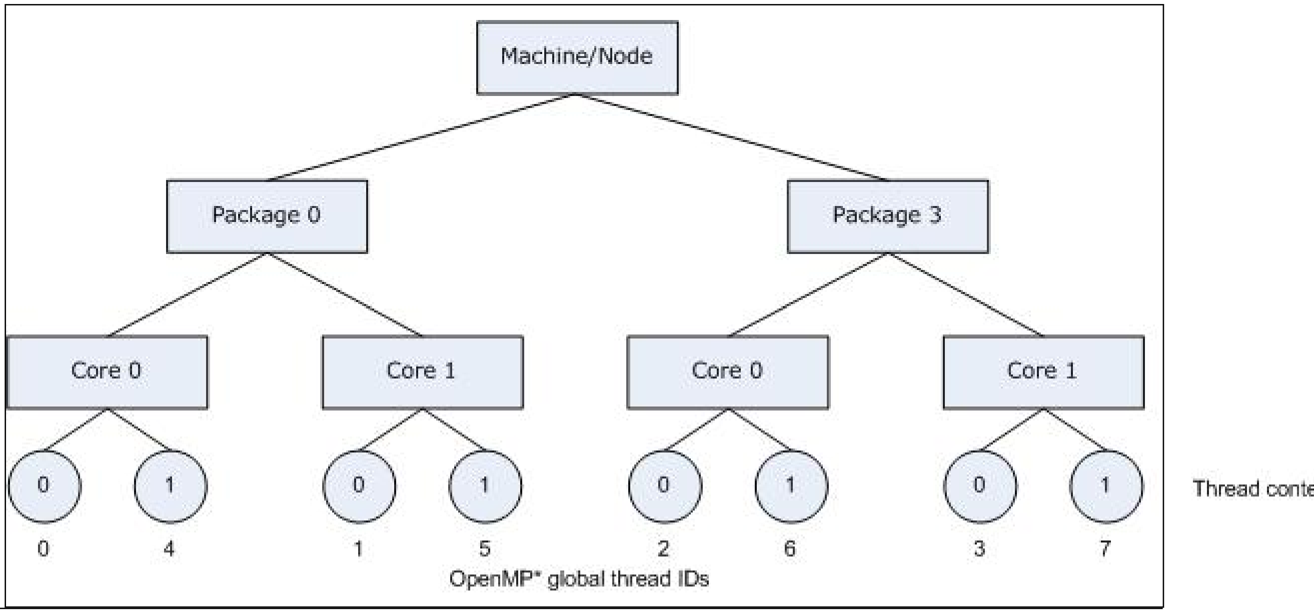
The `permute` specifier controls which levels are most significant when sorting the machine topology map. A value for permute forces the mappings to make the specified number of most significant levels of the sort the least significant, and it inverts the order of significance. The root node of the tree is not considered a separate level for the sort operations.

The `offset` specifier indicates the starting position for thread assignment.

The following figure illustrates the result of specifying `KMP_AFFINITY=granularity=fine,compact,0,5`.



Consider the hardware configuration from the previous example, running an OpenMP application which exhibits data sharing between consecutive iterations of loops. We would therefore like consecutive threads to be bound close together, as is done with `KMP_AFFINITY=compact`, so that communication overhead, cache line invalidation overhead, and page thrashing are minimized. Now, suppose the application also had a number of parallel regions which did not utilize all of the available OpenMP threads. It is desirable to avoid binding multiple threads to the same core and leaving other cores not utilized, since a thread normally executes faster on a core where it is not competing for resources with another active thread on the same core. Since a thread normally executes faster on a core where it is not competing for resources with another active thread on the same core, you might want to avoid binding multiple threads to the same core while leaving other cores unused. The following figure illustrates this strategy of using `KMP_AFFINITY=granularity=fine,compact,1,0` as a setting.



The OpenMP thread *n*+1 is bound to a thread context as close as possible to OpenMP thread *n*, but on a different core. Once each core has been assigned one OpenMP thread, the subsequent OpenMP threads are assigned to the available cores in the same order, but they are assigned on different thread contexts.

Modifier Values for Affinity Types

Modifiers are optional arguments that precede type. If you do not specify a modifier, the `noverbose`, `respect`, and `granularity=core` modifiers are used automatically.

Modifiers are interpreted in order from left to right, and they may conflict. Following conflicting modifier is ignored. For example, specifying `KMP_AFFINITY=verbose,noverbose,scatter` is therefore equivalent to setting `KMP_AFFINITY=verbose,scatter`.

modifier = noverbose (default)

Does not print verbose messages.

modifier = verbose

Prints messages concerning the supported affinity. The messages include information about the number of packages, number of cores in each package, number of thread contexts for each core, and OpenMP thread bindings to physical thread contexts.

Information about binding OpenMP threads to physical thread contexts is indirectly shown in the form of the mappings between hardware thread contexts and the operating system (OS) processor (proc) IDs. The affinity mask for each OpenMP thread is printed as a set of OS processor IDs.

For example, specifying `KMP_AFFINITY=verbose,scatter` on a dual core system with two processors, with Intel® Hyper-Threading Technology (Intel® HT Technology) disabled, results in a message listing similar to the following when then program is executed:

Verbose, scatter message

```
...
KMP_AFFINITY: Initial OS proc set respected: 0,1,2,3
KMP_AFFINITY: affinity capable, using hwloc.
KMP_AFFINITY: 4 available OS procs
KMP_AFFINITY: Uniform topology
KMP_AFFINITY: 2 sockets x 2 cores/socket x 1 threads/core (4 total cores)
```

Verbose, scatter message
``KMP_AFFINITY: OS proc to physical thread map:```

```
KMP_AFFINITY: OS proc to physical thread map:
KMP_AFFINITY: OS proc 0 maps to socket 0 core 0 thread 0
KMP_AFFINITY: OS proc 2 maps to socket 0 core 1 thread 0
KMP_AFFINITY: OS proc 1 maps to socket 3 core 0 thread 0
KMP_AFFINITY: OS proc 3 maps to socket 3 core 1 thread 0
KMP_AFFINITY: pid 79739 tid 79739 thread 0 bound to OS proc set 0
KMP_AFFINITY: pid 79739 tid 79740 thread 2 bound to OS proc set 2
KMP_AFFINITY: pid 79739 tid 79741 thread 3 bound to OS proc set 3
KMP_AFFINITY: pid 79739 tid 79742 thread 1 bound to OS proc set 1
```

The verbose modifier generates several standard, general messages. The following table summarizes how to read the messages.

Message String	Description
"affinity capable"	Indicates that all components (compiler, operating system, and hardware) support affinity, so thread binding is possible.
"decoding x2APIC ids"	Indicates that the machine topology was discovered by binding a thread to each operating system processor and decoding the output of the `cpuid` instruction.
"using hwloc"	Indicates that the Portable Hardware Locality* (`hwloc`) library used to determine machine topology.
"using /proc/cpuinfo"	Linux only. Indicates that `cpuinfo` is being used to determine machine topology.
"using flat"	Operating system processor ID is assumed to be equivalent to physical package ID. This method of determining machine topology is used if none of the other methods will work, but may not accurately detect the actual machine topology.
"uniform topology"	The machine topology map is a full tree with no missing leaves at any level.

The mapping from the operating system processors to thread context ID is printed next. The binding of OpenMP thread context ID is printed next unless the affinity type is none. For more information, see Determining Machine Topology.

modifier = granularity

Binding OpenMP threads to particular packages and cores will often result in a performance gain on systems with Intel processors with Intel® Hyper-Threading Technology (Intel® HT Technology) enabled; however, it is usually not beneficial to bind each OpenMP thread to a particular thread context on a specific core. Granularity describes the lowest levels that OpenMP threads are allowed to float within a topology map.

This modifier supports the following additional specifiers.

Specifier	Description
`core`	Default. Allows all the OpenMP threads bound to a core to float between the different thread contexts.
`fine` or `thread`	The finest granularity level. Causes each OpenMP thread to be bound to a single thread context. The two specifiers are functionally equivalent.

Specifier	Description
`tile`, or `die`, or `node`, or `group`, or `socket`	Allows all the OpenMP threads bound to a tile, or die, or NUMA node, or group, or socket to float between the different thread contexts of cores the tile, or die, or NUMA node, or group, or socket consists of.

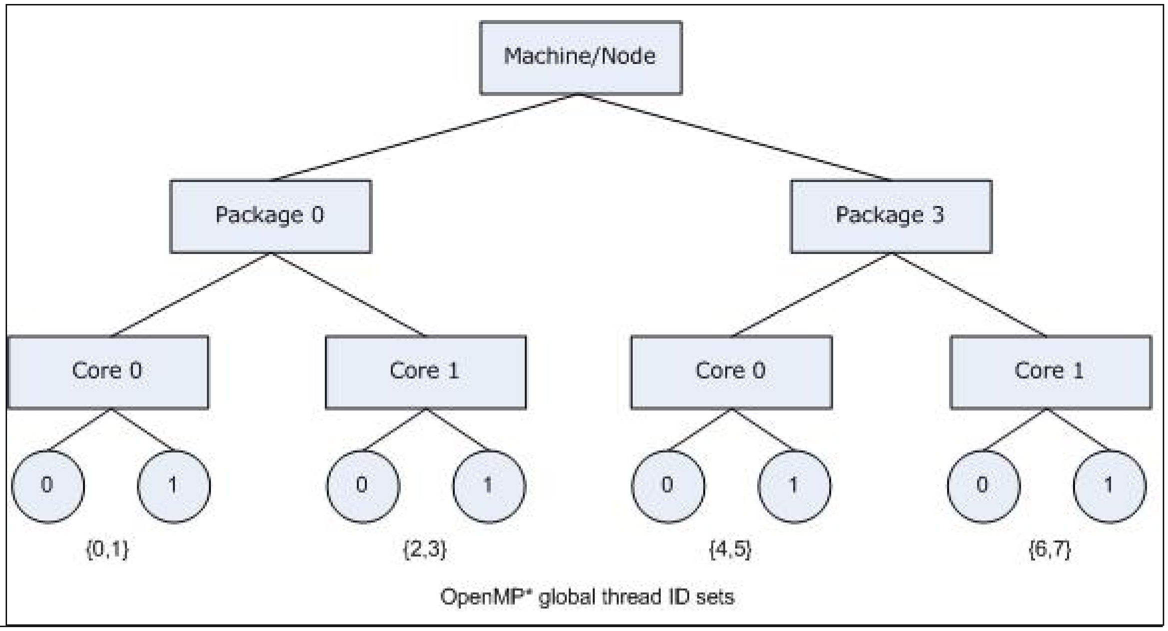
Specifying `KMP_AFFINITY=verbose,granularity=core,compact` on the same dual core system with two processors as in the previous section, but with Intel® Hyper-Threading Technology (Intel® HT Technology) enabled, results in a message listing similar to the following when the program is executed:

Verbose, granularity=core,compact message

```
KMP_AFFINITY: Initial OS proc set respected: 0-7
KMP_AFFINITY: decoding x2APIC ids.
KMP_AFFINITY: 8 available OS procs
KMP_AFFINITY: Uniform topology
KMP_AFFINITY: 2 sockects x 2 cores/socket x 2 threads/core (4 total cores)
KMP_AFFINITY: OS proc to physical thread map:
KMP_AFFINITY: OS proc 0 maps to socket 0 core 0 thread 0
KMP_AFFINITY: OS proc 4 maps to socket 0 core 0 thread 1
KMP_AFFINITY: OS proc 2 maps to socket 0 core 1 thread 0
KMP_AFFINITY: OS proc 6 maps to socket 0 core 1 thread 1
KMP_AFFINITY: OS proc 1 maps to socket 3 core 0 thread 0
KMP_AFFINITY: OS proc 5 maps to socket 3 core 0 thread 1
KMP_AFFINITY: OS proc 3 maps to socket 3 core 1 thread 0
KMP_AFFINITY: OS proc 7 maps to socket 3 core 1 thread 1
KMP_AFFINITY: pid 40880 tid 40880 thread 0 bound to OS proc set 0,4
KMP_AFFINITY: pid 40880 tid 40881 thread 1 bound to OS proc set 0,4
KMP_AFFINITY: pid 40880 tid 40882 thread 2 bound to OS proc set 2,6
KMP_AFFINITY: pid 40880 tid 40883 thread 3 bound to OS proc set 2,6
KMP_AFFINITY: pid 40880 tid 40884 thread 4 bound to OS proc set 1,5
KMP_AFFINITY: pid 40880 tid 40885 thread 5 bound to OS proc set 1,5
KMP_AFFINITY: pid 40880 tid 40886 thread 6 bound to OS proc set 3,7
KMP_AFFINITY: pid 40880 tid 40887 thread 7 bound to OS proc set 3,7
```

The affinity mask for each OpenMP thread is shown in the listing (above) as the set of operating system processor to which the OpenMP thread is bound.

The following figure illustrates the machine topology map, for the above listing, with OpenMP thread bindings.



In contrast, specifying `KMP_AFFINITY=verbose,granularity=fine,compact` or `KMP_AFFINITY=verbose,granularity=thread,compact` binds each OpenMP thread to a single hardware thread context when the program is executed:

Verbose, granularity=fine,compact message

```
KMP_AFFINITY: Initial OS proc set respected: 0-7
KMP_AFFINITY: decoding x2APIC ids.
KMP_AFFINITY: 8 available OS procs
KMP_AFFINITY: Uniform topology
KMP_AFFINITY: 2 sockets x 2 cores/socket x 2 threads/core (4 total cores)
KMP_AFFINITY: OS proc to physical thread map:
KMP_AFFINITY: OS proc 0 maps to socket 0 core 0 thread 0
KMP_AFFINITY: OS proc 4 maps to socket 0 core 0 thread 1
KMP_AFFINITY: OS proc 2 maps to socket 0 core 1 thread 0
KMP_AFFINITY: OS proc 6 maps to socket 0 core 1 thread 1
KMP_AFFINITY: OS proc 1 maps to socket 3 core 0 thread 0
KMP_AFFINITY: OS proc 5 maps to socket 3 core 0 thread 1
KMP_AFFINITY: OS proc 3 maps to socket 3 core 1 thread 0
KMP_AFFINITY: OS proc 7 maps to socket 3 core 1 thread 1
KMP_AFFINITY: pid 40895 tid 40895 thread 0 bound to OS proc set 0
KMP_AFFINITY: pid 40895 tid 40896 thread 1 bound to OS proc set 4
KMP_AFFINITY: pid 40895 tid 40897 thread 2 bound to OS proc set 2
KMP_AFFINITY: pid 40895 tid 40898 thread 3 bound to OS proc set 6
KMP_AFFINITY: pid 40895 tid 40899 thread 4 bound to OS proc set 1
KMP_AFFINITY: pid 40895 tid 40900 thread 5 bound to OS proc set 5
KMP_AFFINITY: pid 40895 tid 40901 thread 6 bound to OS proc set 3
KMP_AFFINITY: pid 40895 tid 40902 thread 7 bound to OS proc set 7
```

The OpenMP to hardware context binding for this example was illustrated in the first example.

Specifying `granularity=fine` will always cause each OpenMP thread to be bound to a single OS processor. This is equivalent to `granularity=thread`, currently the finest granularity level.

modifier = respect (default)

Respect the process' original affinity mask, or more specifically, the affinity mask in place for the thread that initializes the OpenMP run-time library. The behavior differs between Linux and Windows:

- On Windows: Respect original affinity mask for the process.
- On Linux: Respect the affinity mask for the thread that initializes the OpenMP run-time library.

NOTE On Windows with multiple processor groups, the `norespect` affinity modifier is the default when the process affinity mask equals a single processor group (which is default on Windows). Otherwise, the respect affinity modifier is the default.

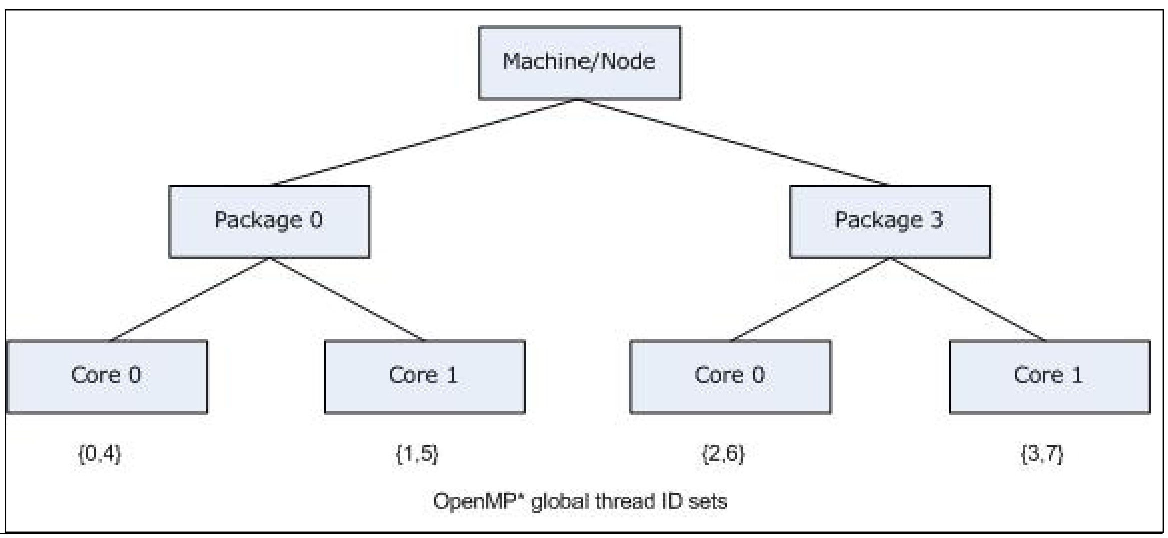
Specifying `KMP_AFFINITY=verbose,compact` for the same system used in the previous example, with Intel® Hyper-Threading Technology (Intel® HT Technology) enabled, and invoking the library with an initial affinity mask of {4,5,6,7} (thread context 1 on every core) causes the compiler to model the machine as a dual core, two-processor system with Intel® HT Technology disabled.

Verbose,compact message

```
KMP_AFFINITY: Initial OS proc set respected: 4-7
KMP_AFFINITY: decoding x2APIC ids.
KMP_AFFINITY: 4 available OS procs
KMP_AFFINITY: Uniform topology
KMP_AFFINITY: 2 sockets x 2 cores/socket x 1 threads/core (4 total cores)
KMP_AFFINITY: OS proc to physical thread map:
KMP_AFFINITY: OS proc 4 maps to socket 0 core 0 thread 1
KMP_AFFINITY: OS proc 6 maps to socket 0 core 1 thread 1
KMP_AFFINITY: OS proc 5 maps to socket 3 core 0 thread 1
KMP_AFFINITY: OS proc 7 maps to socket 3 core 1 thread 1
KMP_AFFINITY: pid 41032 tid 41032 thread 0 bound to OS proc set 4
KMP_AFFINITY: pid 41032 tid 41033 thread 1 bound to OS proc set 6
KMP_AFFINITY: pid 41032 tid 41034 thread 2 bound to OS proc set 5
KMP_AFFINITY: pid 41032 tid 41035 thread 3 bound to OS proc set 7
```

Because there are four thread contexts accessible on the machine, by default the compiler created four threads for an OpenMP `parallel` construct.

The following figure illustrates the corresponding machine topology map and threads placement in case eight OpenMP threads requested via `OMP_NUM_THREADS=8`



When using the local `cpuid` information to determine the machine topology, it is not always possible to distinguish between a machine that does not support Intel® Hyper-Threading Technology (Intel® HT Technology) and a machine that supports it, but has it disabled. Therefore, the compiler does not include a level in the map if the elements (nodes) at that level had no siblings, with the exception that the package level is always modeled. As mentioned earlier, the package level will always appear in the topology map, even if there only a single package in the machine.

modifier = norespect

Do not respect original affinity mask for the process. Binds OpenMP threads to all operating system processors.

In early versions of the OpenMP run-time library that supported only the `physical` and `logical` affinity types, `norespect` was the default and was not recognized as a modifier.

The default was changed to respect when types `compact` and `scatter` were added; therefore, thread bindings may have changed with the newer compilers in situations where the application specified a partial initial thread affinity mask.

modifier = nowarnings

Do not print warning messages from the affinity interface.

modifier = warnings (default)

Print warning messages from the affinity interface (default).

modifier = noreset (default)

Do not reset the primary thread's affinity after each outermost parallel region is complete. This setting preserves the primary thread's OpenMP affinity setting between parallel regions. For example, if `KMP_AFFINITY=compact,granularity=core`, then the primary thread's affinity is set to the first core for the first parallel region and kept that way for the thread's lifetime, even during serial regions.

modifier = reset

Reset the primary thread's affinity after each outermost parallel region is complete. This setting will reset the primary thread's affinity back to the initial affinity before OpenMP was initialized after each outermost parallel region is complete.

Determining Machine Topology

On IA-32 and Intel® 64 architecture systems, if the package has an APIC (Advanced Programmable Interrupt Controller), the compiler will use the `cpuid` instruction to obtain the `package id`, `core id`, and `thread context id`. Under normal conditions, each thread context on the system is assigned a unique APIC ID at boot time. The compiler obtains other pieces of information obtained by using the *cpuid* instruction, which together with the number of OS thread contexts (total number of processing elements on the machine), determine how to break the APIC ID down into the `package ID`, `core ID`, and `thread context ID`.

There are several ways to specify the APIC ID in the *cpuid* instruction - the legacy method in leaf 4, and the more modern method in leaf 11 and leaf 31. Only 256 unique APIC IDs are available in leaf 4. Leaf 11 and leaf 31 have no such limitation.

Normally, all `core ids` on a package and all `thread context ids` on a core are contiguous; however, numbering assignment gaps are common for `package ids`, as shown in the figure above.

If the compiler cannot determine the machine topology using any other method, but the operating system supports affinity, a warning message is printed, and the topology is assumed to be `flat`. For example, a flat topology assumes the operating system process *N* maps to package *N*, and there exists only one thread context per core and only one core for each package.

If the machine topology cannot be accurately determined as described above, the user can manually copy `/proc/cpuinfo` to a temporary file, correct any errors, and specify the machine topology to the OpenMP runtime library via the environment variable `KMP_CPUINFO_FILE=<temp_filename>`, as described in the section `KMP_CPUINFO_FILE` and `/proc/cpuinfo`.

Regardless of the method used in determining the machine topology, if there is only one thread context per core for every core on the machine, the thread context level will not appear in the topology map. If there is only one core per package for every package in the machine, the core level will not appear in the machine topology map. The topology map need not be a full tree, because different packages may contain a different number of cores, and different cores may support a different number of thread contexts.

The package level will always appear in the topology map, even if there only a single package in the machine.

KMP_CPUINFO_FILE and /proc/cpuinfo

One of the methods the Intel® oneAPI DPC++/C++ Compiler OpenMP runtime library can use to detect the machine topology on Linux systems is to parse the contents of `/proc/cpuinfo`. If the contents of this file (or a device mapped into the Linux file system) are insufficient or erroneous, you can consider copying its contents to a writable temporary file `<temp_file>`, correct it or extend it with the necessary information, and set `KMP_CPUINFO_FILE=<temp_file>`.

If you do this, the OpenMP runtime library will read the `<temp_file>` location pointed to by `KMP_CPUINFO_FILE` instead of the information contained in `/proc/cpuinfo` or attempting to detect the machine topology by decoding the APIC IDs. That is, the information contained in the *<temp_file>* overrides these other methods. You can use the KMP_CPUINFO_FILE interface on Windows systems, where `/proc/cpuinfo` does not exist.

The content of `/proc/cpuinfo` or `<temp_file>` should contain a list of entries for each processing element on the machine. Each processor element contains a list of entries (descriptive name and value on each line). A blank line separates the entries for each processor element. Only the following fields are used to determine the machine topology from each entry, either in *<temp_file>* or `/proc/cpuinfo`:

Field	Description
processor :	Specifies the OS ID for the processing element. The OS ID must be unique. The `processor` and `physical id` fields are the only ones that are required to use the interface.
physical id :	Specifies the package ID, which is a physical chip ID. Each package may contain multiple cores. The package level always exists in the compiler's OpenMP run-time library model of the machine topology.
core id :	Specifies the core ID. If it does not exist, it defaults to 0. If every package on the machine contains only a single core, the core level will not exist in the machine topology map (even if some of the core ID fields are non-zero).
apicid :	Specifies the thread ID. If it does not exist, it defaults to 0. If every core on the machine contains only a single thread, the thread level will not exist in the machine topology map (even if some thread ID fields are non-zero).
node_*n* id :	This is a extension to the normal contents of `/proc/cpuinfo` that can be used to specify the nodes at different levels of the memory interconnect on Non-Uniform Memory Access

Field	Description
	(NUMA) systems. Arbitrarily many levels *n* are supported. The node_0 level is closest to the package level; multiple packages comprise a node at level 0. Multiple nodes at level 0 comprise a node at level 1, and so on.

Each entry must be spelled exactly as shown, in lowercase, followed by optional whitespace, a colon (:), more optional whitespace, then the integer ID. Fields other than those listed are simply ignored.

NOTE
It is common for the `thread id` field to be missing from `/proc/cpuinfo` on many Linux variants, and for a field labeled `siblings` to specify the number of threads per node or number of nodes per package. However, the Intel OpenMP runtime library ignores fields labeled `siblings` so it can distinguish between the `thread id` and `siblings` fields. When this situation arises, the warning message `Physical node/pkg/core/thread ids not unique` appears (unless the `type specified is nowarnings`).

Windows Processor Groups

On a 64-bit Windows operating system, it is possible for multiple processor groups to accommodate more than 64 processors. Each group is limited in size, up to a maximum value of sixty-four (64) processors.

If multiple processor groups are detected, the default is to model the machine as a 2-level tree, where level 0 are for the processors in a group, and level 1 are for the different groups. Threads are assigned to a group until there are as many OpenMP threads bound to the groups as there are processors in the group. Subsequent threads are assigned to the next group, and so on.

By default, threads are allowed to float among all processors in a group, that is to say, granularity equals the group [granularity=group]. You can override this binding and explicitly use another affinity type like compact, scatter, and so on. If you do so, the granularity must be sufficiently fine to prevent a thread from being bound to multiple processors in different groups.

Using a Specific Machine Topology Modeling Method (KMP_TOPOLOGY_METHOD)

You can set the `KMP_TOPOLOGY_METHOD` environment variable to force OpenMP to use a particular machine topology modeling method.

Value	Description
`cpuid_leaf11`	Decodes the APIC identifiers as specified by leaf 11 of the *cpuid* instruction.
`cpuid_leaf4`	Decodes the APIC identifiers as specified in leaf 4 of the *cpuid* instruction.
`cpuinfo`	If `KMP_CPUINFO_FILE` is not specified, forces OpenMP to parse `/proc/cpuinfo` to determine the topology (Linux only). If `KMP_CPUINFO_FILE` is specified as described above, uses it (Windows or Linux).

Value	Description
`group`	Models the machine as a 2-level map, with level 0 specifying the different processors in a group, and level 1 specifying the different groups (Windows 64-bit only) .
`flat`	Models the machine as a flat (linear) list of processors.
`hwloc`	Models the machine as the Portable Hardware Locality* (hwloc) library does. This model is the most detailed and includes, but is not limited to: numa nodes, packages, cores, hardware threads, caches, and Windows processor groups.

Explicitly Specifying OS Processor IDs (GOMP_CPU_AFFINITY)

NOTE
You must set the `GOMP_CPU_AFFINITY` environment variable before the first parallel region, or certain API calls including `omp_get_max_threads()`, `omp_get_num_procs()` and any affinity API calls, as described in Low Level Affinity API, below.

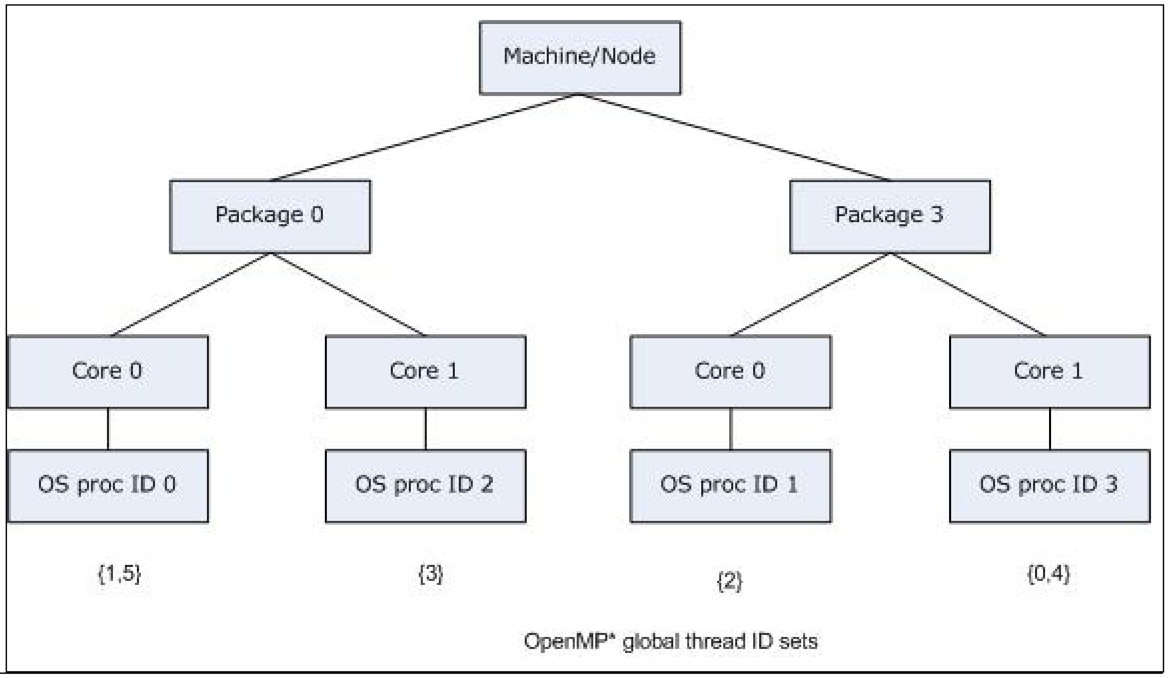
Instead of allowing the library to detect the hardware topology and automatically assign OpenMP threads to processing elements, the user may explicitly specify the assignment by using a list of operating system (OS) processor (proc) IDs. However, this requires knowledge of which processing elements the OS proc IDs represent.

On Linux systems, when using the Intel OpenMP compatibility libraries enabled by the compiler option `-qopenmp-lib compat`, you can use the `GOMP_AFFINITY` environment variable to specify a list of OS processor IDs. Its syntax is identical to that accepted by `libgomp` (assume that <*proc_list*> produces the entire `GOMP_AFFINITY` environment string):

Value	Description
<*proc_list*> :=	<entry> | <*elem*> , <list> | <elem> <whitespace> <list>
<*elem*> :=	<*proc_spec*> | <*range*>
<*proc_spec*> :=	<*proc_id*>
<*range*> :=	<*proc_id*> - <*proc_id*> | <*proc_id*> - <*proc_id*> : <int>
<*proc_id*> :=	<positive_int>

OS processors specified in this list are then assigned to OpenMP threads, in order of OpenMP Global Thread IDs. If more OpenMP threads are created than there are elements in the list, then the assignment occurs modulo the size of the list. That is, OpenMP Global Thread ID *n* is bound to list element *n* mod <*list_size*>.

Consider the machine previously mentioned: a dual core, dual-package machine without Intel® Hyper-Threading Technology (Intel® HT Technology) enabled, where the OS proc IDs are assigned in the same manner as the example in a previous figure. Suppose that the application creates six OpenMP threads instead of 4 (the default), oversubscribing the machine. If `GOMP_AFFINITY=3,0-2`, then OpenMP threads are bound as shown in the figure below, just as should happen when compiling with `gcc` and linking with `libgomp`:

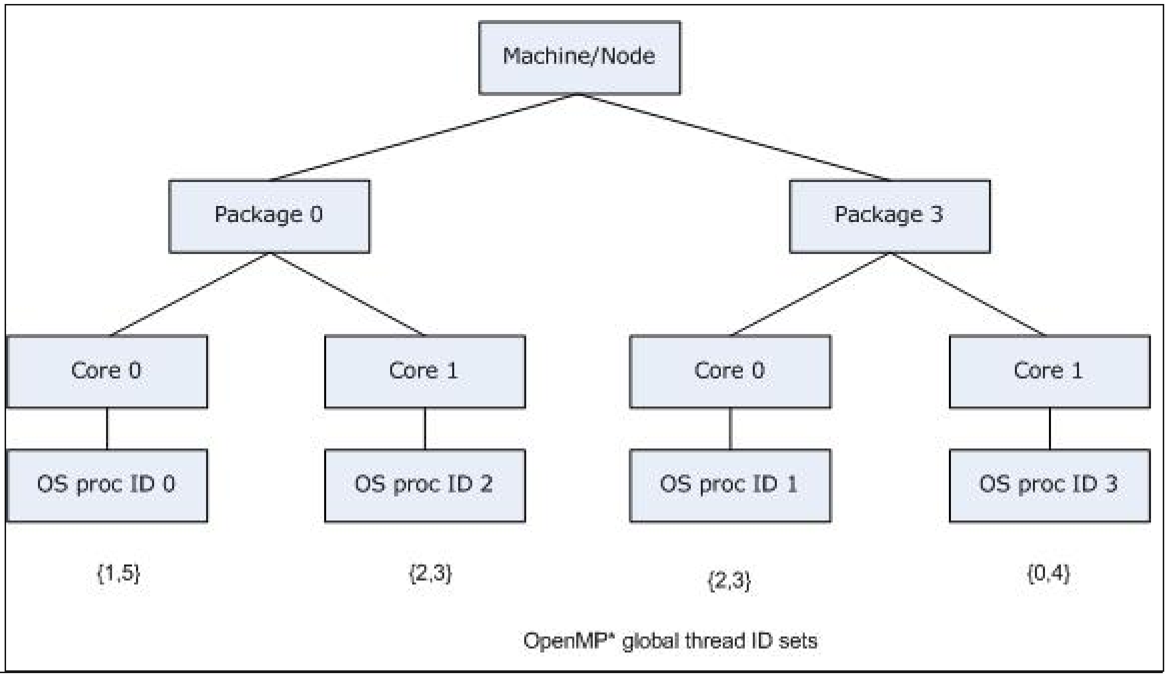


The same syntax can be used to specify the OS proc ID list in the proclist=[<proc_list>] modifier in the `KMP_AFFINITY` environment variable string. There is a slight difference: in order to have strictly the same semantics as in the `gcc` OpenMP runtime library `libgomp`: the `GOMP_AFFINITY` environment variable implies `granularity=fine`. If you specify the OS proc list in the `KMP_AFFINITY` environment variable without a `granularity=` specifier, then the default `granularity` is not changed. That is, OpenMP threads are allowed to float between the different thread contexts on a single core. Thus `GOMP_AFFINITY=<proc_list>` is an alias for `KMP_AFFINITY="granularity=fine,proclist=[<proc_list>],explicit"`.

In the `KMP_AFFINITY` environment variable string, the syntax is extended to handle operating system processor ID sets. The user may specify a set of operating system processor IDs among which an OpenMP thread may execute ("float") enclosed in brackets:

Value	**Description**
<*proc_list*> :=	<proc_id> | { <float_list> }
<*float_list*> :=	<proc_id> | <proc_id> , <float_list>

This allows functionality similarity to the `granularity= specifier`, but it is more flexible. The OS processors on which an OpenMP thread executes may exclude other OS processors nearby in the machine topology, but include other distant OS processors. Building upon the previous example, we may allow OpenMP threads 2 and 3 to "float" between OS processor 1 and OS processor 2 by using `KMP_AFFINITY="granularity=verbose,fine,proclist=[3,0,{1,2},{1,2}],explicit"`, as shown in the figure below:



If `verbose` were also specified, the output when the application is executed would include:

KMP_AFFINITY="granularity=verbose,fine,proclist=[3,0,{1,2},{1,2}],explicit"

```
KMP_AFFINITY: Initial OS proc set respected: 0,1,2,3
KMP_AFFINITY: decoding x2APIC ids.
KMP_AFFINITY: 4 available OS procs
KMP_AFFINITY: Uniform topology
KMP_AFFINITY: 2 sockets x 2 cores/socket x 1 threads/core (4 total cores)
KMP_AFFINITY: OS proc to physical thread map:
KMP_AFFINITY: OS proc 0 maps to socket 0 core 0 thread 0
KMP_AFFINITY: OS proc 2 maps to socket 0 core 1 thread 0
KMP_AFFINITY: OS proc 1 maps to socket 3 core 0 thread 0
KMP_AFFINITY: OS proc 3 maps to socket 3 core 1 thread 0
KMP_AFFINITY: pid 41464 tid 41464 thread 0 bound to OS proc set 3
KMP_AFFINITY: pid 41464 tid 41465 thread 1 bound to OS proc set 0
KMP_AFFINITY: pid 41464 tid 41466 thread 2 bound to OS proc set 1,2
KMP_AFFINITY: pid 41464 tid 41467 thread 3 bound to OS proc set 1,2
KMP_AFFINITY: pid 41464 tid 41468 thread 4 bound to OS proc set 3
KMP_AFFINITY: pid 41464 tid 41469 thread 5 bound to OS proc set 0
```

Low Level Affinity API

Instead of relying on the user to specify the OpenMP thread to OS proc binding by setting an environment variable before program execution starts (or by using the `kmp_settings` interface before the first parallel region is reached), each OpenMP thread can determine the desired set of OS procs on which it is to execute and bind to them with the `kmp_set_affinity` API call.

Caution
When you use this affinity interface you take complete control of the hardware resources on which your threads run. To do that sensibly you need to understand in detail how the logical CPUs, the enumeration of hardware threads controlled by the OS, map to the physical hardware of the specific machine on which you are running. That mapping can be, and likely is, different on different machines, so you risk binding machine-specific information into your code, which can result in explicitly forcing bad affinities when your code runs on a different machine. And if you are concerned with optimization at this level of detail, your code is probably valuable, and therefore will probably move to another machine.

This interface may also allow you to ignore the resource limitations that were set by the program startup mechanism, such as Message Passing Interface (MPI), specifically to prevent multiple OpenMP processes on the same node from using the same hardware threads. Again, this can result in explicitly forcing affinities that cause bad performance, and the OpenMP runtime will neither prevent this from happening, nor warn you when it does. These are expert interfaces and you must use them with caution.

It is recommended, therefore, to use the higher level affinity settings if you possibly can, because they are more portable and do not require this low level knowledge.

The C/C++ API interfaces follow, where the type name `kmp_affinity_mask_t` is defined in `omp.h`:

Syntax	Description
`int kmp_set_affinity (kmp_affinity_mask_t *mask)`	Sets the affinity mask for the current OpenMP thread to `*mask`, where `*mask` is a set of OS proc IDs that has been created using the API calls listed below, and the thread will only execute on OS procs in the set. Returns either a zero (0) upon success or a nonzero error code.
`int kmp_get_affinity (kmp_affinity_mask_t *mask)`	Retrieves the affinity mask for the current OpenMP thread, and stores it in `*mask`, which must have previously been initialized with a call to `kmp_create_affinity_mask()`. Returns either a zero (0) upon success or a nonzero error code.
`int kmp_get_affinity_max_proc (void)`	Returns the maximum OS proc ID that is on the machine, plus 1. All OS proc IDs are guaranteed to be between 0 (inclusive) and `kmp_get_affinity_max_proc()` (exclusive).
`void kmp_create_affinity_mask (kmp_affinity_mask_t *mask)`	Allocates a new OpenMP thread affinity mask, and initializes `*mask` to the empty set of OS procs. The implementation is free to use an object of `kmp_affinity_mask_t` either as the set itself, a pointer to the actual set, or an index into a table describing the set. Do not make any assumption as to what the actual representation is.
`void kmp_destroy_affinity_mask (kmp_affinity_mask_t *mask)`	Deallocates the OpenMP thread affinity mask. For each call to `kmp_create_affinity_mask()`, there should be a corresponding call to `kmp_destroy_affinity_mask()`.

Syntax	Description
`int kmp_set_affinity_mask_proc (int proc, kmp_affinity_mask_t *mask)`	Adds the OS proc ID proc to the set `*mask`, if it is not already. Returns either a zero (0) upon success or a nonzero error code.
`int kmp_unset_affinity_mask_proc (int proc, kmp_affinity_mask_t *mask)`	If the OS proc ID proc is in the set `*mask`, it removes it. Returns either a zero (0) upon success or a nonzero error code.
`int kmp_get_affinity_mask_proc (int proc, kmp_affinity_mask_t *mask)`	Returns 1 if the OS proc ID proc is in the set `*mask`; if not, it returns 0.

Once an OpenMP thread has set its own affinity mask via a successful call to `kmp_set_affinity()`, then that thread remains bound to the corresponding OS proc set until at least the end of the parallel region, unless reset via a subsequent call to `kmp_set_affinity()`.

Between parallel regions, the affinity mask (and the corresponding OpenMP thread to OS proc bindings) can be considered thread private data objects, and have the same persistence as described in the OpenMP Application Program Interface. For more information, see the OpenMP API specification (http://www.openmp.org), some relevant parts of which are provided below:

In order for the affinity mask and thread binding to persist between two consecutive active parallel regions, all three of the following conditions must hold:

- Neither parallel region is nested inside another explicit parallel region.
- The number of threads used to execute both parallel regions is the same.
- The value of the dyn-var internal control variable in the enclosing task region is false at entry to both parallel regions."

Therefore, by creating a parallel region at the start of the program whose sole purpose is to set the affinity mask for each thread, you can mimic the behavior of the `KMP_AFFINITY` environment variable with low-level affinity API calls, if program execution obeys the three aforementioned rules from the OpenMP specification.

The following example shows how these low-level interfaces can be used. This code binds the executing thread to the specified logical CPU:

Example

```
// Force the executing thread to execute on logical CPU i
// Returns 1 on success, 0 on failure.
int forceAffinity(int i)
{
kmp_affinity_mask_t mask;

kmp_create_affinity_mask(&mask);
kmp_set_affinity_mask_proc(i, &mask);

return (kmp_set_affinity(&mask) == 0);
}
```

This program fragment was written with knowledge about the mapping of the OS proc IDs to the physical processing elements of the target machine. On another machine, or on the same machine with a different OS installed, the program would still run, but the OpenMP thread to physical processing element bindings could differ and you might be explicitly force a bad distribution.

OpenMP* Memory Spaces and Allocators

For storage and retrieval variables, OpenMP* provides memory known as memory spaces. Different memory spaces have different traits. Depending on how a variable is to be used and accessed determines which memory space is appropriate for allocation of the variable.

Each memory space has a unique allocator that is used to allocate and deallocate memory in that space. The allocators allocate variables in contiguous space that does not overlap any other allocation in the memory space. Multiple memory spaces with different traits may map to a single memory resource.

The behavior of the allocator is affected by the allocator traits that you specify. The allocator traits, their possible values, and their default values are shown in the following table:

Allocator Trait	Values That Can Be Specified	Default Value
`access`	• `all` • `cgroup` • `pteam` • `thread`	All
`alignment`	A positive integer value that is a power of 2 specifying number of bytes	1 byte
`fallback`	• `abort_fb` • `allocator_fb` • `default_mem_fb` • `null_fb`	`default_mem_fb`
`fb_data`	An allocator handle	None
`partition`	• `blocked` • `environment` • `interleaved` • `nearest`	`environment`
`pinned`	• `true` • `false`	`false`
`pool_size`	a positive integer value	Implementation defined
`sync_hint`	• `contended` • `uncontended` • `private` • `serialized`	`contended`

The `access` trait specifies the accessibility of the allocated memory. The following are values you can specify for `access`:

- `all`

  This value indicates that the allocated memory must be accessible by all threads in the device where the memory allocation occurs.

  This is the default setting.
- `cgroup`

  This value indicates that the allocated memory must be accessible by all threads of the same contention group as the thread that requested the allocation. Accessing the allocated memory thread that is not part of the same contention group results in undefined behavior.
- `pteam`

This value indicates that the allocated memory is accessible by all threads that bind to the same parallel region as the thread that requests the allocations. Access to the memory by a thread that does not bind to the same parallel region as the thread that allocated the memory results in undefined behavior.

- `thread`

  This value indicates that the memory allocated is accessible only by the thread that allocated it. Attempts to allocate the memory by another thread result in undefined behavior.

The `alignment` trait specifies how allocated variables will be aligned. Variables will be byte-aligned to at least the value specified for this trait. The default setting is 1 byte. Alignment can also be affected by directives and OpenMP runtime allocator routines that specify alignment requirements.

The `fallback` trait indicates how an allocator behaves if it is unable to satisfy an allocation request. The following are values you can specify for `fallback`:

- `abort_fb`

  This value indicates that the program terminates if the allocation request fails.
- `allocator_fb`

  If this value is specified and the allocation request fails, the allocation will be tried by the allocator specified by the `fb_data` trait.
- `default_mem_fb`

  This value indicates that a failed allocation request will be retried in the `omp_default_mem_space` memory space. All traits for the `omp_default_mem_space` allocator should be set to the default trait values, except the `fallback` trait should be set to `null_fb`. This is the default setting.
- `null_fb`

  This value indicates the allocator returns a zero value when an allocation request fails.

The `fb_data` trait lets you specify a fall back allocator to be used if the requested allocator fails to satisfy the allocation request. The `fallback` trait of the failing allocator must be set to `allocator_fb` in order for the allocator specified by the `fb_data` trait to be used.

The `partition` trait describes the partitioning of allocated memory over the storage resources represented by the memory space of the allocator. The following are values you can specify for `partition`:

- `blocked`

  This value indicates the allocated memory is partitioned into blocks of memory of approximately equal size with one block per storage resource.
- `environment`

  This value indicates the allocated memory placement is determined by the runtime execution environment. This is the default setting.
- `interleaved`

  This value indicates the allocated memory is distributed in a round-robin fashion across the storage resources.
- `nearest`

  This value indicates that the allocated memory will be placed in the storage resource nearest to the thread that requested the allocation.

If the `pinned` trait has the value `true`, the allocator ensures each allocation made by the allocator will remain in the storage resource at the same location where it was allocated until it is deallocated. The default setting is `false`.

The value of `pool_size` is the total number of bytes of storage available to an allocator when there have been no allocations. The following affect `pool_size`:

- If the `access` trait has the value all, the value of `pool_size` is the limit for all allocations for all threads having access to the allocator.
- If the `access` trait of the allocator has the value `cgroup`, the value of `pool_size` is the limit for allocations made from the threads within the same contention group.

- For allocators with the `access` access trait value of `pteam`, the value of `pool_size` is the limit for allocations made within the same parallel team.
- If the `access` trait has the value `thread`, the value of `pool_size` is the limit for allocations made from each thread using the allocator.
- An allocation request for more space than the value of `pool_size` results in the allocator not fulfilling the allocation request.

The `sync_hint` trait describes the way that multiple threads can access an allocator. The following are values you can specify for `sync_hint`:

- `contended` or `uncontended`

  Value `contended` indicates that many threads are anticipated to make simultaneous allocation requests while the value `uncontended` indicates that few threads are anticipated to make simultaneous allocation. The default setting is `contended`.
- `private`

  This value indicates that all allocation requests will come from the same thread. Specifying `private` when this is not the case and two or more threads make allocation requests by the same allocator results in undefined behavior.
- `serialized`

  This value indicates that only one thread will request an allocation at a given time. The behavior is undefined if two threads request an allocation simultaneously by an allocator whose `sync_hint` value is serialized.

There are five predefined memory spaces in OpenMP:

- The system default memory is referred to as `omp_default_mem_space`.
- Large capacity memory is referred to as `omp_large_cap_mem_space`.
- High bandwidth memory is referred to as `omp_high_bw_mem_space`.
- Low latency memory is referred to as `omp_low_lat_mem_space`.
- Memory designed for optimal storage of constant values is referred to as `omp_const_mem_space`.

  It can be initialized with compile-time constant expressions or by using a firstprivate clause.

  Writing to variables in `omp_const_mem_space` results in undefined behavior.

There are three additional predefined memory spaces that are extensions to the OpenMP standard:

- `omp_target_host_mem_space` is host memory that is accessible by the device.
- `omp_target_shared_mem_space` is memory that can migrate between the host and the device.
- `omp_target_device_mem_space` is memory that is accessible to the device.

The following table shows the predefined memory allocators, the memory space they are associated with, and the non-default memory trait values they possess.

Allocator Name	**Associated Memory Space**	**Non-Default Trait Values**
`omp_default_mem_alloc`	`omp_default_mem_space`	fallback=null_fb
`omp_large_cap_mem_alloc`	`omp_large_cap_mem_space`	none
`omp_low_lat_mem_alloc`	`omp_low_lat_mem_space`	none
`omp_high_bw_mem_alloc`	`omp_high_bw_mem_space`	none
`omp_const_mem_alloc`	`omp_const_mem_space`	none
`omp_cgroup_mem_alloc`	implementation/system defined	`access=cgroup`
`omp_pteam_mem_alloc`	implementation/system defined	`access=pteam`

Allocator Name	Associated Memory Space	Non-Default Trait Values
omp_thread_mem_alloc	implementation/system defined	access=thread
omp_target_host_mem_alloc	omp_target_host_mem_space	none
omp_target_shared_mem_alloc	omp_target_shared_mem_space	none
omp_target_device_mem_alloc	omp_target_device_mem_space	none

See Also

OpenMP* Run-time Library Routines

OpenMP* Advanced Issues

This topic discusses how to use the OpenMP* library functions and environment variables and discusses some guidelines for enhancing performance with OpenMP.

OpenMP provides specific function calls, and environment variables. See the following topics to refresh your memory about the primary functions and environment variable used in this topic:

- OpenMP Run-time Library Routines
- OpenMP Environment Variables

To use the function calls, include the `omp.h` header file. This file is installed in the INCLUDE directory during the compiler installation and compile the application using the `/Qopenmp` (Windows*) or `-qopenmp` (Linux*) option.

The following example demonstrates how to use the OpenMP functions to print the alphabet and illustrates several important concepts:

1. When using functions instead of pragmas, your code must be rewritten; rewrites can mean extra debugging, testing, and maintenance efforts.
2. It becomes difficult to compile without OpenMP support.
3. It is very easy to introduce simple bugs, as in the loop (shown in example) that fails to print all the letters of the alphabet when the number of threads is not a multiple of 26.
4. You lose the ability to adjust loop scheduling without creating your own work-queue algorithm, which is a lot of extra effort. You are limited by your own scheduling, which is mostly likely static scheduling as shown in the example.

```
#include <stdio.h>
#include <omp.h>

int main(void) {
    int i;
    omp_set_num_threads(4);

    #pragma omp parallel private(i)
    {
        // OMP_NUM_THREADS is not a multiple of 26,
        // which can be considered a bug in this code.
        int LettersPerThread = 26 / omp_get_num_threads();
        int ThisThreadNum = omp_get_thread_num();
        int StartLetter = 'a'+ThisThreadNum*LettersPerThread;
        int EndLetter = 'a'+ThisThreadNum*LettersPerThread+LettersPerThread;
```

```
        for (i=StartLetter; i<EndLetter; i++) { printf("%c", i); }
    }
    printf("\n");
    return 0;
}
```

Debugging threaded applications is a complex process because debuggers change the run-time performance, which can mask race conditions. Even print statements can mask issues, because they use synchronization and operating system functions. OpenMP itself also adds some complications, because it introduces additional structure by distinguishing private variables and shared variables and inserts additional code. A debugger that supports OpenMP can help you to examine variables and step through threaded code. You can use Intel® Inspector to detect many hard-to-find threading errors analytically. Sometimes, a process of elimination can help identify problems without resorting to sophisticated debugging tools.

Remember that most mistakes are race conditions. Most race conditions are caused by shared variables that really should have been declared private. Start by looking at the variables inside the parallel regions and make sure that the variables are declared private when necessary. Next, check functions called within parallel constructs. By default, variables declared on the stack are private, but the C/C++ keyword `static` changes the variable to be placed on the global heap and therefore shared for OpenMP loops.

The `default(none)` clause can be used to help find those hard-to-spot variables. If you specify `default(none)`, then every variable must be declared with a data-sharing attribute clause. For example:

```
#pragma omp parallel for default(none) private(x,y) shared(a,b)
```

Another common mistake is using uninitialized variables. Remember that private variables do not have initial values upon entering a parallel construct. Use the `firstprivate` and `lastprivate` clauses to initialize them only when necessary, because doing so adds extra overhead.

If you still can't find the bug, then consider the possibility of reducing the scope. Try a binary-hunt. Force parallel sections to be serial again with `if(0)` on the parallel construct or commenting out the pragma altogether. Another method is to force large chunks of a parallel region to be critical sections. Pick a region of the code that you think contains the bug and place it within a critical section. Try to find the section of code that suddenly works when it is within a critical section and fails when it is not. Now look at the variables, and see if the bug is apparent. If that still doesn't work, try setting the entire program to run in serial by setting the compiler-specific environment variable `KMP_LIBRARY=serial`.

If the code is still not working, and you are not using any OpenMP API function calls, compile it without the `/Qopenmp` (Windows) or `-qopenmp` (Linux) option to make sure the serial version works. If you are using OpenMP API function calls, use the `/Qopenmp-stubs` (Windows) or `-qopenmp-stubs` (Linux) option.

Performance

OpenMP threaded application performance is largely dependent upon the following things:

- The underlying performance of the single-threaded code.
- CPU utilization, idle threads, and load balancing.
- The percentage of the application that is executed in parallel by multiple threads.
- The amount of synchronization and communication among the threads.
- The overhead needed to create, manage, destroy, and synchronize the threads, made worse by the number of single-to-parallel or parallel-to-single transitions called fork-join transitions.
- Performance limitations of shared resources such as memory, bus bandwidth, and CPU execution units.
- Memory conflicts caused by shared memory or falsely shared memory.

Performance always begins with a properly constructed parallel algorithm or application. For example, parallelizing a bubble-sort, even one written in hand-optimized assembly language, is not a good place to start. Keep scalability in mind; creating a program that runs well on two CPUs is not as efficient as creating one that runs well on *n* CPUs. With OpenMP, the number of threads is chosen by the compiler, so programs that work well regardless of the number of threads are highly desirable. Producer/consumer architectures are rarely efficient, because they are made specifically for two threads.

Once the algorithm is in place, make sure that the code runs efficiently on the targeted Intel® architecture; a single-threaded version can be a big help. Turn off the `/Qopenmp` (Windows) or `-qopenmp` (Linux) option to generate a single-threaded version, or build with the `/Qopenmp-stubs` (Windows) or `-qopenmp-stubs` (Linux) option, and run the single-threaded version through the usual set of optimizations.

Once you have gotten the single-threaded performance, it is time to generate the multi-threaded version and start doing some analysis.

Optimizations are really a combination of patience, experimentation, and practice. Make little test programs that mimic the way your application uses the computer resources to get a feel for what things are faster than others. Be sure to try the different scheduling clauses for the parallel sections of code. If the overhead of a parallel region is large compared to the compute time, you may want to use an `if` clause to execute the section serially.

See Also

OpenMP* Run-time Library Routines
Worksharing Using OpenMP*
qopenmp, Qopenmp
qopenmp-stubs, Qopenmp-stubs

OpenMP* Implementation-Defined Behaviors

This topic summarizes the behaviors that are described as implementation defined in the OpenMP* API specification.

NOTE
Internal Control Variables (ICVs) mentioned below are discussed in the OpenMP API specification.

Name	Description
`single` construct	The first thread that encounters the single construct executes the structured block.
`teams` construct	The number of teams that are created is equal to 1 if you don't specify the `num_teams` clause.
`dist_schedule` clause, `distribute` construct	If you don't specify the `dist_schedule` clause, then the schedule for the `distribute` construct is `static`.
`omp_set_num_threads` routine	If the argument is not a positive integer, then Intel's OpenMP implementation sets the value of the first element of the `nthreads-var` ICV of the current task to 1.
`omp_set_max_active_levels` routine	If the argument is a negative integer this call is ignored and the last valid setting is used.
`omp_get_max_active_levels` routine	When called from within any explicit parallel region the binding thread set, and binding region, if required, for the `omp_get_max_active_levels` region is the current task region.
`OMP_SCHEDULE` environment variable	If the value of the variable does not conform to the specified format then the value of the `run-sched-var` ICV is set to `static`.

Name	Description
`OMP_NUM_THREADS` environment variable	If any value of the list specified in the environment variable is negative then the whole list is ignored. If any value of the list is zero then this value is set to 1.
`OMP_PROC_BIND` environment variable	If the value is not `true`, `false`, or a comma separated list of `master` (deprecated), `primary`, `close`, or `spread`, then Intel's OpenMP implementation sets the value of `bind-var` ICV to false.
`OMP_DYNAMIC` environment variable	If the value is neither `true` nor `false`, then the implementation sets the value of `dyn-var` ICV to false.
`OMP_NESTED` environment variable	If the value is neither `true` nor `false`, then the implementation sets the value of `nest-var` ICV to false.
`OMP_STACKSIZE` environment variable	If the value does not conform to the specified format or the implementation cannot provide a stack of the specified size, then Intel's OpenMP implementation sets the value of `stacksize-var` ICV to the default size, which is specified as being from 1MB to 4MB depending on the architecture. On Linux* or macOS*, the implementation can set the value of `stacksize-var` ICV up to 256MB, respecting the operating system's stack size limit.
`OMP_MAX_ACTIVE_LEVELS` environment variable	If the value is a negative integer or is greater than the number of parallel levels an implementation can support, then Intel's OpenMP implementation sets the value of the `max-active-levels-var` ICV to 1.
`OMP_THREAD_LIMIT` environment variable	If the requested value is greater than the number of threads an implementation can support, or if the value is a negative integer, then Intel's OpenMP implementation sets the value of the `thread-limit-var` ICV to the maximum number of threads supported on a particular platform. If the requested value is zero then the implementation sets the value of the `thread-limit-var` ICV to 1.
Runtime library definitions	Intel's OpenMP implementation provides both the include file `omp.h` and `omp-tools.h` .

OpenMP* Examples

The following examples show how to use several OpenMP* features.

A Simple Difference Operator

This example shows a simple parallel loop where the amount of work in each iteration is different. Dynamic scheduling is used to improve load balancing.

The `for` pragma has a `nowait` clause because there is an implicit barrier at the end of the parallel region. Therefore it is not necessary to also have a barrier at the end of the `for` region.

```
void for1(float a[], float b[], int n) {
  int i, j;
  #pragma omp parallel shared(a,b,n) {
   #pragma omp for schedule(dynamic,1) private (i,j) nowait
    for (i = 1; i < n; i++)
       for (j = 0; j < i; j++)
         b[j + n*i] = (a[j + n*i] + a[j + n*(i-1)]) / 2.0;
  }
}
```

Two Difference Operators: for Loop Version

This example uses two parallel loops fused to reduce fork/join overhead. The first `for` pragma has a `nowait` clause because all the data used in the second loop is different than all the data used in the first loop.

```
void for2(float a[], float b[], float c[], float d[], int n, int m) {
  int i, j;
  #pragma omp parallel shared(a,b,c,d,n,m) private(i,j) {
    #pragma omp for schedule(dynamic,1) nowait
    for (i = 1; i < n; i++)
      for (j = 0; j < i; j++)
        b[j + n*i] = ( a[j + n*i] + a[j + n*(i-1)] )/2.0;
    #pragma omp for schedule(dynamic,1) nowait
    for (i = 1; i < m; i++)
      for (j = 0; j < i; j++)
        d[j + m*i] = ( c[j + m*i] + c[j + m*(i-1)] )/2.0;
  }
}
```

Two Difference Operators: sections Version

This example demonstrates the use of the `sections` pragma . The logic is identical to the preceding `for` pragma example, but uses a `sections` pragma instead of a `for` pragma . Here the speedup is limited to two because there are only two units of work whereas in the example above there are `(n-1) + (m-1)` units of work.

```
void sections1(float a[], float b[], float c[], float d[], int n, int m) {
  int i, j;
  #pragma omp parallel shared(a,b,c,d,n,m) private(i,j) {
    #pragma omp sections nowait {
      #pragma omp section
       for (i = 1; i < n; i++)
         for (j = 0; j < i; j++)
           b[j + n*i] = ( a[j + n*i] + a[j + n*(i-1)] )/2.0;
      #pragma omp section
       for (i = 1; i < m; i++)
         for (j = 0; j < i; j++)
           d[j + m*i] = ( c[j + m*i] + c[j + m*(i-1)] )/2.0;
     }
   }
}
```

Update a Shared Scalar

This example demonstrates how to use a `single` construct to update an element of the shared array `a`. The optional `nowait` clause after the first loop is omitted because it is necessary to wait at the end of the loop before proceeding into the `single` construct to avoid a race condition.

```
void sp_1a(float a[], float b[], int n) {
  int i;
  #pragma omp parallel shared(a,b,n) private(i) {
    #pragma omp for
      for (i = 0; i < n; i++)
        a[i] = 1.0 / a[i];
      #pragma omp single
        a[0] = MIN( a[0], 1.0 );
      #pragma omp for nowait
      for (i = 0; i < n; i++)
      b[i] = b[i] / a[i];
   }
}
```

Intel® oneAPI Level Zero

The objective of the Intel® oneAPI Level Zero (Level Zero) Application Programming Interface (API) is to provide direct-to-metal interfaces to offload accelerator devices. Its programming interface can be tailored to any device needs and can be adapted to support broader set of languages features such as function pointers, virtual functions, unified memory, and I/O capabilities.

Most applications should not require the additional control provided by the Level Zero API. The Level Zero API is intended for providing explicit controls needed by higher-level runtime APIs and libraries.

While initially influenced by other low-level APIs, such as OpenCL™ API and Vulkan*, the Level Zero APIs are designed to evolve independently. While initially influenced by graphics processing unit architecture, the Level Zero APIs are designed to be supportable across different compute device architectures, such as Field Programmable Gate Arrays (FPGAs) and other types of accelerator architectures.

Intel® oneAPI Level Zero Switch

Data Parallel C++ (DPC++) is just one of the many components of the oneAPI project. The Intel® oneAPI Level Zero (Level Zero) API provides low-level direct-to-metal interfaces that are tailored to the devices on a oneAPI project. While heavily influenced by other low-level APIs, such as OpenCL™ API, Level Zero is designed to evolve independently.

More information on Level Zero is available in the oneAPI Specification.

Packages to Install

The packages you must install are `intel-level-zero-gpu` and `level-zero`.

Level Zero Loader

Level Zero is supportable across different oneAPI compute device architectures. The Level Zero loader discovers all Level Zero drivers in the system. In addition, the Level Zero loader is also the Level Zero software development kit: It carries the Level Zero headers and libraries where you build Level Zero programs.

Level Zero GPU Driver

The driver is open-source and regular public releases are maintained. It does not come with DPC++ and must be installed independently. The Level Zero driver and OpenCL™ driver come in the same package. More info about the Level Zero driver is available at GitHub.

DPC++ Plugins

SYCL targets a variety of devices: CPU, GPU, and Field Programmable Gate Array (FPGA). Different devices can be operated through different low-level drivers, such as OpenCL for FPGA. The Plugin Interface (PI) is a unified DPC++ API for working with different devices in a unified way. DPC++ plugins implement specific translations of the PI API into low-level runtime. The Level Zero PI Plugin was created in DPC++ to enable devices supported through the Level Zero system.

Scenario	Information
SYCL Device Selection	The PI for DPC++ performs device discovery of all available devices through all available PI plugins. The same physical hardware device can be seen as multiple different SYCL devices if multiple plugins support it (for example, OpenCL Gen90 and Level Zero Gen90). The SYCL runtime performs device selection from the available devices based on device selectors. The device selectors can be user-defined or built in (for example, `gpu_selector`).
Discovery of Multiple PI Plugins	The implication of support for the discovery of multiple plugins is that the same GPU card can be seen as multiple different GPU devices available under different PI plugins. **NOTE** Corresponding runtimes (OpenCL and/or Level Zero) must be installed correctly and independently for PI to see their devices. The SYCL specification does not define which device will be used if there are multiple devices that match criteria (for example, `is_gpu()`).
Default Preference is Given to a Level Zero GPU on Linux	By default, if no special action is taken and the Level Zero runtime reports support for the installed GPU, then the SYCL runtime uses the installed GPU. This is true for standard built-in device selectors and custom device selectors, where no action is taken to change the default behavior. Currently, on Windows, the preference is given to an OpenCL GPU. Devices that are not supported with the Level Zero runtime (CPU/FPGA) continue to run with OpenCL.
How to Change the Default Preference	Use the `SYCL_BE` environment variable to change the default preference. The valid values are `PI_OPENCL` and `PI_LEVEL0`.

Scenario	Information
	For example, if you specify `SYCL_BE=PI_OPENCL` and the PI OpenCL plugin reports the availability of the device of the required type, then that device is used. It overrides the default preference that is given to the Level Zero GPU, if the GPU is supported by the installed version of OpenCL. **NOTE** The `SYCL_BE` setting only works when there are multiple choices. **Recommendation** If your code does not work, try running it with `SYCL_BE=PI_OPENCL` to see if the problem is related to Level Zero.
How to See Where the Code is Running	Use the `SYCL_PI_TRACE=1` environment variable to see where your code is running. It reports the choice made by the built-in device selectors, if they are used. Use `SYCL_PI_TRACE=-1` to enable verbose tracing of the PI and show all the devices detected by the PI discovery process.
How to Find all DPC++ Plugins and Supported Devices Discovered in the System	Use the `sycl-ls` utility to find all the DPC++ plugins on your system. `sycl-ls` queries all the platforms and devices available through the plugins, and prints useful information about SYCL devices and their ID numbers. This information is useful when you want to designate a specific device to run a SYCL program. The `SYCL_DEVICE_FILTER` string is printed at each line to show three information pieces: • `The backend that the plugin supports` • `The device_type` • `The device_id` Verbose output is available with `$ sycl-ls --verbose`, which gives you the same choices that are made by standard built-in device selectors and other custom device selectors.

SYCL_DEVICE_FILTER

The `SYCL_DEVICE_FILTER` environment variable limits the SYCL runtime to use a subset of the system's devices. Setting this environment variable affects all of the device query functions (`platform::get_devices() and platform::get_platforms()`) and all of the device selectors.

The value of this environment variable is a comma separated list of filters, where each filter is a triple of the form `backend:device_type:device_num`. Each element of the triple is optional, but each filter must have at least one value. The possible values of the backend are:

- `host`
- `level_zero`
- `opencl`
- `cuda`
- `hip`
- `*`

The possible values of the `device_type` are:

- `host`
- `cpu`
- `gpu`
- `acc`
- `*`

The `device_num` is an integer that indexes the enumeration of devices from the `sycl-ls` utility tool, where the first device in that enumeration has index zero in each backend. For example, `SYCL_DEVICE_FILTER=2` returns all devices with index '2' from all different backends. If multiple devices satisfy this device number (GPU and CPU devices can be assigned device number '2'), then the `default_selector` chooses the device with the highest heuristic point. When `SYCL_DEVICE_ALLOWLIST` is set, it is applied before counting devices and affects `device_num` values.

If a filter has all three elements of the triple, it selects only those devices that come from the given backend, have the specified device type, and have the given device index. If more than one filter is specified, the runtime is restricted to the union of devices selected by all filters. The runtime does not include the host backend and the host device automatically, unless one of the filters explicitly specifies the host device type. `SYCL_DEVICE_FILTER=host` should be set your program uses the host device only.

NOTE
All device selectors throw an exception if the filtered list of devices does not include a device that satisfies the selector. For instance, `SYCL_DEVICE_FILTER=cpu,level_zero` causes `host_selector()` to throw an exception. `SYCL_DEVICE_FILTER` also limits loading only specified plugins into the SYCL runtime.

`SYCL_DEVICE_FILTER=level_zero` causes the `cpu_selector` to throw an exception since the SYCL runtime only loads the `level_zero` backend, which does not support any CPU devices at this time. When multiple devices satisfy the filter (example: `SYCL_DEVICE_FILTER=gpu`), only one of them is selected.

Intel® oneAPI Level Zero Backend Specification

The Intel® oneAPI Level Zero (Level Zero) extension introduces a Level Zero backend for SYCL. It is built on top of Level Zero runtime enabled with the oneAPI Level Zero Specification. The Level Zero backend aims to provide the best possible performance of SYCL application on a variety of targets supported. The currently supported targets are all Intel GPUs starting with Gen9.

This extension provides a feature-test macro as described in the SYCL spec's section, Feature Test Macros. Any implementation supporting this extension must predefine the macro `SYCL_EXT_ONEAPI_BACKEND_LEVEL_ZERO` to one of the values defined in the table below. Applications can test for the existence of this macro to see if the implementation supports this feature, or they can test the macro's value to see the extension APIs the implementation supports:

Value	Description
1	Initial extension version.

Value	Description
2	Added support for the make_buffer() API.
3	Added device member to backend_input_t<backend::ext_oneapi_level_zero, queue>.

NOTE This extension is following SYCL 2020 backend specification. Prior APIs for interoperability with Level Zero are marked as deprecated and will be removed in the next release.

Prerequisites

The Level Zero loader and drivers must be installed on your system for the SYCL runtime to recognize and enable the Level Zero backend. Visit Intel® oneAPI DPC++/C++ Compiler System Requirements for specific instructions.

User-visible Level Zero Backend Selection and Default Backend

The Level Zero backend is added to the sycl::backend enumeration with:

```
enum class backend {
  // ...
  ext_oneapi_level_zero,
  // ...
};
```

The sections below explain the different ways the Level Zero backend can be selected.

Through an Environment Variable

The SYCL_DEVICE_FILTER environment variable limits the SYCL runtime to use only a subset of the system's devices. By using level_zero for the backend in SYCL_DEVICE_FILTER, you can select the use of Level Zero as a SYCL backend. For more information, see the Environment Variables.

Through a Programming API

The Filter Selector extension is described in SYCL Proposals: Filter Selector. Similar to how the SYCL_DEVICE_FILTER applies filtering to the entire process, this device selector can be used to select the Level Zero backend.

When neither the environment variable nor the filtering device selector is used, the implementation chooses the Level Zero backend for GPU devices supported by the installed Level Zero runtime. The serving backend for a SYCL platform can be queried with the get_backend() member function sycl::platform.

Interoperability with the Level Zero API

The sections below describe the various interoperabilities that are possible between SYCL and Level Zero. The application must include the following headers to use any of the inter-operation APIs described in this section. These headers must be included in the order shown:

```
#include "level_zero/ze_api.h"
#include "sycl/ext/oneapi/backend/level_zero.hpp"
```

Mapping of SYCL Objects to Level Zero Handles

These SYCL objects encapsulate the corresponding Level Zero handles:

SYCL Type	backend_return_t <backend::ext_oneapi_level_zero, SyclType>	backend_input_t<backend::ext_oneapi_level_zero, SyclType>
platform	ze_driver_handle_t	ze_driver_handle_t
device	ze_device_handle_t	ze_device_handle_t
context	ze_context_handle_t	struct { ze_context_handle_t NativeHandle; std::vector<device> DeviceList; ext::oneapi::level_zero::ownership Ownership{ ext::oneapi::level_zero::ownership::transfer}; }
queue	ze_command_queue_handle_t	struct { ze_command_queue_handle_t NativeHandle; ext::oneapi::level_zero::ownership Ownership{ ext::oneapi::level_zero::ownership::transfer}; } Deprecated in Version 3 of the Level Zero Backend Specification. struct { ze_command_queue_handle_t NativeHandle; device Device; ext::oneapi::level_zero::ownership Ownership{ ext::oneapi::level_zero::ownership::transfer}; } Supported since Version 3 of the Level Zero Backend Specification.
event	ze_event_handle_t	struct { ze_event_handle_t NativeHandle; ext::oneapi::level_zero::ownership Ownership{ ext::oneapi::level_zero::ownership::transfer}; }
kernel_bundle	std::vector<ze_module_handle_t>	struct { ze_module_handle_t NativeHandle; ext::oneapi::level_zero::ownership Ownership{

SYCL Type	backend_return_t <backend::ext_oneapi_level_zero, SyclType>	backend_input_t<backend::ext_oneapi_level_zero, SyclType>
		ext::oneapi::level_zero::ownership::transfer}; }
kernel	ze_kernel_handle_t	struct { kernel_bundle<bundle_state::executable> KernelBundle; ze_kernel_handle_t NativeHandle; ext::oneapi::level_zero::ownership Ownership{ ext::oneapi::level_zero::ownership::transfer}; }
buffer	void *	struct { void *NativeHandle; ext::oneapi::level_zero::ownership Ownership{ ext::oneapi::level_zero::ownership::transfer}; }

Obtaining Built-in Level Zero Handles from SYCL Objects

The sycl::get_native<backend::ext_oneapi_level_zero> free-function is how you can use a raw built-in Level Zero handle to obtain a specific SYCL object. The function is supported for the SYCL platform, device, context, queue, event and program classes. You can use a free-function defined in the cl::sycl namespace instead of the member function with:

```
template <backend BackendName, class SyclObjectT>
auto get_native(const SyclObjectT &Obj)
    -> backend_return_t<BackendName, SyclObjectT>
```

This function is supported for SYCL platform, device, context, queue, event, kernel_bundle, and kernel classes.

The sycl::get_native<backend::ext_oneapi_level_zero> free-function is not supported for the SYCL buffer class. The built-in backend object associated with the buffer can be obtained using the interop_hande class as described in the SYCL spec's section, Class interop_handle. The pointer is returned by get_native_mem<backend::ext_oneapi_level_zero> method of the interop_handle class, which is the value returned from a call to zeMemAllocShared(), zeMemAllocDevice(), or zeMemAllocHost() and not directly accessible from the host. You may need to copy your data to the host to access the data. You can get information on the type of the allocation using the type data member of the ze_memory_allocation_properties_t struct that is returned by zeMemGetAllocProperties.

```
Queue.submit([&](handler &CGH) {
    auto BufferAcc = Buffer.get_access<access::mode::write>(CGH);
    CGH.host_task([=](const interop_handle &IH) {
        void *DevicePtr =
            IH.get_native_mem<backend::ext_oneapi_level_zero>(BufferAcc);
        ze_memory_allocation_properties_t MemAllocProperties{};
```

```
        ze_result_t Res = zeMemGetAllocProperties(
            ZeContext, DevicePtr, &MemAllocProperties, nullptr);
        ze_memory_type_t ZeMemType = MemAllocProperties.type;
    });
}).wait();
```

Construct a SYCL Object from a Level Zero Handle

The following free functions, defined in the `sycl` namespace are specialized for the Level Zero backend to allow an application to create a SYCL object that encapsulates a corresponding Level Zero object, see the table below for specific functions.

Level Zero Interoperability Function	Description
`make_platform<backend::ext_oneapi_level_zero>( const backend_input_t< backend::ext_oneapi_level_zero, platform> &)`	Constructs a SYCL platform instance from a Level Zero `ze_driver_handle_t`. The SYCL execution environment contains a fixed number of platforms that are counted with `sycl::platform::get_platforms()`. Calling this function does not create a platform, it creates a `sycl::platform` object that is a copy of one of the platforms from that enumeration.
`make_device<backend::ext_oneapi_level_zero>( const backend_input_t< backend::ext_oneapi_level_zero, device> &)`	Constructs a SYCL device instance from a Level Zero `ze_device_handle_t`. The SYCL execution environment for the Level Zero backend contains a fixed number of devices that are counted with `sycl::device::get_devices()` and a fixed number of sub-devices that are counted with `sycl::device::create_sub_devices(...)`. Calling this function does not create a device, it creates a `sycl::device` object that is a copy of one of the devices from those enumerations.
`make_context<backend::ext_oneapi_level_zero>( const backend_input_t< backend::ext_oneapi_level_zero, context> &)`	Constructs a SYCL context instance from a Level Zero `ze_context_handle_t`. The context is created against the devices passed in a `DeviceList` structure member. There must be at least one device given and all the devices must be from the same SYCL platform and from the same Level Zero driver. The `Ownership` input structure member specifies if the SYCL runtime should take ownership of the passed built-in handle. The default behavior is to transfer the ownership to the SYCL runtime. See section Level Zero Handle Ownership and Thread-safety for details.
`make_queue<backend::ext_oneapi_level_zero>( const backend_input_t< backend::ext_oneapi_level_zero, queue> &, const context &Context)`	Constructs a SYCL queue instance from a Level Zero `ze_command_queue_handle_t`. The `Context` argument must be a valid SYCL context encapsulating a Level Zero context. The `Device` input structure member specifies the device to create the `queue` against and must be in `Context`. The `Ownership` input structure member specifies if the SYCL runtime should take ownership of the passed built-in handle. The default behavior is to

Level Zero Interoperability Function	Description
	transfer the ownership to the SYCL runtime. See Level Zero Handle Ownership and Thread-safety for details. If the deprecated variant of `backend_input_t<backend::ext_oneapi_level_zero, queue>` is passed to `make_queue`, the queue is attached to the first device in `Context`.
`make_event<backend::ext_oneapi_level_zero>( const backend_input_t< backend::ext_oneapi_level_zero, event> &, const context &Context)`	Constructs a SYCL event instance from a Level Zero `ze_event_handle_t`. The `Context` argument must be a valid SYCL context encapsulating a Level Zero context. The Level Zero event should be allocated from an event pool created in the same context. The `Ownership` input structure member specifies if the SYCL runtime should take ownership of the passed built-in handle. The default behavior is to transfer the ownership to the SYCL runtime. See Level Zero Handle Ownership and Thread-safety for details.
`make_kernel_bundle<backend::ext_oneapi_level_zero, bundle_state::executable>( const backend_input_t< backend::ext_oneapi_level_zero, kernel_bundle<bundle_state::executable>> &, const context &Context)`	Constructs a SYCL `kernel_bundle` instance from a Level Zero `ze_module_handle_t`. The `Context` argument must be a valid SYCL context encapsulating a Level Zero context, and the Level Zero module must be created on the same context. The Level Zero module must be fully linked (it cannot require further linking through `zeModuleDynamicLink`). The SYCL `kernel_bundle` is created in the executable state. The `Ownership` input structure member specifies if the SYCL runtime should take ownership of the passed built-in handle. The default behavior is to transfer the ownership to the SYCL runtime. See Level Zero Handle Ownership and Thread-safety for details. If the behavior is `transfer`, then the runtime is going to destroy the input Level Zero module, and the application must not have any outstanding `ze_kernel_handle_t` handles to the underlying `ze_module_handle_t` by the time this interoperability `kernel_bundle` destructor is called.
`make_kernel<backend::ext_oneapi_level_zero>( const backend_input_t< backend::ext_oneapi_level_zero, kernel> &, const context &Context)`	Constructs a SYCL kernel instance from a Level Zero `ze_kernel_handle_t`. The `KernelBundle` input structure specifies the `kernel_bundle` corresponding to the Level Zero module from which the kernel is created. There must be exactly one Level Zero module in the `KernelBundle`. The `Context` argument must be a valid SYCL context encapsulating a Level Zero context, and the Level Zero module must be created on the same context. The `Ownership` input structure member specifies if

Level Zero Interoperability Function	Description
	the SYCL runtime should take ownership of the passed built-in handle. The default behavior is to transfer the ownership to the SYCL runtime. See Level Zero Handle Ownership and Thread-safety for details. If the behavior is `transfer`, then the runtime is going to destroy the input Level Zero kernel.
`make_buffer( const backend_input_t<backend::ext_oneapi_level_zero, buffer<T, Dimensions, AllocatorT>> &, const context &Context)`	This API is available starting with revision 2 of the Level Zero Backend Specification. Construct a SYCL buffer instance from a pointer to a Level Zero memory allocation. The pointer must be the value returned from a previous call to `zeMemAllocShared()`, `zeMemAllocDevice()`, or `zeMemAllocHost()`. The input SYCL context `Context` must be associated with a single device, matching the device used at the prior allocation. The `Context` argument must be a valid SYCL context encapsulating a Level Zero context, and the Level Zero memory must be allocated on the same context. Created SYCL buffer can be accessed in another contexts, not only in the provided input context. The `Ownership` input structure member specifies if the SYCL runtime should take ownership of the passed built-in handle. The default behavior is to transfer the ownership to the SYCL runtime. See Level Zero Handle Ownership and Thread-safety for details. If the behavior is `transfer`, then the runtime is going to free the input Level Zero memory allocation. Synchronization rules for a buffer that is created with this API are described in Interoperability Buffer Synchronization Rules.
`make_buffer( const backend_input_t<backend::ext_oneapi_level_zero, buffer<T, Dimensions, AllocatorT>> &, const context &Context, event AvailableEvent)`	This API is available starting with revision 2 of the Level Zero Backend Specification. Construct a SYCL buffer instance from a pointer to a Level Zero memory allocation. Refer to `make_buffer` description above for semantics and restrictions. The additional `AvailableEvent` argument must be a valid SYCL event. The instance of the SYCL buffer class template being constructed must wait for the SYCL event parameter to signal that the memory built-in handle is ready to be used.

Level Zero Handle Ownership and Thread-safety

The Level Zero runtime does not do reference-counting of its objects, so it is crucial to adhere to these practices of how Level Zero handles are managed. By default, the ownership is transferred to the SYCL runtime, but some interoperability API supports overriding this behavior and keeps the ownership in the application. Use this enumeration for explicit specification of the ownership:

```
namespace sycl {
namespace ext {
namespace oneapi {
namespace level_zero {

enum class ownership { transfer, keep };

} // namespace level_zero
} // namespace oneapi
} // namespace ext
} // namespace sycl
```

- **SYCL Runtime Takes Ownership (default)**: Whenever the application creates a SYCL object from the corresponding Level Zero handle, with one of the `make_*` functions, the SYCL runtime takes ownership of the Level Zero handle if no explicit `ownership::keep` was specified. The application must not use the Level Zero handle after the last host copy of the SYCL object is destroyed. The application must not destroy the Level Zero handle. For more information, see the SYCL Common Reference Semantics section.
- **Application Keeps Ownership (explicit)**: If a SYCL object is created with an interoperability API explicitly asking to keep the built-in handle ownership in the application with `ownership::keep`, then the SYCL runtime does not take the ownership and will not destroy the Level Zero handle at the destruction of the SYCL object. The application is responsible for destroying the built-in handle when it no longer needs it, but it must not destroy the handle before the last host copy of the SYCL object is destroyed (as described in the core SYCL specification under SYCL Common Reference Semantics.
- **Obtaining Built-in Handle Does Not Change Ownership**: The application may call the `get_native<backend::ext_oneapi_level_zero>` free function on a SYCL object to retrieve the underlying Level Zero handle. Doing so does not change the ownership of the Level Zero handle. The application may not use this handle after the last host copy of the SYCL object is destroyed (as described in the core SYCL specification under SYCL Common Reference Semantics unless the SYCL object was created by the application with `ownership::keep`.
- **Considerations for Multi-threaded Environment**: The Level Zero API is not thread-safe, refer to Multithreading and Concurrency for more information. Applications must make sure that the Level Zero handles are not used simultaneously from different threads. The SYCL runtime takes ownership of the Level Zero handles and should not attempt further direct use of those handles.

Interoperability Buffer Synchronization Rules

A SYCL buffer that is constructed with this interop API uses the Level Zero memory allocation for its full lifetime. The contents of the Level Zero memory allocation are unspecified for the lifetime of the SYCL buffer. If the application modifies the contents of that Level Zero memory allocation during the lifetime of the SYCL buffer, the behavior is undefined. The initial contents of the SYCL buffer will be the initial contents of the Level Zero memory allocation at the time of the SYCL buffer's construction.

The behavior of the SYCL buffer destructor depends on the Ownership flag. As with other SYCL buffers, this behavior is triggered only when the last reference count to the buffer is dropped, as described in the SYCL spec's section, Buffer Synchronization Rules.

- If the ownership is `keep` (the application retains ownership of the Level Zero memory allocation), then the SYCL buffer destructor blocks until all work in queues on the buffer have completed. The contents of the buffer is not copied back to the Level Zero memory allocation.
- If the ownership is transfer (the SYCL runtime has ownership of the Level Zero memory allocation), then the SYCL buffer destructor does not need to block, even if work on the buffer has not completed. The SYCL runtime frees the Level Zero memory allocation asynchronously when it is no longer in use in queues.

Level Zero Additional Functionality

Device Information Descriptors

The Level Zero backend provides the following device information descriptors that an application can use to query information about a Level Zero device. Applications use these queries with the `device::get_backend_info<>()` member function as shown in the example below, which illustrates the `free_memory` query:

```
sycl::queue Queue;
auto Device = Queue.get_device();

size_t freeMemory =
  Device.get_backend_info<sycl::ext::oneapi::level_zero::info::device::free_memory>();
```

New descriptors have been added as part of this specification, and are described in the table and example below.

Descriptor	Description
`sycl::ext::oneapi::level_zero::info::device::free_memory`	Returns the number of bytes of free memory for the device.

```
namespace sycl{
namespace ext {
namespace oneapi {
namespace level_zero {
namespace info {
namespace device {

struct free_memory {
    using return_type = size_t;
};

} // namespace device;
} // namespace info
} // namespace level_zero
} // namespace oneapi
} // namespace ext
} // namespace sycl
```

Programming with the Intel® oneAPI Level Zero Backend

This page shows the supported scenarios for multi-card and multi-tile programming with the Intel® oneAPI Level Zero (Level Zero) Backend.

Device Discovery

Root-devices

In this programming model, Intel GPUs are represented as SYCL GPU devices, or root-devices. You can find your root-device with the `sycl-ls` tool. For example:

```
sycl-ls
```

Example output:

```
[opencl:gpu:0] Intel(R) OpenCL HD Graphics, Intel(R) UHD Graphics 630 [0x3e92] 3.0 [21.49.21786]
[opencl:cpu:1] Intel(R) OpenCL, Intel(R) Core(TM) i7-8700K CPU @ 3.70GHz 2.1
[2020.11.11.0.03_160000]
```

```
[ext_oneapi_level_zero:gpu:0] Intel(R) Level-Zero, Intel(R) UHD Graphics 630 [0x3e92] 1.2
[1.2.21786]
[host:host:0] SYCL host platform, SYCL host device 1.2 [1.2]
```

`sycl-ls` shows the devices and platforms of all the SYCL backends, which are seen by the SYCL runtime. The example above shows the CPU (managed by an OpenCL™ backend) and two GPUs that correspond to the single physical GPU (managed by an OpenCL™ or Level Zero backend). There are two options to filter the observable root-devices:

Option One

Use the environment variable `SYCL_DEVICE_FILTER`, which is described in the Environment Variables. For example:

```
SYCL_DEVICE_FILTER=ext_oneapi_level_zero sycl-ls
```

Example output:

```
[ext_oneapi_level_zero:gpu:0] Intel(R) Level-Zero, Intel(R) UHD Graphics 630 [0x3e92] 1.2
[1.2.21786]
```

Option Two

Use a similar API, as described in the Filter Selector, for example, the `filter_selector("ext_oneapi_level_zero")` only sees Level Zero operated devices.

If there are multiple GPUs in a system, they are seen as multiple root-devices. On Linux, you will see multiple SYCL root-devices of the same SYCL platform. On Windows, you will see root-devices of multiple different SYCL platforms.

You can use `CreateMultipleRootDevices=N NEOReadDebugKeys=1` environment variables to emulate multiple GPU cards. For example:

```
CreateMultipleRootDevices=2 NEOReadDebugKeys=1 SYCL_DEVICE_FILTER=ext_oneapi_level_zero sycl-ls
```

Example output:

```
[ext_oneapi_level_zero:gpu:0] Intel(R) Level-Zero, Intel(R) UHD Graphics 630 [0x3e92] 1.2
[1.2.21786]
[ext_oneapi_level_zero:gpu:1] Intel(R) Level-Zero, Intel(R) UHD Graphics 630 [0x3e92] 1.2
[1.2.21786]
```

NOTE`CreateMultipleRootDevices` is experimental, not validated, and is used for debug/experimental purposes only.

Sub-devices

Some Intel GPU hardware is composed of multiple tiles, where the root-devices can be partitioned into sub-devices that correspond to the physical tiles. For example:

```
try {
  vector<device> SubDevices = RootDevice.create_sub_devices<
  cl::sycl::info::partition_property::partition_by_affinity_domain>(
  cl::sycl::info::partition_affinity_domain::next_partitionable);
}
```

Each call to `create_sub_devices` returns the same sub-devices in their persistent order. Use the `ZE_AFFINITY_MASK` environment variable to control what sub-devices are exposed by the Level Zero driver. The `partition_by_affinity_domain` is the only type of partitioning supported for Intel GPUs. The `next_partitionable` and `numa` properties are the only partitioning properties supported.

The `CreateMultipleSubDevices=N NEOReadDebugKeys=1` environment variables can be used to emulate multiple tiles of a GPU.

> **NOTE**`CreateMultipleSubDevices` is experimental, not validated, and is used for debug/ experimental purposes only.

Contexts

Contexts are used for resource isolation and sharing. A SYCL context may consist of one or multiple devices. Both root-devices and sub-devices can be found within a single context, but they need to be from the same SYCL platform. A SYCL `kernel_bundle` created against a context with multiple devices is built to each of the root-devices in the context. For a context that consists of multiple sub-devices of the same root-device, only a single build (to that root-device) is needed.

Memory

Unified Shared Memory (USM)

There are multiple ways to allocate memory:

- `malloc_device`:
  - Allocation can only be accessed by the specified device, but not by other devices in the context or by the host.
  - The data always stays on the device and is the fastest available for kernel execution.
  - Explicit copy is needed for transferring data to the host or other devices in the context.
- `malloc_host`:
  - Allocation can be accessed by the host and any other device in the context.
  - The data always stays on the host and is accessed via Peripheral Component Interconnect (PCI) from the devices.
  - No explicit copy is needed for synchronizing of the data with the host or devices.
- `malloc_shared`:
  - Allocation can only be accessed by the host and the specified device.
  - The data can migrate (operated by the Level Zero driver) between the host and the device for faster access.
  - No explicit copy is necessary for synchronizing between the host and the device, but it is needed for other devices in the context.

Memory allocated against a root-device is accessible by all of its sub-devices (tiles). If you are operating on a context with multiple sub-devices of the same root-device, then you can use `malloc_device` on that root-device instead of using the slower `malloc_host`. If you are using `malloc_device` you need an explicit copy out to the host to see the data located there.

Buffers

SYCL buffers that are created against a context and under the hood are mapped to the Level Zero USM allocation. The mapping details are:

- Allocation on an integrated device is made on the host and is accessible by the host and the device without copying.
- Memory buffers for context with sub-devices of the same root-device (possibly including the root-device itself) are allocated on that root-device. They are accessible by all the devices in the context. The synchronization with the host is performed by a SYCL runtime with map/unmap performing implicit copies when necessary.
- Memory buffers for context with devices from different root-devices in it are allocated on host (and are accessible to all devices).

Queues

A SYCL queue is always attached to a single device in a potential multi-device context. The following example scenarios are listed from most to least performant:

Scenario One

Context with a single sub-device in it, where the queue is attached to that sub-device (tile):

- The execution/visibility is limited to the single sub-device only.
- This offers the best performance per tile.

For example:

```
try {
  vector<device> SubDevices = ...;
  for (auto &D : SubDevices) {
    // Each queue is in its own context, no data sharing across them.
    auto Q = queue(D);
    Q.submit([&](handler& cgh) {...});
  }
}
```

Scenario Two

Context with multiple sub-devices of the same root-device (multi-tile):

- The queues are attached to the sub-devices, which implement explicit scaling.
- The root-device should not be passed to this context for better performance.

For example:

```
try {
  vector<device> SubDevices = ...;
  auto C = context(SubDevices);
  for (auto &D : SubDevices) {
    // All queues share the same context, data can be shared across queues.
    auto Q = queue(C, D);
    Q.submit([&](handler& cgh) {...});
  }
}
```

Scenario Three

Context with a single root-device in it, where the queue is attached to that root-device:

- The work is automatically distributed across all sub-devices/tiles via implicit scaling by the driver.
- The simplest way to enable multi-tile hardware, but this does not offer possibility to target specific tiles.

For example:

```
try {
  // The queue is attached to the root-device, driver distributes to sub-devices, if any.
  auto D = device(gpu_selector{});
  auto Q = queue(D);
  Q.submit([&](handler& cgh) {...});
}
```

Scenario Four

Contexts with multiple root-devices (multi-card):

- The most unrestrictive context with queues attached to different root-devices.
- Offers most sharing possibilities at the cost of slow access through host memory or explicit copies needed.

For example:

```
try {
  auto P = platform(gpu_selector{});
  auto RootDevices = P.get_devices();
  auto C = context(RootDevices);
  for (auto &D : RootDevices) {
    // Context has multiple root-devices, data can be shared across multi-card (requires
explicit copying)
    auto Q = queue(C, D);
    Q.submit([&](handler& cgh) {...});
  }
}
```

NOTE Do not forget to allocate/synchronize your memory for your programming model and algorithm.

Multi-tile/card Examples

For your next steps, you can explore two examples of multi-tile and multi-card programming:

- dgemm
- gpu2gpu

Vectorization

Vectorization is the process of converting an algorithm from a scalar implementation, which does an operation one pair of operands at a time, to a vector process where a single instruction can refer to a vector (a series of adjacent values).

Automatic Vectorization

The automatic vectorizer (also called the auto-vectorizer) is a component of the compiler that automatically uses SIMD instructions in the Intel® Streaming SIMD Extensions (Intel® SSE, Intel® SSE2, Intel® SSE3 and Intel® SSE4), Supplemental Streaming SIMD Extensions (SSSE3) instruction sets, Intel® Advanced Vector Extensions (Intel® AVX, Intel® AVX2) instruction sets, and Intel® Advanced Vector Extensions 512 (Intel® AVX-512) instruction set. The vectorizer detects operations in the program that can be done in parallel and converts the sequential operations to parallel; for example, the vectorizer converts the sequential SIMD instruction that processes up to 16 elements into a parallel operation, depending on the data type.

Automatic vectorization occurs when the compiler generates packed SIMD instructions to unroll a loop. Because the packed instructions operate on more than one data element at a time, the loop executes more efficiently. This process is referred to as auto-vectorization only to emphasize that the compiler identifies and optimizes suitable loops on its own, without external input. However, it is useful to note that in some cases, certain keywords or directives may be applied in the code for auto-vectorization to occur.

The compiler supports a variety of auto-vectorizing hints that can help the compiler to generate effective vector instructions. Automatic vectorization is supported on IA-32 (for C++ only) and Intel® 64 architectures. Intel® Advisor, a separate tool included in the Intel® oneAPI Base Toolkit, provides a Vectorization Advisor feature that can analyze the compiler's optimization reports and make recommendations for enhancing vectorization.

NOTE
This option enables vectorization at default optimization levels for both Intel® microprocessors and non-Intel microprocessors. Vectorization may call library routines that can result in additional performance gain on Intel® microprocessors than on non-Intel microprocessors.

Vectorization Programming Guidelines

The goal of including the vectorizer component in the Intel® oneAPI DPC++/C++ Compiler is to exploit single-instruction multiple data (SIMD) processing automatically. Users can help by supplying the compiler with additional information; for example, by using auto-vectorizer hints or pragmas.

NOTE
This option enables vectorization at default optimization levels for both Intel® microprocessors and non-Intel microprocessors. Vectorization may call library routines that can result in additional performance gain on Intel® microprocessors than on non-Intel microprocessors.

Guidelines to Vectorize Innermost Loops

Follow these guidelines to vectorize innermost loop bodies.

Use:

- Straight-line code (a single basic block).
- Vector data only (arrays and invariant expressions on the right hand side of assignments). Array references can appear on the left hand side of assignments.
- Only assignment statements.

Avoid:

- Function calls (other than math library calls).
- Non-vectorizable operations (either because the loop cannot be vectorized, or because an operation is emulated through a number of instructions).
- Mixing vectorizable types in the same loop (leads to lower resource utilization).
- Data-dependent loop exit conditions (leads to loss of vectorization).

To make your code vectorizable, you need to edit your loops. You should only make changes that enable vectorization, and avoid these common changes:

- Loop unrolling, which the compiler performs automatically.
- Decomposing one loop with several statements in the body into several single-statement loops.

Restrictions

There are a number of restrictions that you should consider. Vectorization depends on two major factors: hardware and style of source code.

Factor	Description
Hardware	The compiler is limited by restrictions imposed by the underlying hardware. Intel® Streaming SIMD Extensions (Intel® SSE) has vector memory operations that are limited to stride-1 accesses with a preference to 16-byte-aligned memory references. This means that if the compiler abstractly recognizes a loop as vectorizable, it still might not vectorize it for a distinct target architecture.

Factor	Description
Style of source code	The style in which you write source code can inhibit vectorization. For example, a common problem with global pointers is that they often prevent the compiler from being able to prove that two memory references refer to distinct locations. Consequently, this prevents certain reordering transformations.

Many stylistic issues that prevent automatic vectorization by compilers are found in loop structures. The ambiguity arises from the complexity of the keywords, operators, data references, pointer arithmetic, and memory operations within the loop bodies.

By understanding these limitations and by knowing how to interpret diagnostic messages, you can modify your program to overcome the known limitations and enable effective vectorization.

Guidelines for Writing Vectorizable Code

Follow these guidelines to write vectorizable code:

- Use simple `for` loops. Avoid complex loop termination conditions – the upper iteration limit must be invariant within the loop. For the innermost loop in a nest of loops, you could set the upper limit iteration to be a function of the outer loop indices.
- Write straight-line code. Avoid branches such as `switch`, `goto`, or `return` statements; most function calls; or `if` constructs that cannot be treated as masked assignments.
- Avoid dependencies between loop iterations or at the least, avoid read-after-write dependencies.
- Try to use array notations instead of the use of pointers. C programs in particular impose very few restrictions on the use of pointers; aliased pointers may lead to unexpected dependencies. Without help, the compiler often cannot tell whether it is safe to vectorize code containing pointers.
- Wherever possible, use the loop index directly in array subscripts instead of incrementing a separate counter for use as an array address.
- Access memory efficiently:
  - Favor inner loops with unit stride.
  - Minimize indirect addressing.
  - Align your data to 16-byte boundaries (for Intel® SSE instructions).
- Choose a suitable data layout with care. Most multimedia extension instruction sets are rather sensitive to alignment. The data movement instructions of Intel® SSE, for example, operate much more efficiently on data that is aligned at a 16-byte boundary in memory. Therefore, the success of a vectorizing compiler also depends on its ability to select an appropriate data layout which, in combination with code restructuring (like loop peeling), results in aligned memory accesses throughout the program.
- Use aligned data structures: Data structure alignment is the adjustment of any data object in relation with other objects.

  You can use the declaration `__declspec(align)`.

  Caution Use this hint with care. Incorrect usage of aligned data movements result in an exception when using Intel® SSE.

- Use structure of arrays (SoA) instead of array of structures (AoS): An array is the most common type of data structure that contains a contiguous collection of data items that can be accessed by an ordinal index. You can organize this data as an array of structures (AoS) or as a structure of arrays (SoA). While AoS organization is excellent for encapsulation, it can be a hindrance for use of vector processing. To make vectorization of the resulting code more effective, you can also select appropriate data structures.

Dynamic Alignment Optimizations

Dynamic alignment optimizations can improve the performance of vectorized code, especially for long trip count loops. Disabling such optimizations can decrease performance, but it may improve bitwise reproducibility of results, factoring out data location from possible sources of discrepancy.

To enable or disable dynamic data alignment optimizations, specify the option `Qopt-dynamic-align[-]` (Windows) or `[no-]qopt-dynamic-align[-]` (Linux).

Use Aligned Data Structures

Data structure alignment is the adjustment of any data object with relation to other objects. The Intel® oneAPI DPC++/C++ Compiler may align individual variables to start at certain addresses to speed up memory access. Misaligned memory accesses can incur large performance losses on certain target processors that do not support them in hardware.

Alignment is a property of a memory address, expressed as the numeric address modulo of powers of two. In addition to its address, a single datum also has a size. A datum is called 'naturally aligned' if its address is aligned to its size, otherwise it is called 'misaligned'. For example, an 8-byte floating-point datum is naturally aligned if the address used to identify it is aligned to eight (8).

A data structure is a way of storing data in a computer so that it can be used efficiently. Often, a carefully chosen data structure allows a more efficient algorithm to be used. A well-designed data structure allows a variety of critical operations to be performed, using as little resources (execution time and memory space) as possible.

```
struct MyData{
   short   Data1;
   short   Data2;
   short   Data3;
};
```

In the example data structure above, if the type `short` is stored in two bytes of memory then each member of the data structure is aligned to a boundary of two bytes. `Data1` would be at offset `0`, `Data2` at offset `2` and `Data3` at offset `4`. The size of this structure is six bytes. The type of each member of the structure usually has a required alignment, meaning that it is aligned on a pre-determined boundary, unless you request otherwise. In cases where the compiler has taken sub-optimal alignment decisions, you can use the declaration `declspec(align(base,offset))`, where `0 <= offset < base` and `base` is a power of two, to allocate a data structure at offset from a certain base.

Consider as an example, that most of the execution time of an application is spent in a loop of the following form:

```
double a[N], b[N];
  ...
for (i = 0; i < N; i++){ a[i+1] = b[i] * 3; }
```

If the first element of both arrays is aligned at a 16-byte boundary, then either an unaligned load of elements from `b` or an unaligned store of elements into `a` must be used after vectorization.

In this instance, peeling off an iteration does not help but you can enforce the alignment shown below. This alignment results in two aligned access patterns after vectorization (assuming an 8-byte size for doubles):

```
__declspec(align(16, 8))  double a[N];
__declspec(align(16, 0))  double b[N];
/* or simply "align(16)"  */
```

If pointer variables are used, the compiler is usually not able to determine the alignment of access patterns at compile time. Consider the following simple `fill()` function:

```
void fill(char *x) {
  int i;
  for (i = 0; i < 1024; i++){ x[i] = 1; }
}
```

Without more information, the compiler cannot make any assumption on the alignment of the memory region accessed by the above loop. At this point, the compiler may decide to vectorize this loop using unaligned data movement instructions or, generate the runtime alignment optimization shown here:

```
peel = x & 0x0f;
if (peel != 0) {
  peel = 16 - peel;
  /* runtime peeling loop */
  for (i = 0; i < peel; i++) { x[i] = 1; }
}

/* aligned access */
for (i = peel; i < 1024; i++) { x[i] = 1; }
```

Runtime optimization provides a generally effective way to obtain aligned access patterns at the expense of a slight increase in code size and testing. If incoming access patterns are aligned at a 16-byte boundary, you can avoid this overhead with the hint `__assume_aligned(x, 16);` in the function to convey this information to the compiler.

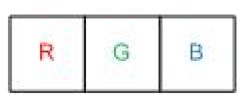
For example, suppose you can introduce an optimization in the case where a block of memory with address `n2` is aligned on a 16-byte boundary. You could use `_assume(n2%16==0)`.

Caution Incorrect use of aligned data movements result in an exception for Intel® SSE.

Use Structure of Arrays Versus Array of Structures

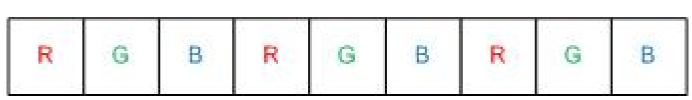
The most common and well-known data structure is the array that contains a contiguous collection of data items, which can be accessed by an ordinal index. This data can be organized as an array of structures (AoS) or as a structure of arrays (SoA). While AoS organization works excellently for encapsulation, for vector processing it works poorly.

You can select appropriate data structures to make vectorization of the resulting code more effective. To illustrate this point, compare the traditional array of structures (AoS) arrangement for storing the `r`, `g`, `b` components of a set of three-dimensional points with the alternative structure of arrays (SoA) arrangement for storing this set.



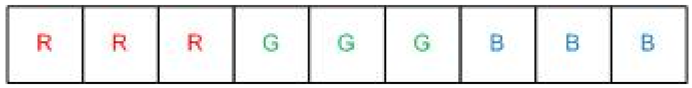
For example, a point structure with data in an AoS arrangement:

```
struct Point{
   float r;
   float g;
   float b;
}
```



For example, a points structure with data in a SoA arrangement:

```
struct Points{
   float* x;
   float* y;
   float* z;
}
```



With the AoS arrangement, a loop that visits all components of an RGB point before moving to the next point exhibits a good locality of reference. This is because all elements in the fetched cache lines are used. The disadvantage of the AoS arrangement is that each individual memory reference in such a loop exhibits a non-unit stride, which, in general, adversely affects vector performance. Furthermore, a loop that visits only one component of all points exhibits less satisfactory locality of reference because many of the elements in the fetched cache lines remain unused.

With the SoA arrangement, the unit-stride memory references are more amenable to effective vectorization and still exhibit good locality of reference within each of the three data streams. Consequently, an application that uses the SoA arrangement may outperform an application based on the AoS arrangement when compiled with a vectorizing compiler. This performance difference may not be obviously apparent during the early implementation phase.

Before you start vectorization, try out some simple rules:

- Make your data structures vector-friendly.
- Make sure that inner loop indices correspond to the outermost (last) array index in your data (row-major order).
- Use structure of arrays over array of structures.

For instance when dealing with three-dimensional coordinates, use three separate arrays for each component (SoA), instead of using one array of three-component structures (AoS). To avoid dependencies between loops that will eventually prevent vectorization, use three separate arrays for each component (SoA), instead of one array of three-component structures (AoS). When you use the AoS arrangement, each iteration produces one result by computing XYZ, but it can at best use only 75% of the SSE unit because the fourth component is not used. Sometimes, the compiler may use only one component (25%). When you use the SoA arrangement, each iteration produces four results by computing XXXX, YYYY and ZZZZ, using 100% of the SSE unit. A drawback for the SoA arrangement is that your code will likely be three times as long.

If your original data layout is in AoS format, you may want to consider a conversion to SoA before the critical loop:

- Use the smallest data types that give the needed precision to maximize potential SIMD width. (If only 16-bits are needed, using a `short` rather than an `int` can make the difference between 8-way or four-way SIMD parallelism.)
- Avoid mixing data types to minimize type conversions.
- Avoid operations not supported in SIMD hardware.
- Use all the instruction sets available for your processor. Use the appropriate command line option for your processor type, or select the appropriate IDE option (Windows only):
  - **Project** > **Properties** > **C/C++** > **Code Generation** > **Intel Processor-Specific Optimization**, if your application runs only on Intel® processors.
  - **Project** > **Properties** > **C/C++** > **Code Generation** > **Enable Enhanced Instruction Set**, if your application runs on compatible, non-Intel processors.
- Vectorizing compilers usually have some built-in efficiency heuristics to decide whether vectorization is likely to improve performance. The Intel® oneAPI DPC++/C++ Compiler disables vectorization of loops with many unaligned or non-unit stride data access patterns. If experimentation reveals that vectorization

improves performance, you can override this behavior using the `#pragma vector always` hint before the loop. The compiler vectorizes any loop regardless of the outcome of the efficiency analysis (provided that vectorization is safe).

See Also

__declspec(align)

Vectorization and Loops

Loop Constructs

Use Automatic Vectorization

Automatic vectorization is supported on Intel® 64 architectures. The information below will guide you in setting up the auto-vectorizer.

Vectorization Speedup

Where does the vectorization speedup come from? Consider the following sample code, where `a`, `b`, and `c` are integer arrays:

```
for (i=0;i<=MAX;i++)
   c[i]=a[i]+b[i];
```

If vectorization is not enabled, and you compile using the `O1`, `-no-vec-` (Linux), or `/Qvec-` (Windows) option, the compiler processes the code with unused space in the SIMD registers, even though each register can hold three additional integers. If vectorization is enabled (compiled using `O2` or higher options), the compiler may use the additional registers to perform four additions in a single instruction. The compiler looks for vectorization opportunities whenever you compile at default optimization (`O2`) or higher.

NOTE
This option enables vectorization at default optimization levels for both Intel® microprocessors and non-Intel microprocessors. Vectorization may call library routines that can result in additional performance gain on Intel® microprocessors than on non-Intel microprocessors.

To get details about the type of loop transformations and optimizations that took place, use the `[Q]opt-report-phase` option by itself or along with the `[Q]opt-report` option.

Linux

To evaluate performance enhancement, run `vec_samples`:

1. Source an environment script such as `vars.sh` in the `<installdir>` directory and use the attribute appropriate for the architecture.
2. Navigate to the `<installdir>\Samples\<locale>\C++\` directory. This application multiplies a vector by a matrix using the following loop:

```
for (j = 0;j < size2; j++) { b[i] += a[i][j] * x[j]; }
```

3. Build and run the application, first without enabling auto-vectorization. The default `O2` optimization enables vectorization, so you need to disable it with a separate option.

```
icx -O2 -no-vec  Multiply.c -o NoVectMult
./NoVectMult
```

4. Build and run the application, this time with auto-vectorization.

```
icx -O2 -qopt-report=3 -vec Multiply.c -o VectMult
./VectMult
```

Windows

To evaluate performance enhancement, run `vec_samples`:

1. Under the **Start** menu item for your product, select an icon under **Intel oneAPI <version>** > **Intel oneAPI Command Prompt** for oneAPI Compilers.

2. Navigate to the `<installdir>\Samples\<locale>\C++\`directory. On Windows, unzip the sample project `vec_samples.zip` to a writable directory. This small application multiplies a vector by a matrix using the following loop:

```
for (j = 0;j < size2; j++) { b[i] += a[i][j] * x[j]; }
```

3. Build and run the application, first without enabling auto-vectorization. The default `O2` optimization enables vectorization, so you need to disable it with a separate option.

```
icx /O2 /Qvec- Multiply.c /FeNoVectMult
NoVectMult
```

4. Build and run the application, this time with auto-vectorization.

```
icx /O2 /Qopt-report:3 /Qvec Multiply.c /FeVectMult
VectMult
```

When you compare the timing of the two runs, you may see that the vectorized version runs faster. The time for the non-vectorized version is only slightly faster than would be obtained by compiling with the `O1` option.

Obstacles to Vectorization

The following issues do not always prevent vectorization, but frequently cause the compiler to decide that vectorization would not be worthwhile.

- **Non-contiguous memory access:** Four consecutive integers or floating-point values, or two consecutive doubles, may be loaded directly from memory in a single SSE instruction. But if the four integers are not adjacent, they must be loaded separately using multiple instructions, which is considerably less efficient. The most common examples of non-contiguous memory access are loops with non-unit stride or with indirect addressing, shown in the examples below. The compiler rarely vectorizes these loops, unless the amount of computational work is larger compared to the overhead from non-contiguous memory access.

```
// arrays accessed with stride 2
for (int i=0; i<SIZE; i+=2)  b[i] += a[i] * x[i];

// inner loop accesses a with stride SIZE
for (int j=0; j<SIZE; j++)  {
  for (int i=0; i<SIZE; I++)   b[i] += a[i][j] * x[j];
}

// indirect addressing of x using index array
  for (int i=0; i<SIZE; i+=2)  b[i] += a[i] * x[index[i]];
```

  The typical message from the vectorization report is: `vectorization possible but seems inefficient`, although indirect addressing may also result in the following report: `existence of vector dependence`.

- **Data dependencies:** Vectorization entails changes in the order of operations within a loop, since each SIMD instruction operates on several data elements at once. Vectorization is only possible if this change of order does not change the results of the calculation.

- The simplest case is when data elements that are written (stored to) do not appear in any other iteration of the individual loop. In this case, all the iterations of the original loop are independent of each other, and can be executed in any order, without changing the result. The loop may be safely executed using any parallel method, including vectorization.
- When a variable is written in one iteration and read in a subsequent iteration, there is a read-after-write dependency, also known as a flow dependency, for example:

```
A[0]=0;
for (j=1; j<MAX; j++)  A[j]=A[j-1]+1;
          // this is equivalent to:
   A[1]=A[0]+1;
   A[2]=A[1]+1;
   A[3]=A[2]+1;
   A[4]=A[3]+1;
```

  The value of `j` is propagated to all `A[j]`. This cannot safely be vectorized: if the first two iterations are executed simultaneously by a SIMD instruction, the value of `A[1]` is used by the second iteration before it has been calculated by the first iteration.
- When a variable is read in one iteration and written in a subsequent iteration, this is a write-after-read dependency, also known as an anti-dependency, for example:

```
for (j=1; j<MAX; j++)  A[j-1]=A[j]+1;
          // this is equivalent to:
   A[0]=A[1]+1;
   A[1]=A[2]+1;
   A[2]=A[3]+1;
   A[3]=A[4]+1;
```

  This write-after-read dependency is not safe for general parallel execution, since the iteration with the write may execute before the iteration with the read. No iteration with a higher value of `j` can complete before an iteration with a lower value of `j`, and so vectorization is safe (it gives the same result as non-vectorized code).

  The following example may not be safe, since vectorization might cause some elements of `A` to be overwritten by the first SIMD instruction before being used for the second SIMD instruction.

```
for (j=1; j<MAX; j++)  {
  A[j-1]=A[j]+1;
}

   // this is equivalent to:
   A[0]=A[1]+1;
   A[1]=A[2]+1;
   A[2]=A[3]+1;
   A[3]=A[4]+1;
```

- Read-after-read situations are not really dependencies, and do not prevent vectorization or parallel execution. If a variable is unwritten, it does not matter how often it is read.
- Write-after-write, or output dependencies, where the same variable is written to in more than one iteration, are generally unsafe for parallel execution, including vectorization.
- One important exception that contains all of the above types of dependency is:

```
sum=0;
for (j=1; j<MAX; j++)  sum = sum + A[j]*B[j]
```

  Although `sum` is both read and written in every iteration, the compiler recognizes such reduction idioms, and is able to vectorize them safely. The loop in the first example was another example of a reduction, with a loop-invariant array element in place of a scalar.

  These types of dependencies between loop iterations are sometimes known as loop-carried dependencies.

The above examples are of proven dependencies. The compiler cannot safely vectorize a loop if there is even a potential dependency. For example:

```
for (i = 0; i < size; i++) { c[i] = a[i] * b[i]; }
```

In the above example, the compiler needs to determine whether, for some iteration `i`, `c[i]` might refer to the same memory location as `a[i]` or `b[i]` for a different iteration. Such memory locations are sometimes said to be aliased. For example, if `a[i]` pointed to the same memory location as `c[i-1]`, there would be a read-after-write dependency. If the compiler cannot exclude this possibility, it will not vectorize the loop unless you provide the compiler with hints.

Help the Compiler Vectorize

Sometimes the compiler has insufficient information to decide to vectorize a loop. There are several ways to provide additional information to the compiler:

- **Pragmas:**
  - `#pragma ivdep`: may be used to tell the compiler that it may safely ignore any potential data dependencies. (The compiler will not ignore proven dependencies). Use of this pragma when there are dependencies may lead to incorrect results.

    There are cases where the compiler cannot tell by a static dependency analysis that it is safe to vectorize. Consider the following loop:

```
void copy(char *cp_a, char *cp_b, int n) {
  for (int i = 0; i < n; i++) { cp_a[i] = cp_b[i]; }
}
```

Without more information, a vectorizing compiler must conservatively assume that the memory regions accessed by the pointer variables `cp_a` and `cp_b` may (partially) overlap, which can cause potential data dependencies that prohibit straightforward conversion of this loop into SIMD instructions. At this point, the compiler may decide to keep the loop serial or generate a runtime test for overlap, where the loop in the true-branch can be converted into SIMD instructions:

```
if (cp_a + n < cp_b || cp_b + n < cp_a)
          /* vector loop */
  for (int i = 0; i < n; i++) cp_a[i] = cp_b [I];
  else
          /* serial loop */
  for (int i = 0; i < n; i++) cp_a[i] = cp_b[i];
```

Runtime data-dependency testing provides a way to exploit implicit parallelism in C or C++ code at the expense of a slight increase in code size and testing overhead. If the function copy is only used in specific ways, you can help the compiler:

- If the function is mainly used for small values of `n` or for overlapping memory regions, you can prevent vectorization and the corresponding runtime overhead by inserting a `#pragma novector` hint before the loop.
- Conversely, if the loop is guaranteed to operate on non-overlapping memory regions, you can provide this information to the compiler by means of a `#pragma ivdep` hint before the loop. This tells the compiler that conservatively assumed data dependencies that prevent vectorization can be ignored and results in vectorization of the loop without runtime data-dependency testing.

```
#pragma ivdep
void copy(char *cp_a, char *cp_b, int n) {
  for (int i = 0; i < n; i++) { cp_a[i] = cp_b[i]; }
}
```

NOTE You can also use the `restrict` keyword.

  - `#pragma loop count (n)`: gives the typical trip count of the loop. This helps the compiler decide if vectorization is worthwhile, or if it should generate alternative code paths for the loop.
  - `#pragma vector always`: asks the compiler to vectorize the loop.
  - `#pragma vector align`: asserts that data within the following loop is aligned (to a 16-byte boundary, for Intel® SSE instruction sets).
  - `#pragma novector`: asks the compiler not to vectorize a particular loop.
  - `#pragma vector nontemporal`: gives a hint to the compiler that data will not be reused, and to use streaming stores that bypass cache.
- **Keywords:** The `restrict` keyword is used to assert that the memory referenced by a pointer is not aliased. The keyword requires the use of the `[Q]std=c99` compiler option. The example under `#pragma ivdep` above can also be handled using the `restrict` keyword.

  You may use the `restrict` keyword in the declarations of `cp_a` and `cp_b`, as shown below, to inform the compiler that each pointer variable provides exclusive access to a certain memory region. The `restrict` qualifier in the argument list lets the compiler know that there are no other aliases to the memory where the pointers point. The pointer where it is used provides the only means of accessing the memory in the scope where the pointers live. Even if the code gets vectorized without the `restrict` keyword, the compiler checks for aliasing at runtime, if the `restrict` keyword was used.

```
void copy(char * __restrict cp_a, char * __restrict cp_b, int n) {
  for (int i = 0; i < n; i++) cp_a[i] = cp_b[i];
}
```

  This method is best used when the exclusive access property holds for the pointer variables in your code with many loops, because it avoids annotating each of the vectorizable loops individually. Both the loop-specific `#pragma ivdep` hint, and the pointer variable-specific `restrict` hint must be used with care because incorrect usage may change the semantics intended in the original program.

  Another example is the following loop that may also not get vectorized because of a potential aliasing problem between pointers `a`, `b`, and `c`:

```
void add(float *a, float *b, float *c) {
  for (int i=0; i<SIZE; i++) { c[i] += a[i] + b[i]; }
}
```

  If the `restrict` keyword is added to the parameters, the compiler assumes that you will not access the memory in question with any other pointer and vectorize the code properly:

```
// let the compiler know, the pointers are safe with restrict
void add(float * __restrict a, float * __restrict b, float * __restrict c) {
  for (int i=0; i<SIZE; i++) { c[i] += a[i] + b[i]; }
}
```

  The down-side of using `restrict` is that not all compilers support this keyword, so your source code may lose portability.
- **Options/switches:** You can use options to enable different levels of optimizations to achieve automatic vectorization:

  - **Interprocedural optimization (IPO):** Enable IPO using the `[Q]ipo` option across source files. You provide the compiler with additional information (trip counts, alignment, or data dependencies) about a loop. Enabling IPO may also allow inlining of function calls.
  - **High-level optimizations (HLO):** Enable HLO with option `O3`. This enables additional loop optimizations that make it easier for the compiler to vectorize the transformed loops.

See Also

qopt-report, Qopt-report compiler option

Vectorization and Loops

This topic provides more information on the interaction between the auto-vectorizer and loops.

See Programming Guidelines for Vectorization.

In some rare cases, a successful loop parallelization (either automatically or by means of OpenMP directives) may affect the messages reported by the compiler for a non-vectorizable loop in a non-intuitive way.

Types of Vectorized Loops

For integer loops, the 128-bit Intel® Streaming SIMD Extensions (Intel® SSE) and the Intel® Advanced Vector Extensions (Intel® AVX) provide SIMD instructions for most arithmetic and logical operators on 32-bit, 16-bit, and 8-bit integer data types, with limited support for the 64-bit integer data type.

Vectorization may proceed if the final precision of integer wrap-around arithmetic is preserved. A 32-bit shift-right operator, for instance, is not vectorized in 16-bit mode if the final stored value is a 16-bit integer. Also, note that because the Intel® SSE and the Intel® AVX instruction sets are not fully orthogonal (shifts on byte operands, for instance, are not supported), not all integer operations can actually be vectorized.

For loops that operate on 32-bit single-precision and 64-bit double-precision floating-point numbers, Intel® SSE provides SIMD instructions for the following arithmetic operators:

- Addition (+)
- Subtraction (-)
- Multiplication (*)
- Division (/)

Additionally, Intel® SSE provide SIMD instructions for the binary `MIN` and `MAX` and unary `SQRT` operators. SIMD versions of several other mathematical operators (like the trigonometric functions `SIN`, `COS`, and `TAN`) are supported in software in a vector mathematical run-time library that is provided with the compiler.

To be vectorizable, loops must be:

- **Countable:** The loop trip count must be known at entry to the loop at runtime, though it need not be known at compile time (that is, the trip count can be a variable but the variable must remain constant for the duration of the loop). This implies that exit from the loop must not be data-dependent.
- **Single entry and single exit:** as is implied by stating that the loop must be countable.
- **Contain straight-line code:** SIMD instruction perform the same operation on data elements from multiple iterations of the original loop, therefore, it is not possible for different iterations to have different control flow; that is, they must not branch. It follows that `switch` statements are not allowed. However, `if` statements are allowed if they can be implemented as masked assignments, which is usually the case. The calculation is performed for all data elements but the result is stored only for those elements for which the mask evaluates to true.
- **Innermost loop of a nest:** The only exception is if an original outer loop is transformed into an inner loop as a result of some other prior optimization phase, such as unrolling, loop collapsing or interchange, or an original outermost loop is transformed to an innermost loop due to loop materialization.
- **Without function calls:** Even a `print` statement is sufficient to prevent a loop from getting vectorized. The vectorization report message is typically: non-standard loop is not a vectorization candidate. The two major exceptions are for intrinsic math functions and for functions that may be inlined.

Intrinsic math functions are allowed, because the compiler runtime library contains vectorized versions of these functions. See below for a list of these functions; most exist in both float and double versions:

- `acos`
- `acosh`
- `asin`
- `asinh`
- `atan`

- atan2
- atanh
- cbrt
- ceil
- cos
- cosh
- erf
- erfc
- erfinv
- exp
- exp2
- fabs
- floor
- fmax
- fmin
- log
- log2
- log10
- pow
- round
- sin
- sinh
- sqrt
- tan
- tanh
- trunc

Statements in the Loop Body

The vectorizable operations are different for floating-point and integer data.

Integer Array Operations

The statements within the loop body may contain *char*, *unsigned char*, *short*, *unsigned short*, *int*, and *unsigned int*. Calls to functions such as `sqrt` and `fabs` are also supported. Arithmetic operations are limited to addition, subtraction, bitwise `AND`, `OR`, and `XOR` operators, division (via run-time library call), multiplication, `min`, and `max`. You can mix data types but this may potentially cost you in terms of lowering efficiency. Some example operators where you can mix data types are multiplication, shift, or unary operators.

Other Operations

No statements other than the preceding floating-point and integer operations are allowed. In particular, note that the special `__m64` `__m128`, and `__m256` data types are not vectorizable. The loop body cannot contain any function calls. Use of Intel® SSE intrinsics (for example, `_mm_add_ps`) or Intel® AVX intrinsics (for example, `_mm256_add_ps`) are not allowed.

See Also

Programming Guidelines for Vectorization

Loop Constructs

Loops can be formed with the usual `for` and `while` constructs. Loops must have a single entry and a single exit to be vectorized. The following examples illustrate loop constructs that can and cannot be vectorized. The non-vectorizable structure example shows a loop that cannot be vectorized because of the inherent potential for an early exit from the loop.

Vectorizable structure:

```
void vec(float a[], float b[], float c[]) {
  int i = 0;
  while (i < 100) {
// The if branch is inside body of loop.
    a[i] = b[i] * c[i];
    if (a[i] < 0.0)
        a[i] = 0.0;
        i++;
  }
}
```

Non-vectorizable structure:

```
void no_vec(float a[], float b[], float c[]) {
  int i = 0;
  while (i < 100) {
    if (a[i] < 50)
// The next statement is a second exit
// that allows an early exit from the loop.
      break;
    ++i;
  }
}
```

Loop Exit Conditions

Loop exit conditions determine the number of iterations a loop executes. For example, fixed indexes for loops determine the iterations. The loop iterations must be countable and the number of iterations must be expressed as one of the following:

- A constant.
- A loop invariant term.
- A linear function of outermost loop indices.

In the case where a loops exit depends on computation, the loops are not countable. The examples below show loop constructs that are countable and non-countable. The non-countable loop example demonstrates a loop construct that is non-countable due to dependency loop variant count value.

Countable loop, example one:

```
void cnt1(float a[], float b[], float c[],
          int n, int lb) {
// Exit condition specified by "N-1b+1"
  int cnt=n, i=0;
  while (cnt >= lb) {
// lb is not affected within loop.
    a[i] = b[i] * c[i];
    cnt--;
```

```
    i++;
  }
}
```

Countable loop, example two:

```
void vec(float a[], float b[], float c[]) {
  int i = 0;
  while (i < 100) {
// The if branch is inside body of loop.
    a[i] = b[i] * c[i];
    if (a[i] < 0.0)
        a[i] = 0.0;
        i++;
  }
}
```

Non-countable loop:

```
void no_cnt(float a[], float b[], float c[]) {
  int i=0;
// Iterations dependent on a[i].
  while (a[i]>0.0) {
    a[i] = b[i] * c[i];
    i++;
  }
}
```

Strip-mining and Cleanup

Strip-mining, also known as loop sectioning, is a loop transformation technique for enabling SIMD-encoding of loops, as well as a means of improving memory performance. By fragmenting a large loop into smaller segments or strips, this technique transforms the loop structure in two ways:

- By increasing the temporal and spatial locality in the data cache if the data is reusable in different passes of an algorithm.
- By reducing the number of iterations of the loop by a factor of the length of each vector, or number of operations being performed per SIMD operation. With the Intel® Streaming SIMD Extensions (Intel® SSE), the vector or strip-length is reduced by four times: four floating-point data items per single Intel® SSE single-precision floating-point SIMD operation are processed.

First introduced for vectorizers, this technique consists of the generation of code when each vector operation is done for a size less than or equal to the maximum vector length on a given vector machine.

The compiler automatically strip-mines your loop and generates a cleanup loop. For example, assume the compiler attempts to strip-mine the loop before vectorization. After vectorization, the compiler might handle the strip mining and loop cleaning by restructuring the loop.

Before vectorization:

```
i=0;
while(i<n) {
  // Original loop code
  a[i]=b[i]+c[i];
  ++i;
}
```

After vectorization:

```
// The vectorizer generates the following two loops
i=0;
while(i<(n-n%4)) {
  // Vector strip-mined loop
  // Subscript [i:i+3] denotes SIMD execution
  a[i:i+3]=b[i:i+3]+c[i:i+3];
  i=i+4;
}
while(i<n) {
  // Scalar clean-up loop
  a[i]=b[i]+c[i];
  ++i;
}
```

Loop Blocking

It is possible to treat loop blocking as strip-mining in two or more dimensions. Loop blocking is a useful technique for memory performance optimization. The main purpose of loop blocking is to eliminate as many cache misses as possible. This technique transforms the memory domain into smaller chunks rather than sequentially traversing through the entire memory domain. Each chunk should be small enough to fit all the data for a given computation into the cache, maximizing data reuse.

Consider the following example, loop blocking allows arrays *A* and *B* to be blocked into smaller rectangular chunks so that the total combined size of two blocked (*A* and *B*) chunks is smaller than cache size, which can improve data reuse.

The transformed loop after blocking example illustrates loop blocking the `add` function (from the original loop example). In order to benefit from this optimization, you might have to increase the cache size.

Original loop:

```
#include <time.h>
#include <stdio.h>
#define MAX 7000

void add(int a[][MAX], int b[][MAX]);
int main() {
int i, j;
int A[MAX][MAX];
int B[MAX][MAX];
time_t start, elaspe;
int sec;

//Initialize array
for(i=0;i<MAX;i++) {
  for(j=0;j<MAX; j++) {
    A[i][j]=j;
    B[i][j]=j;
  }
}

 start= time(NULL);
 add(A, B);
 elaspe=time(NULL);
 sec = elaspe - start;
 printf("Time %d",sec); //List time taken to complete add function
}
```

```
void add(int a[][MAX], int b[][MAX]) {
 int i, j;
 for(i=0;i<MAX;i++) {
  for(j=0; j<MAX;j++ {
     a[i][j] = a[i][j] + b[j][i]; //Adds two matrices
    }
  }
}
```

Transformed loop after blocking:

```
#include <stdio.h>
#include <time.h>
#define MAX 7000
void add(int a[][MAX], int b[][MAX]);

int main() {
  #define BS 8  //Block size is selected as the loop-blocking factor.
  int i, j;
  int A[MAX][MAX];
  int B[MAX][MAX];
  time_t start, elaspe;
  int sec;

//initialize array
for(i=0;i<MAX;i++) {
  for(j=0;j<MAX;j++) {
    A[i][j]=j;
    B[i][j]=j;
  }
}
start= time(NULL);

add(A, B);
elapse=time(NULL);
sec = elapse - start;
printf("Time %d",sec); //Display time taken to complete loopBlocking function
}

void add(int a[][MAX], int b[][MAX]) {
  int i, j, ii, jj;
  for(i=0;i<MAX;i+=BS) {
   for(j=0; j<MAX;j+=BS) {
     for(ii=i; ii<i+BS; ii++) {   //outer loop
       for(jj=j;jj<j+BS; jj++) {  //Array B experiences one cache miss
                                  //for every iteration of outer loop
         a[ii][jj] = a[ii][jj] + b[jj][ii]; //Add the two arrays
        }
      }
    }
  }
}
```

Loop Interchange and Subscripts with Matrix Multiply

Loop interchange is often used for improving memory access patterns. Matrix multiplication is commonly written as shown in the typical matrix multiplication example.

The use of *B*(*K*,*J*) is not a stride-1 reference and therefore will not be vectorized efficiently.

If the loops are interchanged, all the references become stride-1 as shown in the matrix multiplication with stride-1 example.

Typical matrix multiplication:

```
void matmul_slow(float *a[], float *b[], float *c[]) {
  int N = 100;
  for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
      for (int k = 0; k < N; k++)
        c[i][j] = c[i][j] + a[i][k] * b[k][j];
}
```

Matrix multiplication with stride-1:

```
void matmul_fast(float *a[], float *b[], float *c[]) {
  int N = 100;
  for (int i = 0; i < N; i++)
    for (int k = 0; k < N; k++)
      for (int j = 0; j < N; j++)
        c[i][j] = c[i][j] + a[i][k] * b[k][j];
}
```

Interchanging is not always possible because of dependencies, which can lead to different results.

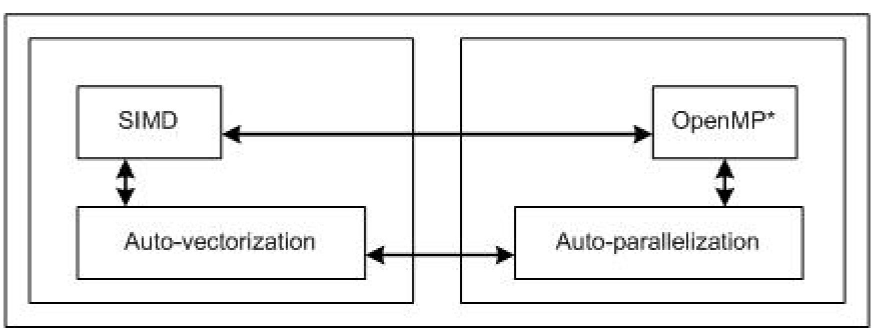
Explicit Vector Programming

This section contains information about explicit vector programming.

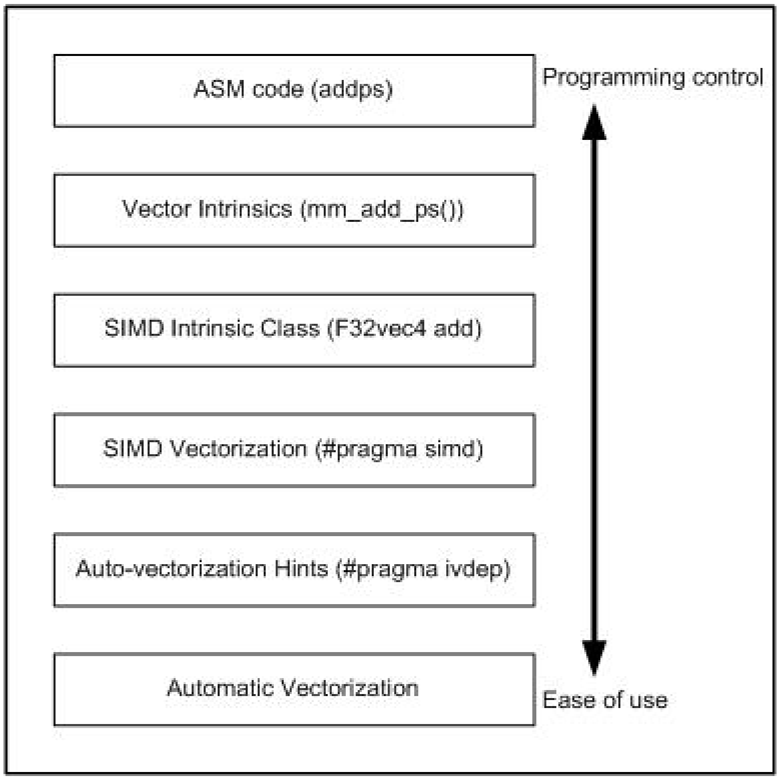
User-mandated or SIMD Vectorization

User-mandated or SIMD vectorization supplements automatic vectorization just like OpenMP parallelization supplements automatic parallelization. The following figure illustrates this relationship. User-mandated vectorization is implemented as a single-instruction-multiple-data (SIMD) feature and is referred to as SIMD vectorization.

NOTE
The SIMD vectorization feature is available for both Intel® microprocessors and non-Intel microprocessors. Vectorization may call library routines that can result in additional performance gain on Intel® microprocessors than on non-Intel microprocessors.



The following figure illustrates how SIMD vectorization is positioned among various approaches that you can take to generate vector code that exploits vector hardware capabilities. The programs written with SIMD vectorization are very similar to those written using auto-vectorization hints. You can use SIMD vectorization to minimize the number of code changes that you may have to go through in order to obtain vectorized code.



SIMD vectorization uses the `#pragma omp simd` pragma to effect loop vectorization.

Consider an example in C++ where the function `add_floats()` uses too many unknown pointers for the compiler's automatic runtime independence check optimization to kick in. You can give a data dependence assertion using the auto-vectorization hint via `#pragma ivdep` and let the compiler decide whether the auto-vectorization optimization should be applied to the loop. Or you can now enforce vectorization of this loop by using `#pragma omp simd` .

The difference between using `#pragma omp simd` and auto-vectorization hints is that with `#pragma omp simd`, the compiler generates a warning when it is unable to vectorize the loop. With auto-vectorization hints, actual vectorization is still under the discretion of the compiler, even when you use the `#pragma vector always` hint.

`#pragma omp simd` has optional clauses to guide the compiler on how vectorization must proceed. Use these clauses appropriately so that the compiler obtains enough information to generate correct vector code. For more information on the clauses, see the `#pragma omp simd` description.

Additional Semantics

Note the following points when using the omp simd pragma.

- A variable may belong to zero or one of the following: private, linear, or reduction.
- Within the vector loop, an expression is evaluated as a vector value if it is private, linear, reduction, or it has a sub-expression that is evaluated to a vector value. Otherwise, it is evaluated as a scalar value (that is, broadcast the same value to all iterations). Scalar value does not necessarily mean loop invariant, although that is the most frequently seen usage pattern of scalar value.
- A vector value may not be assigned to a scalar L-value. It is an error.
- A scalar L-value may not be assigned under a vector condition. It is an error.
- The `switch` statement is not supported.

NOTE
You may find it difficult to describe vector semantics using the SIMD pragma for some auto-vectorizable loops. One example is `MIN`/`MAX` reduction in C since the language does not have `MIN`/`MAX` operators.

Restrictions on Using a #pragma omp declare simd Declaration

Vectorization depends on two major factors: hardware and the style of source code. When using the vector declaration, the following features are not allowed:

- Thread creation and joining through , OpenMP `parallel/for/sections/task/target/teams`, and explicit threading API calls.
- Locks, barriers, atomic construct, critical sections (These are allowed inside `#pragma omp ordered simd` blocks).
- Inline ASM code, VM, and Vector Intrinsics (for example, SVML intrinsics).
- Using setjmp, longjmp, SHE and computed `GOTO`.
- EH is not allowed and all vector functions are considered noexcept.
- The `switch` statement (in some cases this may be supported and converted to `if` statements, but this is not reliable).
- The `exit()`/`abort()` calls.

Non-vector function calls are generally allowed within vector functions but calls to such functions are serialized lane-by-lane and so might perform poorly. Also for SIMD-enabled functions it is not allowed to have side effects except writes by their arguments. This rule can be violated by non-vector function calls, so be careful executing such calls in SIMD-enabled functions.

Formal parameters must be of the following data types:

- (un)signed 8, 16, 32, or 64-bit integer
- 32- or 64-bit floating point
- 64- or 128-bit complex
- A pointer (C++ reference is considered a pointer data type)

See Also

Function Annotations and the SIMD Directive for Vectorization

SIMD-enabled Functions

SIMD-enabled functions (formerly called elemental functions) are a general language construct to express a data parallel algorithm. A SIMD-enabled function is written as a regular C/C++ function, and the algorithm describes the operation on one element, using scalar syntax. The function can then be called as a regular C/C ++ function to operate on a single element or it can be called in a data parallel context to operate on many elements.

How SIMD-enabled Functions Work

When you write a SIMD-enabled function, the compiler generates short vector variants of the function that you requested, which can perform your function's operation on multiple arguments in a single invocation. The short vector variant may be able to perform multiple operations as fast as the regular implementation performs a single one by using the vector instruction set architecture (ISA) in the CPU. When a call to a SIMD-enabled function occurs in a SIMD loop or another SIMD-enabled function, the compiler replaces the scalar call with the best fit from the available short-vector variants of the function.

In addition, when invoked from a `pragma omp` construct, the compiler may assign different copies of the SIMD-enabled functions to different threads (or workers), executing them concurrently. The result is that your data parallel operation executes on the CPU using both the parallelism available in the multiple cores and the parallelism available in the vector ISA. In other words, if the short vector function is called inside a parallel loop, (a vectorized auto-parallelized loop) you can achieve both vector-level and thread-level parallelism.

Declare a SIMD-enabled Function

You need to use the appropriate syntax from below in your code for the compiler to generate the short vector function:

Linux

Use the `__attribute__((vector (clauses)))` declaration:

```
__attribute__((vector (clauses))) return_typesimd_enabled_function_name(parameters)
```

Alternately, you can use the following OpenMP pragma, which requires the `[q or Q]openmp` or `[q or Q]openmp-simd` compiler option:

```
#pragma omp declare simd clauses
```

Windows

Use the `__declspec(vector (clauses))` declaration:

```
__declspec(vector (clauses)) return_type simd_enabled_function_name(parameters)
```

The clauses in the vector declaration may be used for achieving better performance by overriding defaults. These clauses at SIMD-enabled function definition declare one or several short vector variants for a SIMD-enabled function. Multiple vector declarations with different set of clauses may be attached to one function in order to declare multiple different short vector variants available for a SIMD-enabled function.

The clauses are defined as follows:

Clause	Definition
processor(cpuid)	Tells the compiler to generate a vector variant using the instructions, the caller/callee interface, and the default vector length selection scheme suitable to the specified processor. Use of this clause is highly recommended, especially for processors with wider vector register support (example: `core_2nd_gen_avx` and newer). `cpuid` takes one of the following values: • `core_4th_gen_avx_tsx` • `core_4th_gen_avx` • `core_3rd_gen_avx` • `core_2nd_gen_avx` • `core_aes_pclmulqdq` • `core_i7_sse4_2` • `atom` • `core_2_duo_sse4_1` • `core_2_duo_ssse3` • `pentium_4_sse3` • `pentium_m` • `pentium_4` • `haswell` • `broadwell` • `skylake`

Clause	Definition
	• skylake_avx512 • knl • knm
vectorlength(n) / *simdlen(n)* (for omp declare simd)	Where n is a vector length that is a power of 2, no greater than 32. The *simdlen* clause tells the compiler that each routine invocation at the call site should execute the computation equivalent to n times the scalar function execution. When omitted the compiler selects the vector length automatically depending on the routine return value, parameters, and/or the processor clause. When multiple vector variants are called from one vectorization context (for example, two different functions called from the same vector loop), explicit use of identical *simdlen* values are advised to achieve good performance.
linear(list_item[, list_item...]), where *list_item* is one of: • *param[:step]* • *val(param[:step])* • *ref(param[:step])* • *uval(param[:step])*	The *linear* clause tells the compiler that for each consecutive invocation of the routine in a serial execution, the value of param is incremented by step, where *param* is a formal parameter of the specified function or the C++ keyword this. The linear clause can be used on parameters that are either scalar (non-arrays and of non-structured types), pointers, or C++ references. *step* is a compile-time integer constant expression, which defaults to 1 if omitted. If more than one step is specified for a particular parameter, a compile-time error occurs. Multiple linear clauses will be merged as a union. The meaning of each variant of the clause is as follows: • *linear(param[:step])*: For parameters that are not C++ references: the clause tells the compiler that on each iteration of the loop from which the routine is called the value of the parameter will be incremented by *step*. The clause can also be used for C++ references for backward compatibility, but it is not recommended. • *linear(val(param[:step]))*: For parameters that are C++ references: the clause tells the compiler that on each iteration of the loop from which the routine is called the referenced value of the parameter will be incremented by *step*. • *linear(uval(param[:step]))*: For C++ references: means the same as linear(val()). It differs from linear(val()) so if linear(val()) a

Clause	Definition
	vector of references is passed to vector variant of the routine but in case of linear(uval()) only one reference is passed (and thus linear(uval()) is better to use in terms of performance). • *linear(ref(param[:step]))* :For C++ references: means that the reference itself is linear, i.e. the referenced values (that form a vector for calculations) are located sequentially, like in array with the distance between elements equal to *step*.
uniform(param [, param,]…)	Where `param` is a formal parameter of the specified function or the C++ keyword `this`. The *uniform* clause tells the compiler that the values of the specified arguments can be broadcast to all iterations as a performance optimization. It is often useful in generating more favorable vector memory references. An acknowledgment of a *uniform* clause may allow broadcast operations to be hoisted out of the caller loop. Evaluate carefully the performance implications. Multiple uniform clauses are merged as a union.
mask / nomask	The *mask* and *nomask* clauses tell the compiler to generate only masked or unmasked (respectively) vector variants of the routine. When omitted, both masked and unmasked variants are generated. The masked variant is used when the routine is called conditionally.
inbranch / notinbranch	The *inbranch* and *notinbranch* clauses are used with `#pragma omp declare simd`. The *inbranch* clause works the same as the *mask* clause above and the *notinbranch* clause works the same as the *nomask* clause above.

Write the code inside your function using existing C/C++ syntax and relevant built-in functions (see the section on `__intel_simd_lane()` below).

Usage of Vector Function Specifications

You may define several vector variants for one routine with each variant reflecting a possible usage of the routine. Encountering a call, the compiler matches vector variants with actual parameter kinds and chooses the best match. Matching is done by priorities. In other words, if an actual parameter is the loop invariant and the `uniform` clause was specified for the corresponding formal parameter, then the variant with the `uniform` clause has a higher priority. Linear specifications have the following order, from high priority to low: `linear(uval())`, `linear()`, `linear(val())`, `linear(ref())`. Consider the following example loops with the calls to the same routine.

```
// routine prototype
#pragma omp declare simd                            // universal but slowest definition matches
the use in all three loops
#pragma omp declare simd linear(in1) linear(ref(in2)) uniform(mul) // matches the use in the
```

```
first loop
#pragma omp declare simd linear(ref(in2))                            // matches the use in the
second and the third loops
#pragma omp declare simd linear(ref(in2)) linear(mul)              // matches the use in the
second loop
#pragma omp declare simd linear(val(in2:2))                          // matches the use in the
third loop
extern int func(int* in1, int& in2, int mul);

int *a, *b, mul, *c;
int *ndx, nn;
...
// loop examples
   for (int i = 0; i < nn; i++) {
       c[i] = func(a + i, *(b + i), mul); // in the loop, the first parameter is changed
linearly,
                                          // the second reference is changed linearly too
                                          // the third parameter is not changed
   }

   for (int i = 0; i < nn; i++) {
       c[i] = func(&a[ndx[i]], b[i], i + 1); // the value of the first parameter is
unpredictable,
                                             // the second reference is changed linearly
                                             // the third parameter is changed linearly
   }

   #pragma omp simd
   for (int i = 0; i < nn; i++) {
       int k = i * 2;  // during vectorization, private variables are transformed into arrays: k-
>k_vec[vector_length]
       c[i] = func(&a[ndx[i]], k, b[i]); // the value of the first parameter is unpredictable,
                                         // the second reference and value can be considered
linear
                                         // the third parameter has unpredictable value
                                         // (the #pragma simd linear(val(in2:2))) will be chosen
from the two matching variants)
  }
```

SIMD-enabled Functions and C++

You should use SIMD-enabled functions in modern C++ with caution: C++ imposes strict requirements on compilation and execution environments that may not compose well with semantically-rich language extensions such as SIMD-enabled functions. There are three key aspects of C++ that interrelate with SIMD-enabled functions concept: exception handling, dynamic polymorphism, and the C++ type system.

SIMD-enabled Functions and Exception Handling

Exceptions are currently not supported in SIMD contexts: exceptions cannot be thrown and/or caught in SIMD loops and SIMD-enabled functions. Therefore, all SIMD-enabled functions are considered `noexcept` in C++11 terms. This affects not only short vector variants of a function, but its original scalar routine as well. This is enforced when the function is compiled: it is checked against throw construct and against function calls throwing exceptions. It is also enforced when the SIMD-enabled function call is compiled.

SIMD-enabled Functions and Dynamic Polymorphism

Vector attributes can be applied to virtual functions of classes with some limitations during polymorphic virtual function calls. The syntax of vector declarations is the same as for regular SIMD-enabled class methods: attach vector declarations as described above to the method declarations inside the class declaration.

Vector function attributes for virtual methods are inherited. If a vector attribute is specified for an overriding virtual function, it must match that of the overridden function. Even if the virtual method implementation is overridden in a derived class the vector declarations are inherited and applied. A set of vector variants is produced for the override according to vector variants set on parent. This rule also applies when the parent does not have any vector variants. If a virtual method is introduced as non-SIMD-enabled (no vector declarations supplied), it cannot become SIMD-enabled in the derived class even if the derived class contains its own implementation of the virtual method.

Matching vector variants for a virtual method is done by the declared (static) type of an object for which the method is called. The actual (dynamic) type of an object may either coincide with the static type or be inherited from it.

Unlike regular function calls which transfer control to one target function, the call target of a virtual function depends on the dynamic type of the object for which the method is called and accomplished indirectly via the virtual function table of a class. In a single SIMD chunk, the virtual method may be invoked for objects of multiple classes, for example, elements of a polymorphic collection. This requires multiple calls to different targets within a single SIMD chunk. This works as follows:

1. If a SIMD-enabled virtual function call is matched to a variant with a uniform *this* parameter, multiple calls are not needed. The compiler makes an indirect call to the matched vector variant of a virtual method of the object's dynamic class.
2. If a SIMD-enabled virtual function call is matched to a variant with a non-uniform *this* parameter, all objects in a SIMD chunk may still share virtual method implementation. This is checked and a single, indirect call to the matched vector variant of the target virtual method implementation is invoked.
3. Otherwise, lanes sharing virtual call targets are masked-in and a masked vector variant corresponding to the match is invoked in a loop for each unique virtual call target. If a masked variant is not provided for matching a vector variant and a *this* parameter is not declared uniform, the match will be rejected.

The following example illustrates SIMD-enabled virtual functions:

```
struct Base {
#pragma omp declare simd
#pragma omp declare simd uniform(this)
   virtual int process(int);
};

struct Child1 : Base {
   // int process(int); is inherited
};

struct Child11 : Child1 {
  int process(int); // Overrides implementation, inherits vector declarations
};

struct Child2 : Base {
  int process(int); // Overrides implementation, inherits vector declarations
};

int main() {
    int arr[100];
    Base* c2 = new Child2();
    Base* objs[100];
    int res = 0;
```

```
// SIMD-enabled virtual function call for uniform object
#pragma omp simd reduction(+:res)
    for (int i = 0; i < 100; i++) {
      res += c2->process(arr[i]);  // Variant with uniform this is matched
                                   // call to vector variant of
                                   // Child2::process() is invoked
   }

// Initialize polymorphic array of objects
    for (int i = 0; i < 100; i++) {
       if (i % 16 < 4) objs[i] = new Base();
       else if (i % 16 < 8) objs[i] = new Child1();
       else if (i % 12 < 12) objs[i] = new Child11();
       else objs[i] = new Child2();
   }

// SIMD-enabled virtual function call for non-uniform objects
#pragma omp simd reduction(+:res) simdlen(8)
    for (int i = 0; i < 100; i++) {
       res += objs[i]->process(arr[i]);  // Variant with non-uniform this is
                                         // matched
       // Base and Child1 share the same 'process' implementation, so call
       // targets for each even chunk [i*16:i*16+7] are the same even though
       // this pointers are different for all elements of objs[] array.

       // Odd chunks [i*16+8:i*16+15] consist of objects of classes Child11
       // and Child2 and so require calls to their respective implementations
       // of process() virtual functions. Masked vector variant for
       // Child11::process() is called with mask 0b00001111 (lower lanes of a
       // chunk) and masked vector variant for Child2::process() is called
       // with mask 0b11110000 (upper lanes of a chunk).
   }

   return res;
}
```

The following are limitations to SIMD-enabled virtual function support:

- Multiple inheritance, including virtual inheritance, is not supported for classes having SIMD-enabled virtual methods. This is because calls to virtual functions in multiple inheritance cases may be done through special functions called thunks, which adjust the 'this' pointer and/or virtual function table pointer. The current implementation doesn't support thunks for SIMD-enabled virtual calls because in this case thunks should themselves become SIMD-enabled functions that are not implemented.
- It is not possible to get the address of a SIMD-enabled virtual method. Support of SIMD-enabled virtual functions would require additional information, so their binary representation is different. Such cases will not be handled properly by code expecting a regular pointer to the virtual member.

SIMD-enabled Functions and the C++ Type System

Vector attributes are attributes in the C++11 sense and so are not part of a functional type of SIMD-enabled functions. Vector attributes are bound to the function itself, an instance of a functional type. This has the following implications:

- Template instantiations having SIMD-enabled functions as template parameters won't catch vector attributes, so it is impossible to preserve vector attributes in function wrapper templates like `std::bind` which add indirection. This indirection may sometimes be optimized away by compiler and the resulting direct call will have all vector attributes associated.

- There is no way to overload or specialize templates by vector attributes.
- There is no way to write functional traits to capture vector attributes for the sake of template metaprogramming.

The example below depicts various situations where this situation may be observed:

```
template <int f(int)>   // Function value template - captures exact function
                        // not a function type
int caller1(int x[100]) {
   int res = 0;
#pragma omp simd reduction(+:res)
   for (int i = 0; i < 100; i++) {
      res += f(x[i]);   // Exact function put here upon instantiation
   }
   return res;
}

template <typename F>  // Generic functional type template - captures
                       // object type for functors or entire functional type
                       // for functions. If vector attributes were part of
                       // a functional type they might be captured and applied
                       // but currently they are not.
int caller2(F f, int x[100]) {
   int res = 0;
#pragma omp simd reduction(+:res)
   for (int i = 0; i < 100; i++) {
      res += f(x[i]);  // Will call matching function 'f' indirectly
                       // Will call matching f.operator() directly
   }
   return res;
}

template <typename RET, typename ARG>  // Type-decomposing template
                                       // captures argument and return types.
                                       // Vector attributes would be lost
                                       // even if they were part of a
                                       // functional type.
int caller3(RET (*f)(ARG), int x[100]) {
   int res = 0;
#pragma omp simd reduction(+:res)
   for (int i = 0; i < 100; i++) {
      res += f(x[i]);  // Will call matching function 'f' indirectly
   }
   return res;
}


#pragma omp declare simd
int function(int x); // SIMD-enabled function
int nv_function(int x);                 // Regular scalar function

struct functor {                        // Functor class with
#pragma omp declare simd                      // SIMD-enabled operator()
   int operator()(int x);
};

int arr[100];

int main() {
```

```
   int res;
#pragma noinline
   res = caller1<function>(arr); // This will be instantiated for
                                 // function() and call short vector variant
#pragma noinline
   res += caller1<nv_function>(arr); // This will be separately instantiated
                                     // for nv_function()
#pragma noinline
   res += caller2(function, arr); // This will be instantiated for
                                  // int(*)(int) type and will call scalar
                                  // function() indirectly
#pragma noinline
   res += caller2(nv_function, arr); // This will call the same
                                     // instantiation as above on nv_function

#pragma noinline
   res += caller2(functor(), arr); // This will be instantiated for
                                   // functor type and will call short vector
                                   // variant of functor::operator()
#pragma noinline
   res += caller3(function, arr); // This will be instantiated for
                                  // <int, int> types and will call scalar
                                  // function() indirectly
#pragma noinline
   res += caller3(nv_function, arr); // This will call the same
                                     // instantiation as above on nv_function
   return res;
}
```

NOTE If calls to `caller1`, `caller2` and `caller3` are inlined, the compiler is able to replace indirect calls by direct calls in all cases. In this case `caller2(function, arr)` and `caller3(function, arr)` both call short vector variants of a function as result of the usual replacement of direct calls to function() by matching short vector variants in the SIMD loop.

Invoke a SIMD-enabled Function with Parallel Context

Typically, the invocation of a SIMD-enabled function provides arrays wherever scalar arguments are specified as formal parameters.

The following two invocations will give instruction-level parallelism by having the compiler issue special vector instructions.

```
a[:] = ef_add(b[:],c[:]);    //operates on the whole extent of the arrays a, b, c
```

```
a[0:n:s] = ef_add(b[0:n:s],c[0:n:s]); //use the full array notation construct to also specify n
as an extend and s as a stride
```

NOTE The array notation syntax, as well as calling the SIMD-enabled function from the regular `for` loop, results in invoking the short vector function in each iteration and using the vector parallelism but the invocation is done in a serial loop, without using multiple cores.
Use of array notation syntax and SIMD-enabled functions in a regular `for` loop results in invoking the short vector function in each iteration and using the vector parallelism, but the invocation is done in a serial loop without using multiple cores.

Use the __intel_simd_lane() Built-in Function

When called from within a vectorized loop, the `__intel_simd_lane()` built-in function will return a number between 0 and *vectorlength* - 1 that reflects the current "lane id" within the SIMD vector. `__intel_simd_lane()` will return zero if the loop is not vectorized. Calling `__intel_simd_lane()` outside of an explicit vector programming construct is discouraged. It may prevent auto-vectorization and such a call often results in the function returning zero instead of a value between 0 and *vectorlength*-1.

To see how `__intel_simd_lane()` can be used, consider the following example:

```
void accumulate(float *a, float *b, float *c, d){
  *a+=sin(d);
  *b+=cos(d);
  *c+=log(d);
}

for (i=low; i<high; i++){
    accumulate(&suma, &sumb, &sumc, d[i]);
}
```

The gather-scatter type memory addressing caused by the references to arrays A, B, and C in the SIMD-enabled function `accumulate()` will significantly hurt performance making the whole conversion useless. To avoid this penalty, you may use the `__intel_simd_lane()` built-in function as follows:

```
#pragma omp declare simd uniform(a,b,c) aligned(a,b,c)
void accumulate(float *a, float *b, float *c, float d){
// No need to take "loop index". No need to know VL.
  a[__intel_simd_lane()]+=sin(d);
  b[__intel_simd_lane()]+=cos(d);
  c[__intel_simd_lane()]+=log(d);
}

#define VL 16 // actual SIMD code may use vectorlength of 4 but it's okay.
float a[VL] = {0.0f};
float b[VL] = {0.0f};
float c[VL] = {0.0f};
#pragma omp simd for simdlen(VL)
for (i=low; i<high; i++){
    // If low is known to be zero at compile time, "i & (VL-1)"
    // would accomplish what __intel_simd_lane() is intended for,
    // but only on the caller side.
    accumulate(a, b, c, d[i]);
}
for(i=0;i<VL;i++){
    suma += a[i];
    sumb += b[i];
    sumc += c[i];
}
```

With the use of `__intel_simd_lane()`, the references to the arrays in `accumulate()` will have unit-stride.

Limitations

The following language constructs are not allowed within SIMD-enabled functions:

- The `GOTO` statement.
- The `switch` statement with 16 or more `case` statements.
- Operations on `classes` and `structs` (other than member selection).
- Any OpenMP construct.

See Also

User-Mandated or SIMD Vectorization

Function Annotations and the SIMD Directive for Vectorization
SIMD-Enabled Function Pointers

SIMD-enabled Function Pointers

SIMD-enabled functions (formerly called elemental functions) are a general language construct to express a data parallel algorithm. A SIMD-enabled function is written as a regular C/C++ function, and the algorithm within describes the operation on one element, using scalar syntax. The function can then be called as a regular C/C++ function to operate on a single element or it can be called in a data parallel context to operate on many elements.

In some cases it is desirable to have a pointer for SIMD-enabled functions, but without special effort, the vector nature of a function will be lost: function pointers will point to the scalar function and there will be no way to call the short vector variants existing for this scalar function.

In order to support indirect calls to vector variants of SIMD-enabled functions, SIMD-enabled function pointers were introduced. A SIMD-enabled function pointer is a special kind of pointer incompatible with a regular function pointer. They refer to an entire set of short vector variants as well as the scalar function. This incompatibility incurs the risk of inappropriate misuse, especially in C++ code. Therefore vector function pointer support is disabled by default.

How SIMD-enabled Function Pointers Work

When you write a SIMD-enabled function, the compiler generates short vector variants of the function that you requested, which can perform your function's operation on multiple arguments in a single invocation. The short vector variants may be able to perform multiple operations as fast as the regular implementation performs just one such operation by utilizing the vector instruction set architecture (ISA) in the CPU. When a call to SIMD-enabled function occurs in a SIMD loop or another SIMD-enabled function, the compiler replaces the scalar call with the best fit short vector variant of the function among those available.

Indirect SIMD-enabled function calls are handled similarly, but the set of available variants should be associated with the function pointer variable, not the target function, because actual call targets are unknown at the indirect call. That means all SIMD-enabled functions to be referenced by a SIMD-enabled function pointer should have a set of variants that match the set of variants declared for the pointer.

Declare a SIMD-enabled Function Pointer Variable

In order for the compiler to generate a pointer to a SIMD-enabled function, you need to provide an indication in your code.

Linux

Use the `__attribute__((vector (clauses)))` attribute, as follows:

```
__attribute__((vector (clauses))) return_type (*function_pointer_name) (parameters)
```

Alternately, you can use OpenMP `#pragma omp declare simd`, which requires the `[q or Q]openmp` or `[q or Q]openmp-simd` compiler option.

Windows

Use the `__declspec(vector (clauses))` attribute, as follows:

```
__declspec(vector (clauses)) return_type (*function_pointer_name) (parameters)
```

The clauses are described in the previous topic on SIMD-enabled functions.

Usage of Vector Function Attributes on Pointers

You may associate several vector attributes with one SIMD-enabled function pointer which reflects all the variants available for the target functions to be called through the pointer. The attributes usually reflect a possible use of the function pointer in the loops. Encountering an indirect call, the compiler matches the vector variants declared on the function pointer with the actual parameter kinds and chooses the best match. Matching is done exactly the same way as with direct calls (see the previous topic on SIMD-enabled functions). Consider the following example of the declaration of vector function pointers and loops with indirect calls.

```
// pointer declaration
#pragma omp declare simd                           // universal but slowest definition matches
the use in all three loops
#pragma omp declare simd linear(in1) linear(ref(in2)) uniform(mul)   // matches the use in the
first loop
#pragma omp declare simd linear(ref(in2))                            // matches the use in the
second and the third loops
#pragma omp declare simd linear(ref(in2)) linear(mul)                // matches the use in the
second loop
#pragma omp declare simd linear(val(in2:2))                          // matches the use in the
third loop
int (*func)(int* in1, int& in2, int mul);

int *a, *b, mul, *c;
int *ndx, nn;
...
// loop examples
   for (int i = 0; i < nn; i++) {
       c[i] = func(a + i, *(b + i), mul); // in the loop, the first parameter is changed
linearly,
                                          // the second reference is changed linearly too
                                          // the third parameter is not changed
   }

   for (int i = 0; i < nn; i++) {
       c[i] = func(&a[ndx[i]], b[i], i + 1); // the value of the first parameter is
unpredictable,
                                             // the second reference is changed linearly
                                             // the third parameter is changed linearly
   }

   #pragma omp simd
   for (int i = 0; i < nn; i++) {
       int k = i * 2;  // during vectorization, private variables are transformed into arrays: k-
>k_vec[vector_length]
       c[i] = func(&a[ndx[i]], k, b[i]); // the value of the first parameter is unpredictable,
                                         // the second reference and value can be considered
linear
                                         // the third parameter has unpredictable value
                                         // (the __declspec(vector(linear(val(in2:2)))) will be
chosen from the two matching variants)
   }
```

Before any use in a call, the function pointer should be assigned either the address of a function or another function pointer. Just as with function pointers, vector function pointers should be compatible at assignment and initialization. The compatibility rules are described below.

Vector Function Pointer Compatibility

Pointer assignment compatibility is defined as following:

1. If a SIMD-enabled function pointer is assigned the address of a function, the function should be compatible with the pointer in the usual C/C++ sense, it should be SIMD-enabled, and the set of vector variants declared for the function should be a superset of those declared for the pointer. This includes initializations and passing addresses of SIMD-enabled functions as parameters.
2. If a SIMD-enabled function pointer is assigned another function pointer, the source pointer should be compatible with the destination function pointer in the general C/C++ sense, it should be SIMD-enabled, and the set of vector variants declared for the source pointer should be exactly the same as those declared for destination pointer. This includes initializations and passing SIMD-enabled function pointers as parameters.
3. If a regular (non-SIMD-enabled) function pointer is assigned the address of a SIMD-enabled function, the address of a scalar function is assigned. Vector variants cannot be called through the pointer and it cannot be reinterpreted as or converted into a SIMD-enabled function pointer as discussed in rule 2.
4. If a regular (non-SIMD-enabled) function pointer is assigned a SIMD-enabled function pointer matching in the C/C++ sense, the implicit dynamic casting of the right-hand side of the assignment (RHS) is performed by extracting the address of a scalar function and this address is assigned. Vector variants cannot be called through these pointers and it cannot be reinterpreted as or converted into a SIMD-enabled function pointer as discussed in rule 2.

NOTE
SIMD-enabled function pointers and regular function pointers are binary-incompatible and handled differently. Mixing them may lead to severe unpredictable results. The compiler does its best to check compatibility where it is allowed by C/C++ language standards, but in certain cases it cannot check, such as passing function pointers to undeclared functions or as variable arguments. It is best to refrain from using SIMD-enabled function pointers in these contexts. Additional complexities with respect to the C++ type system are described in the *SIMD-enabled Function Pointers and the C++ Type System* section below.

A SIMD-enabled function pointer may be assigned to a scalar function pointer with a cast as described in rule 4 above, but a SIMD-enabled function pointer cannot refer to a scalar function pointer.

```
// pointer declarations
#pragma omp declare simd
int (*ptr1)(int*, int);
#pragma omp declare simd
int (*ptr1a)(int*, int);

#pragma omp declare simd
#pragma omp declare simd linear(a)
typedef int (*fptr_t2)(int* a, int b);

typedef int (*fptr_t3)(int*, int);

fptr_t2 ptr2, ptr2a;
fptr_t3 ptr3;

// function declarations
#pragma omp declare simd
int func1(int* x, int b);

#pragma omp declare simd
#pragma omp declare simd linear(x)
int func2(int* x, int b);

#pragma omp declare simd
#pragma omp declare simd linear(x)
```

```
int func3(float* x, int b);

//--------------------------------------
  // allowed assignments
  ptr1 = func1;  // same prototype and vector spec
  ptr2 = func2;  // same prototype and vector spec
  ptr1a = ptr1;  // same prototype and vector spec
  ptr1a = func2; // same prototype vector spec on function includes all vector spec on pointer

  ptr3 = func1; // scalar pointer with same prototype - use scalar func1
  ptr3 = func2; // scalar pointer with same prototype - use scalar func2
  ptr3 = ptr1;  // scalar pointer with same prototype - implicit conversion from vector to
scalar pointer
  ptr3 = ptr2;  // scalar pointer with same prototype - implicit conversion from vector to
scalar pointer

  // disallowed assignments
  ptr2 = func1; // vector spec on function does not have all specs on pointer
  ptr2 = func3; // prototype mismatch although vector spec matched
  ptr1 = func3; // prototype mismatch although vector spec matched
  ptr3 = func3; // prototype mismatch
  ptr1 = ptr2;  // pointers should have the same vector spec
  ptr2 = ptr3;  // pointers should have the same vector spec
```

Call Sequence

Unlike regular function calls, which transfer control to a target function, the call target of an indirect call depends on the dynamic content of the function pointer. In a loop, call targets may be different on different iterations of a vectorized loop or on different lanes of a SIMD-enabled function executing the call. When vectorized, such an indirect call may involve multiple calls to different targets within a single SIMD chunk. This works as follows:

1. If the vector function pointer is uniform (refer to the OpenMP specification) or if it can be determined to be uniform by the compiler, then multiple calls are not needed. The compiler makes a single indirect call to a matched vector variant accessible by the pointer.
2. If the vector function pointer is not known to be uniform at compile time, all values of the pointer in a SIMD chunk may still be the same. This is checked at run time and a single indirect call to a matched vector variant is invoked.
3. Otherwise, lanes sharing the same function pointer value (call target) are masked-in and a masked vector variant corresponding to the matched one is invoked in the loop for each unique call target. If the masked variant is not provided for the matching vector variant and the function pointer is not proven to be uniform by compiler the match will be rejected and the compiler may serialize the call, or in other words, generate several scalar calls.

```
// pointer typedefs
#pragma omp declare simd
typedef int (*fptr_t1)(int*, int);

// function declarations
#pragma omp declare simd
int func1(int* x, int b);

// uses of vector function pointers
fptr_t1 *fptr_array;   // array of vector function pointers
void foo(int N, int *x, int y){
  fptr_t1 ptr1 = func1;
#pragma omp simd
  for (int i = 0; i < N; i++) {
```

```
    ptr1(x+i, y);  // ptr1 is uniform by OpenMP rule.
    fptr_t1 ptr1a = ptr1;
    ptr1a(x+i, y); // compiler can prove ptr1a is uniform.
    fptr_t1 ptr1b = fptr_array[i];
    ptr1b(x+i,y);  // ptr1b may or may not be uniform.
  }
}
```

SIMD-enabled Function Pointers and the C++ Type System

Use caution when using SIMD-enabled function pointers in modern C++: C++ imposes strict requirements on compilation and execution environments which may not compose well with semantically-rich language extensions such as SIMD-enabled function pointers. Vector specifications on SIMD-enabled function pointers are attributes in C++11 sense and so are not part of a pointer type even though they make that pointer binary incompatible with another pointer of the same type but without the attribute. Vector specifications are not bound to a pointer type, but instead are bound to the variable or function argument (which is an instance of a pointer type) itself. For a given function pointer, the type of the pointer is the same with or without SIMD-enabled function pointer decoration. This has the following important implications:

- Vector attributes put on a function argument are not reflected in C++ name mangling, so the functions differ only in the vector attributes of a functional pointer argument (or lack thereof) will have the same name and will be treated the same by the C++ linker. This may result in a parameter of incorrect vectorness (having the vector attribute or not) being passed into the function. In some cases there is no way for the compiler to detect this situation, so you're strongly encouraged to distinctly name functions having SIMD-enabled function pointers as parameters.
- The incorrect interpretation of function pointers is extremely dangerous because it may lead to the execution of unwanted code or non-code. To identify these situations the compiler issues the following warning if a vector function pointer is used as a C++ function parameter: Warning #3757: this use of a vector function type is not fully supported. If you are sure that no ambiguity is possible—for example, the function accepting the vector function pointer has a distinct name and is fully declared before all uses—you may ignore this warning. Otherwise, ensure that no ambiguity is possible.
- Template instantiations having SIMD-enabled pointer types as template parameters won't catch vector attributes. The template will be instantiated a parameter matching the non-SIMD-enabled pointer type. All variables, class members, and function arguments bound to the template argument type will be regular function pointers. The use of such templates with a SIMD-enabled function pointer as a template function parameter, template class method parameter, or RHS of template class member assignment will lead to a dynamic cast to the non-SIMD-enabled function pointer and loss of vectorness.
- There is no way to overload or achieve template specialization by the vector attributes of a functional pointer
- There is no way to write functional traits to capture vector attributes for the sake of template metaprogramming.

```
// pointer typedefs and pointer declarations
typedef int
(*fptr_t)(int*, int);

#pragma omp declare simd
typedef int (*fptr_t1)(int*, int);

#pragma omp declare simd
#pragma omp declare simd linear(x)
typedef int (*fptr_t2)(int* a, int b);

fptr_t ptr
fptr_t1 ptr1
fptr_t2 ptr2
```

```
// function prototype that only differs in SIMD-enabled function decoration
// All these will have identical mangled names.
void foo(fptr_t);
void foo(fptr_t1);
void foo(fptr_t2);

// template instantiation
template <typename T>
void bar(T);
…
  bar(fptr);          // bar<fptr_t>
  bar(fptr1);         // bar<fptr_t>
  bar(fptr2);         // bar<fptr_t>
```

Indirect Invocation of a SIMD-enabled Function with Parallel Context

Typically, the invocation of a SIMD-enabled function directly or indirectly provides arrays wherever scalar arguments are specified as formal parameters.

The following invocations will give instruction-level parallelism by having the compiler issue special vector instructions.

```
#pragma omp declare simd
float (**vf_ptr)(float, float);

//operates on the whole extent of the arrays a, b, c
a[:] = vf_ptr[:] (b[:],c[:]);

// use the full array notation construct to also specify n
// as an extend and s as a stride
a[0:n:s] = vf_ptr[0:n:s] (b[0:n:s],c[0:n:s]);
```

NOTE The array notation syntax, as well as calling the SIMD-enabled function from the regular `for` loop, results in invoking the short vector variant in each iteration and utilizing the vector parallelism but the invocation is done in a serial loop, without utilizing multiple cores.

See Also

User-mandated or SIMD Vectorization
Function Annotations and the SIMD Directive for Vectorization
SIMD-enabled functions

Vectorize a Loop Using the _Simd Keyword

In this section we introduce the `_Simd` keyword, which provides an alternative to the `simd` pragma. Just like the `simd` pragma, the `_Simd` keyword modifies a serial `for` loop for vectorization. The syntax is as follows:

```
_Simd [_Safelen(constant-expression)][_Reduction (reduction-identifier : list)]
```

The `_Simd` keyword and any clauses should come after the `for` keyword as in this example:

```
for _Simd (int i=0; i<10; i++){
  // loop body
}
```

Differences between the `simd` pragma and `_Simd` keyword:

- Omission of the `private` and `lastprivate` clauses of the `simd` pragma construct because C and C++ already have variable-scoping rules that allow a programmer to cleanly declare a private variable within the scope of a loop iteration
- The `linear` clause is omitted because the ability to increment multiple variables makes it unnecessary. See the following example:

```
float add_floats(float *a, float *b, int n){
  int i=0;
  int j=0;
  float sum=0;

  for _Simd _Reduction(+:sum) (i=0; i<n; i++, j+=2){
    a[i] = a[i] + b[j];
    sum += a[i];
  }
  return sum;
}
```

To ensure that your loop is vectorized keep the following in mind:

- The countable loop for the _Simd keyword has to conform to the for-loop style of an OpenMP* canonical loop form except that multiple variables may be incremented in the incr-expr (See the OpenMP* specification at www.openmp.org).
- The loop control variable must be a signed integer type.
- The vector values should be signed 8-, 16-, 32-, or 64-bit integers, single or double-precision floating point numbers, or single or double-precision complex numbers.
- You cannot use any control constructs to jump into or out of a SIMD loop. That includes the `break`, `return`, `goto`, and `throw` constructs.
- A SIMD loop may contain another loop (`for`, `while`, `do-while`) in it, but `goto` out of such inner loops is not supported. You may use `break` and `continue` with the inner loop.
- A SIMD loop performs memory references unconditionally. Therefore, all address computations must result in valid memory addresses, even though such locations may not be accessed if the loop is executed sequentially

See Also

User-mandated or SIMD Vectorization

Function Annotations and the SIMD Directive for Vectorization

This topic presents specific C++ language features that better help to vectorize code.

> **NOTE**
> The SIMD vectorization feature is available for both Intel® microprocessors and non-Intel microprocessors. Vectorization may call library routines that can result in additional performance gain on Intel® microprocessors than on non-Intel microprocessors.

The `__declspec(align(`*n*`))` declaration enables you to overcome hardware alignment constraints. The auto-vectorization hints address the stylistic issues due to lexical scope, data dependency, and ambiguity resolution. The SIMD feature's pragma allows you to enforce vectorization of loops.

You can use the `__declspec(vector)` `__attribute__(vector)` and the `__declspec(vector[clauses])` `__attribute__(vector(`*clauses*`))` declarations to vectorize user-defined functions and loops. For SIMD usage, the `vector` function is called from a loop that is being vectorized.

The C/C++ extensions for Array Notations `map` operations can be defined to provide general data parallel semantics, where you do not express the implementation strategy. You can write the same operation regardless of the size of the problem. The implementation uses the construct by combining SIMD, loops and tasking to implement the operation. With these semantics, you can choose more elaborate programming and express a single dimensional operation at two levels. You can use both task constructs and array operations to force a preferred parallel and vector execution.

The usage model of the `vector` declaration takes a small section of code generated for the function (*vectorlength*) of the array and exploits SIMD parallelism. The implementation of task parallelism is done at the call site.

The following table summarizes the language features that help vectorize your code:

Language Feature	Description
`__declspec(align(n))`	Directs the compiler to align the variable to an *n*-byte boundary. Address of the variable is *address* `mod n=0`.
`__declspec(align(n,off))`	Directs the compiler to align the variable to an *n*-byte boundary with offset off within each *n*-byte boundary. Address of the variable is *address* `mod n=off`.
`__declspec(vector)` (Windows) `__attribute__(vector)` (Linux)	Combines with the `map` operation at the call site to provide the data parallel semantics. When multiple instances of the vector declaration are invoked in a parallel context, the execution order among them is not sequenced.
`__declspec(vector[clauses])` (Windows) `__attribute__(vector(clauses))` (Linux	Combines with the `map` operation at the call site to provide the data parallel semantics with the following values for *clauses*: • processor clause: *processor(cpuid)* • vector length clause: *vectorlength(n)* • linear clause: *linear(param1:step1 [, param2:step2]…)* • uniform clause: *uniform(param [, param,]…)* • mask clause: *[no]mask* When multiple instances of the vector declaration are invoked in a parallel context, the execution order among them is not sequenced.
`restrict`	Permits the disambiguator flexibility in alias assumptions, which enables more vectorization.
`__declspec(vector_variant(clauses))` (Windows) `__attribute__(vector_variant(clauses))` (Linux)	Provides the ability to vectorize user-defined functions and loops. The *clauses* are as follows: • implements clause (required): *implements (function declarator) [, simd-clauses])* • simd-clauses (optional): one or more of the clauses allowed for the vector attribute
`__assume_aligned(a,n)`	Instructs the compiler to assume that array *a* is aligned on an *n*-byte boundary; used in cases where the compiler has failed to obtain alignment information.
`__assume(cond)`	Instructs the compiler to assume that the represented condition is true where the keyword appears. Typically used for conveying properties that the compiler can take advantage of for generating more efficient code, such as alignment information.

The following table summarizes the auto-vectorization hints that help vectorize your code:

Hint	Description
`#pragma ivdep`	Instructs the compiler to ignore assumed vector dependencies.
`#pragma vector {aligned|unaligned|always|temporal|nontemporal}`	Specifies how to vectorize the loop and indicates that efficiency heuristics should be ignored. Using the `assert` keyword with the `vector {always}` pragma generates an error-level assertion message if the compiler efficiency heuristics indicate that the loop cannot be vectorized. Use `#pragma ivdep!` to ignore the assumed dependencies.
`#pragma novector`	Specifies that the loop should never be vectorized.

NOTE
Some pragmas are available for both Intel® microprocessors and non-Intel microprocessors, but may perform additional optimizations for Intel® microprocessors than for non-Intel microprocessors.

The following table summarizes the user-mandated pragmas that help vectorize your code:

User-Mandated Pragma	Description
`#pragma simd`	Enforces vectorization of loops.
omp simd	Transforms the loop into a loop that will be executed concurrently using SIMD instructions.

See Also

__declspec(align) declaration

ivdep pragma
vector pragma
SIMD-enabled functions
User-mandated or SIMD Vectorization

Explicit SIMD SYCL Extension

oneAPI provides an Explicit SIMD SYCL extension (ESIMD) for lower-level Intel GPU programming.

ESIMD provides APIs that are similar to Intel's GPU Instruction Set Architecture (ISA), but it enables you to write explicitly vectorized device code. This explicit enabling gives you more control over the generated code and allows you to depend less on compiler optimizations.

The specification, API reference, and working code examples are available on GitHub.

NOTE Some parts of this extension are under active development and the APIs in the `sycl::ext::intel::experimental::esimd` package are subject to change. The restrictions are specified below.

ESIMD kernels and functions always require a subgroup size of one, which means that the compiler does not provide vectorization across work items in a subgroup. Instead, you must explicitly express the vectorization in your code. Below is an example that adds the elements of two arrays and writes the results to the third:

```
float *A = malloc_shared<float>(Size, q);
float *B = malloc_shared<float>(Size, q);
float *C = malloc_shared<float>(Size, q);

for (unsigned i = 0; i != Size; i++) {
  A[i] = B[i] = i;
}

q.submit([&](handler &cgh) {
  cgh.parallel_for<class Test>(
    Size / VL, [=](id<1> i)[[intel::sycl_explicit_simd]] {
    auto offset = i * VL;
    // pointer arithmetic, so offset is in elements:
    simd<float, VL> va(A + offset);
    simd<float, VL> vb(B + offset);
    simd<float, VL> vc = va + vb;
    vc.copy_to(C + offset);
  });
}).wait_and_throw();
```

In the example above, the lambda function passed to the `parallel_for` is marked with a special attribute: `[[intel::sycl_explicit_simd]]`. This attribute tells the compiler that the kernel is ESIMD-based and ESIMD APIs can be used inside it. Here the `simd` objects and `copy_to` intrinsics are used. They are available only in the ESIMD extension.

Fully runnable code samples can be found on GitHub.

Compile and Run ESIMD Code

Code that uses the ESIMD extension can be compiled and run using the same commands as you would with standard SYCL:

To compile using the open-source oneAPI DPC++ Compiler:

```
clang++ -fsycl vadd_usm.cpp
```

To compile using an Intel® oneAPI Toolkit:

```
dpcpp vadd_usm.cpp
```

To run on an Intel specific GPU device, through the oneAPI Level Zero (Level Zero) backend:

```
SYCL_DEVICE_FILTER=level_zero:gpu ./a.out
```

The resulting executable (`$./a.out`) can be run only on Intel GPU hardware, such as Intel® UHD Graphics 600 or later. The SYCL runtime automatically recognizes ESIMD kernels and dispatches their execution, so no additional setup is needed. Both Linux and Windows platforms are supported, including OpenCL™ and Level Zero backends.

ESIMD Emulator

The ESIMD emulator (`ESIMD_EMULATOR`) provides a feature to execute ESIMD kernels on the host CPU without having an Intel GPU device in the system. It provides you with a way to debug ESIMD code in any debugger. Since the emulator tries to model massively parallel GPU kernel execution on CPU hardware, some differences in your execution profile may happen. Take this into account when debugging. You can redirect execution to the ESIMD emulator by setting an environment variable, no program recompilation is needed. When running a kernel via the emulator, the SYCL runtime sees the emulator as a normal GPU device (example, an `is_gpu()` test will return true for it).

Due to the specifics of ESIMD programming model, a standard SYCL host device cannot execute ESIMD kernels and needs supporting libraries to emulate barriers and GPU execution threads. It is impractical for the host part of a SYCL ESIMD app to include or link to all the necessary infrastructure components when there is no ESIMD code, or if debugging is not wanted. It is inconvenient or not possible for you to recompile the app with a switch to execute the ESIMD part on a CPU. The environment variable plus a separate back-end solves both problems.

The ESIMD emulator includes these components:

- The ESIMD emulator plugin, which is a SYCL runtime back-end similar to OpenCL™ or Level Zero.
- Host implementations of low-level ESIMD intrinsics, for example `__esimd_scatter_scaled`.
- The supporting infrastructure linked dynamically to the plugin, for example the `libCM` library.

ESIMD Emulator Requirements

The ESIMD emulator backend uses a CM_EMU library for emulating GPUs using software multi-threading. The library is provided as separate pre-installed library in host machine, or built as part of the open-source oneAPI DPC++ Compiler. The required version for CM_EMU is 1.0.20 or later. To add the CM_EMU library as part of oneAPI DPC++ Compiler for ESIMD emulator backend, build the library during ESIMD emulator plug-in software module generation. Details on building CM_EMU library for ESIMD emulator, including required packages are described in ESIMD CPU Emulation.

Command Line/Environment Variable Options

There are no special command line options or environment variables required for building and running ESIMD kernels with the ESIMD emulator backend.

Running ESIMD Code in Emulation Mode

The compilation step for ESIMD kernels that are prepared for an ESIMD emulator backend is same as for OpenCL and Level Zero backends. The fully runnable code sample, and other samples, used below can be found on Github.

To compile using the open-source oneAPI DPC++ Compiler, use:

```
clang++ -fsycl vadd_usm.cpp
```

To compile using the Intel® oneAPI Toolkits, use:

```
 dpcpp vadd_usm.cpp
```

To run under emulation through the ESIMD emulator backend, use:

```
SYCL_DEVICE_FILTER=ext_intel_esimd_emulator:gpu ./a.out
```

Code Sample

```
# Get sources
git clone https://github.com/intel/llvm-test-suite
cd llvm-test-suite
mkdir build && cd build

# Configure for make utility with compiler tools available in $PATH
cmake \
 -DCMAKE_CXX_COMPILER=clang++ \
 -DTEST_SUITE_SUBDIRS=SYCL \
 -DSYCL_BE="ext_intel_esimd_emulator" \
 -DSYCL_TARGET_DEVICES="gpu" \
 ..

# Build and Run
make check

# Or, for Ninja utility
```

```
cmake -G Ninja \
 -DCMAKE_CXX_COMPILER=clang++ \
 -DTEST_SUITE_SUBDIRS=SYCL \
 -DSYCL_BE="ext_intel_esimd_emulator" \
 -DSYCL_TARGET_DEVICES="gpu" \
 ..

# Build and Run
ninja check
```

NOTE Only ESIMD kernels are tested with the code sample, due to the following limitations:

- The emulator is only available on Linux.
- The emulator has limitation on the number of threads used under Linux. As software multi-threading is used for emulating hardware threads, the number of threads being launched for kernel execution is limited by the max number of threads supported by a Linux host machine.
- The emulator supports only ESIMD kernels. Kernels written for SYCL cannot run with the ESIMD_EMULATOR backend. Kernels containing both SYCL and ESIMD code cannot run with the ESIMD_EMULATOR, unlike GPU backends like OpenCL™ or Level Zero.
- The emulator cannot run in parallel with a Host Device.

Restrictions

This section contains lists of the main restrictions that apply when using the ESIMD extension.

NOTE Some extensions are not enforced by the compiler, which may lead to undefined program behavior.

- Features not supported with ESIMD:
  - C and C++ standard libraries support.
  - Device library extensions.
  - A host device.
- Unsupported standard SYCL APIs:
  - Local accessors. Local memory is allocated and accessed via explicit device-side APIs.
  - 2D and 3D accessors.
  - Constant accessors.
  - `sycl::accessor::get_pointer()`. All memory accesses through an accessor are done via explicit APIs. Example: `sycl::ext::intel::esimd::block_store(acc, offset)`
  - Accessors with offsets and/or access range specified.
  - `sycl::sampler` and `sycl::stream` classes.
- Other restrictions:
  - Only Intel GPU devices are supported.
  - Interoperability between regular SYCL and ESIMD kernels is not yet supported. It is not possible to invoke an ESIMD kernel from SYCL kernel and vice versa.

High-Level Optimization

High-level Optimizations (HLO) exploit the properties of source code constructs (for example, loops and arrays) in applications developed in high-level programming languages. While the default optimization level, option `O2` , performs some high-level optimizations, specifying the `O3` option provides the best chance for performing loop transformations to optimize memory accesses.

NOTE
Loop optimizations may result in calls to library routines that can result in additional performance gain on Intel® microprocessors than on non-Intel microprocessors. Additional HLO transformations may be performed for Intel® microprocessors than for non-Intel microprocessors.

Within HLO, loop transformation techniques include:

- Loop Permutation or Interchange
- Loop Distribution
- Loop Fusion
- Loop Unrolling
- Data Prefetching
- Scalar Replacement
- Unroll and Jam
- Loop Blocking or Tiling
- Partial-Sum Optimization
- Predicate Optimization
- Loop Reversal
- Profile-Guided Loop Unrolling
- Loop Peeling
- Data Transformation: Malloc Combining and Memset Combining, Memory Layout Change
- Loop Rerolling
- Memset and Memcpy Recognition
- Statement Sinking for Creating Perfect Loopnests
- Multiversioning: Checks include Dependency of Memory References, and Trip Counts
- Loop Collapsing

Interprocedural Optimization

Interprocedural Optimization (IPO) is an automatic, multi-step process that allows the compiler to analyze your code to determine where you can benefit from specific optimizations.

The compiler may apply the following optimizations:

- Address-taken analysis
- Array dimension padding
- Alias analysis
- Automatic array transposition
- Automatic memory pool formation
- C++ class hierarchy analysis
- Common block variable coalescing
- Common block splitting
- Constant propagation
- Dead call deletion
- Dead formal argument elimination
- Dead function elimination

- Formal parameter alignment analysis
- Forward substitution
- Indirect call conversion
- Inlining
- Mod/ref analysis
- Partial dead call elimination
- Passing arguments in registers to optimize calls and register usage
- Points-to analysis
- Routine key-attribute propagation
- Specialization
- Stack frame alignment
- Structure splitting and field reordering
- Symbol table data promotion
- Un-referenced variable removal
- Whole program analysis

IPO Compilation Models

IPO supports two compilation models - single-file compilation and multi-file compilation.

The compiler performs some single-file interprocedural optimization at the `O2` default optimization level; additionally the compiler may perform some inlining for the `O1` optimization level, such as inlining functions marked with inlining pragmas or attributes (GNU C and C++) and C++ class member functions with bodies included in the class declaration.

Multi-file compilation uses the `[Q]ipo` option, and results in one or more mock object files rather than normal object files. (See the *Compilation* section below for information about mock object files.) Additionally, the compiler collects information from the individual source files that make up the program. Using this information, the compiler performs optimizations across functions and procedures in different source files.

Compiling with IPO

As each source file is compiled with IPO, the compiler stores an intermediate representation (IR) of the source code in a mock object file. The mock object files contain the IR instead of the normal object code. Mock object files can be ten times or more larger than the size of normal object files.

During the IPO compilation phase only the mock object files are visible.

Linking with IPO

When you link with the `[Q]ipo` compiler option the compiler is invoked a final time. The compiler performs IPO across all mock object files. The mock objects must be linked with the compiler or by using the Intel® linking tools. While linking with IPO, the compiler and other linking tools compile mock object files as well as invoke the real/true object files linkers provided on the user's platform.

Whole Program Analysis

The compiler supports a large number of IPO optimizations that can be applied or have its effectiveness greatly increased when the whole program condition is satisfied.

During the analysis process, the compiler reads all Intermediate Representation (IR) in the mock file, object files, and library files to determine if all references are resolved and whether or not a given symbol is defined in a mock object file. Symbols that are included in the IR in a mock object file for both data and functions are candidates for manipulation based on the results of whole program analysis.

There are two types of whole program analysis - object reader method and table method. Most optimizations can be applied if either type of whole program analysis determines that the whole program conditions exists; however, some optimizations require the results of the object reader method, and some optimizations require the results of table method.

Object reader method

In the object reader method, the object reader emulates the behavior of the native linker and attempts to resolve the symbols in the application. If all symbols are resolved, the whole program condition is satisfied. This type of whole program analysis is more likely to detect the whole program condition.

Table method

In the table method the compiler analyzes the mock object files and generates a call-graph.

The compiler contains detailed tables about all of the functions for all important language-specific libraries, like `libc`. In this second method, the compiler constructs a call-graph for the application. The compiler then compares the function table and application call-graph. For each unresolved function in the call-graph, the compiler attempts to resolve the calls by finding an entry for each unresolved function in the compiler tables. If the compiler can resolve the functions call, the whole program condition exists.

See Also

Inline Expansion of Functions

Interprocedural Optimization Options

ipo, Qipo

O

Use Interprocedural Optimization

Use Interprocedural Optimization

This topic discusses how to use IPO from the command line.

Compiling and Linking Using IPO

To enable IPO, you first compile each source file, then link the resulting source files.

Linux

1. Compile your source files with the `ipo` compiler option:

```
icpx -ipo -c a.cpp b.cpp c.cpp
```

   The command produces `a.o`, `b.o`, and `c.o` object files.

   Use the `c` compiler option to stop compilation after generating `.o` object files. The output files contain compiler intermediate representation (IR) corresponding to the compiled source files.
2. Link the resulting files. The following example command will produce an executable named `app`:

```
icpx -ipo -o app a.o b.o c.o
```

   The command invokes the compiler on the objects containing IR and creates a new list of objects to be linked. Alternately, you can use the `xild` tool, with the appropriate linking options.

The separate compile and link commands from the previous steps can be combined into a single command, for example:

```
icpx -ipo -o app a.cpp b.cpp c.cpp
```

The icpx command, shown in the examples, calls `GCC ld` to link the specified object files and produce the executable application, which is specified by the `-o` option.

Windows

1. Compile your source files with the `/Qipo` compiler option:

```
icx /Qipo /c a.cpp b.cpp c.cpp
```

   The command produces `a.obj`, `b.obj`, and `c.obj` object files.

   Use the `c` compiler option to stop compilation after generating `.obj` files. The output files contain compiler intermediate representation (IR) corresponding to the compiled source files.
2. Link the resulting files. The following example command will produce an executable named `app`:

```
icx /Qipo /Feapp a.obj b.obj c.obj
```

   The command invokes the compiler on the objects containing IR and creates a new list of objects to be linked. Alternately, you can use the `xilink` tool, with the appropriate linking options.

The separate compile and link commands from the previous steps can be combined into a single command, for example:

```
icx /Qipo /Feapp a.cpp b.cpp c.cpp
```

The icx command, shown in the examples, calls `link.exe` to link the specified object files and produce the executable application, which is specified by the `/Fe` option.

NOTE
Linux: Using icpx allows the compiler to use standard C++ libraries automatically; icx will not use the standard C++ libraries automatically.

The Intel linking tools emulate the behavior of compiling at `-O0` (Linux) and `/Od` (Windows) option.

If multiple file IPO is applied to a series of object files, no one which are mock object files, no multi-file IPO is performed. The object files are simply linked with the linker.

See Also

c compiler option
o compiler option
Fe compiler option
ipo, Qipo compiler option
O compiler option

Performance and Large Program Considerations

IPO-related Performance Issues

There are some general optimization guidelines for using IPO that you should keep in mind:

- Using IPO on very large programs might trigger internal limits of other compiler optimization phases.
- Applications where the compiler does not have sufficient intermediate representation (IR) coverage to do whole program analysis might not perform as well as those where IR information is complete.

In addition to these general guidelines, there are some practices to avoid while using IPO. The following list summarizes the activities to avoid:

- Do not use the link phase of an IPO compilation using mock object files produced for your application by a different compiler. Intel® compilers cannot inspect mock object files generated by other compilers for optimization opportunities.
- Update make files to call the appropriate Intel linkers when using IPO from scripts. For Linux, replace all instances of `ld` with `xild`; for Windows, replace all instances of `link` with `xilink`.

IPO for Large Programs

In most cases, IPO generates a single true object file for the link-time compilation. This behavior is not optimal for very large programs, perhaps even making it impossible to use `[Q]ipo` compiler option on the application.

The compiler provides two methods to avoid this problem. The first method is an automatic size-based heuristic, which causes the compiler to generate multiple true object files for large link-time compilations. The second method is to manually instruct the compiler to perform multi-object IPO.

- Use the `[Q]ipo`*N* compiler option and pass an integer value in the place of *N*.

Regardless of the method used, it is best to use the compiler defaults first and examine the results. If the defaults do not provide the desired results then experiment with generating a different number of object files.

Using [Q]ipo*N* to Create Multiple Object Files

If you specify `[Q]ipo0`, which is the same as not specifying a value, the compiler uses heuristics to determine whether to create one or more object files based on the expected size of the application. The compiler generates one object file for small applications, and two or more object files for large applications. If you specify any value greater than 0, the compiler generates that number of object files, unless the value you pass a value that exceeds the number of source files. In that case, the compiler creates one object file for each source file then stops generating object files. The generated object files follow OS-specific naming conventions.

The following example commands demonstrate how to use `[Q]ipo2` option to compile large programs.

Linux

```
dpcpp -ipo2 -c a.cpp b.cpp
```

The resulting object files are `ipo_out.o`, `ipo_out1.o`, and `ipo_out2.o`.

Windows

```
dpcpp-cl /Qipo2 /c a.cpp b.cpp
```

The resulting object files are `ipo_out.obj`, `ipo_out1.obj`, and `ipo_out2.obj`.

Link the resulting object files as shown in Use Interprocedural Optimization.

Understanding Code Layout and Multi-Object IPO

One of the optimizations performed during an IPO compilation is code layout. The analysis performed by the compiler during multi-file IPO determines a layout order for all of the routines for which it has intermediate representation (IR) information. For a multi-object IPO compilation, the compiler must tell the linker about the desired order.

The compiler first puts each routine in a named text section that varies depending on the operating system:

Linux

The first routine is placed in `.text00001`, the second is placed in `.text00002`, and so on.

Windows

The first routine is placed in `.text$00001`, the second is placed in `.text$00002`, and so on.

See Also

O compiler option

ipo, Qipo compiler option

Create a Library from IPO Objects

Libraries are often created using a library manager such as `xiar` for Linux or `xilib` for Windows. Given a list of objects, the library manager will insert the objects into a named library to be used in subsequent link steps.

Linux

Use `xiar` to create a library from a list of objects. For example the following command creates a library named `user.a` containing the `a.o` and `b.o` objects:

```
xiar cru user.a a.o b.o
```

Using `xiar` is the same as specifying `xild -lib`.

Windows

Use `xilib` or `xilink -lib` to create libraries of IPO mock object files and link them on the command line.

For example:

1. Assume that you create three mock object files using a command similar to:

```
icx /c /Qipo a.cpp b.cpp c.cpp
```

2. Further assume `a.obj` contains the main subprogram. Create a library with a command similar to:

```
xilib -out:main.lib b.obj c.obj
```

   or

```
xilink -lib -out:main.lib b.obj c.obj
```

3. Link the library and the main program object file with a command similar to:

```
xilink -out:result.exe a.obj main.lib
```

See Also

static compiler option

Inline Expansion of Functions

Inline function expansion does not require that the applications meet the criteria for whole program analysis normally required by IPO; so this optimization is one of the most important optimizations done in Interprocedural Optimization (IPO). For function calls that the compiler believes are frequently executed, the compiler often decides to replace the instructions of the call with code for the function itself.

In the compiler, inline function expansion is performed on relatively small user functions more often than on functions that are relatively large. This optimization improves application performance by performing the following:

- Removing the need to set up parameters for a function call
- Eliminating the function call branch
- Propagating constants

Function inlining can improve execution time by removing the runtime overhead of function calls; however, function inlining can increase code size, code complexity, and compile times. In general, when you instruct the compiler to perform function inlining, the compiler can examine the source code in a much larger context, and the compiler can find more opportunities to apply optimizations.

Specifying the `[Q]ipo` compiler option, multi-file IPO, causes the compiler to perform inline function expansion for calls to procedures defined in other files.

Caution
Using the [Q]ipo (Windows*) options can, in some cases, significantly increase compile time and code size.

The compiler does a certain amount of inlining at the default level.

Selecting Routines for Inlining

The compiler attempts to select the routines whose inline expansions provide the greatest benefit to program performance. The selection is done using default heuristics.

When you use PGO with [Q]ipo, the compiler uses the following guidelines for applying heuristics:

- The default heuristic focuses on the most frequently executed call sites, based on the profile information gathered for the program.
- The default heuristic always inlines very small functions that meet the minimum inline criteria.

Using IPO with PGO

Combining IPO and PGO typically produces better results than using IPO alone. PGO produces dynamic profiling information that can usually provide better optimization opportunities than the static profiling information used in IPO.

The compiler uses characteristics of the source code to estimate which function calls are executed most frequently. It applies these estimates to the PGO-based guidelines described above. The estimation of frequency, based on static characteristics of the source, is not always accurate.

Inline Expansion of Library Functions

By default, the compiler automatically inlines (expands) a number of standard and math library functions at the point of the call to that function, which usually results in faster computation.

Many routines in the libirc, libm, or the svml library are more highly optimized for Intel microprocessors than for non-Intel microprocessors.

The -fno-builtin (Linux*) or the /Qno-builtin-<name> and /Oi- (Windows*) options disable inlining for intrinsic functions and disable the by-name recognition support of intrinsic functions and the resulting optimizations. The /Qno-builtin-<name> option provides the ability to disable inlining for intrinsic functions, fine-tuning the functionality of the /Oi- option, which disables almost all intrinsic functions when used. Use these options if you redefine standard library routines with your own version and your version of the routine has the same name as the standard library routine.

Inlining and Function Preemption (Linux)

You must specify fpic to use function preemption. By default the compiler does not generate the position-independent code needed for preemption.

Compiler Directed Inline Expansion of Functions

Without directions from the user, the compiler attempts to estimate what functions should be inlined to optimize application performance.

The following options are useful in situations where an application can benefit from user function inlining but does not need specific direction about inlining limits.

Option	Effect
fno-builtin (Linux*) or Oi- (Windows)	Disables inlining for intrinsic functions. Disables the by-name recognition support of intrinsic functions and the resulting optimizations. Use this option if

Option	Effect
	you redefine standard library routines with your own version and your version of the routine has the same name as the standard library routine. By default, the compiler automatically inlines (expands) a number of standard and math library functions at the point of the call to that function, which usually results in faster computation. Many routines in the `libirc`, `libm`, or `svml` library are more highly optimized for Intel microprocessors than for non-Intel microprocessors.
setting `inline-debug-info` for the `debug` option	Indicates that the source position information for an inlined function should be retained, rather than replaced, by that of the call which is being inlined.

Developer Directed Inline Expansion of User Functions

In addition to the options that support compiler directed inline expansion of user functions, the compiler also provides compiler options and pragmas that allow you to more precisely direct when and if inline function expansion should occur.

The compiler measures the relative size of a routine in an abstract value of intermediate language units, which is approximately equivalent to the number of instructions that will be generated. The compiler uses the intermediate language unit estimates to classify routines and functions as relatively small, medium, or large functions. The compiler then uses the estimates to determine when to inline a function; if the minimum criteria for inlining is met and all other things are equal, the compiler has an affinity for inlining relatively small functions and not inlining relative large functions.

Typically, the compiler targets functions that have been marked for inlining based on the following:

- **Inlining keywords:** Tells the compiler to inline the specified function. For example, `__inline`, `__forceinline`.
- **Procedure-specific inlining pragmas:** Tells the compiler to inline calls within the targeted procedure if it is legal to do so. For example,`#pragma inline` or `#pragma forceinline` .
- **GCC function attributes for inlining:** Tells the compiler to inline the function even when no optimization level is specified. For example, `__attribute__((always_inline))`.

If your code hits an inlining limit, the compiler issues a warning at the highest warning level. The warning specifies which of the inlining limits have been hit, and the compiler option and/or pragmas needed to get a full report.

Messages in the report refer directly to the command line options or pragmas that can be used to overcome the limits.

See Also

fbuiltin, Oi compiler option
fpic compiler option
ipo, Qipo compiler option
debug (Linux* OS) compiler option
debug (Windows* OS) compiler option
Zi, Z7, ZI compiler option

Methods to Optimize Code Size

This section provides some guidance on how to achieve smaller object and smaller executable size when using the optimizing features of Intel compilers.

There are two compiler options that are designed to prioritize code size over performance:

Option	Result	Notes
`Os`	Favors size over speed	This option enables optimizations that do not increase code size; it produces smaller code size than option `O2`. Option `Os` disables some optimizations that may increase code size for a small speed benefit.
`O1`	Minimizes code size	Compared to option `Os`, option `O1` disables even more optimizations that are generally known to increase code size. Specifying option `O1` implies option `Os`. As an intermediate step in reducing code size, you can replace option `O3` with option `O2` before specifying option `O1`. Option `O1` may improve performance for applications with very large code size, many branches, and execution time not dominated by code within loops.

For more information about compiler options mentioned in this topic, see their full descriptions in the Compiler Reference.

The rest of this topic briefly discusses other methods that may help you further improve code size even when compared to the default behaviors of options `Os` and `O1`.

Things to remember:

- Some of these methods may already be applied by default when options `Os` and `O1` are specified. All the methods mentioned in this topic can be applied at higher optimization levels.
- Some of the options referred to in this topic will not necessarily cause code size reduction, and they may provide varying results (good, bad, or neutral) based on the characteristics of the target code. Still, these are the recommended things to try to see if they cause your binaries to become smaller while maintaining acceptable performance.

Disable or Decrease the Amount of Inlining

Inlining replaces a call to a function with the body of the function. This lets the compiler optimize the code for the inlined function in the context of its caller, usually yielding more specialized and better performing code. This also removes the overhead of calling the function at runtime.

However, replacing a call to a function by the code for that function usually increases code size. The code size increase can be substantial. To eliminate this code size increase, at the cost of the potential performance improvement, inlining can be disabled.

- Advantage: Disabling or reducing this optimization can reduce code size.
- Disadvantage: Performance is likely to be sacrificed by disabling or reducing inlining especially for applications with many small functions.

Use options:

Linux

`fno-inline`

Windows

`Ob0`

Strip Symbols from Your Binaries

You can specify a compiler option to omit debugging and symbol information from the executable without sacrificing its operability.

- Advantage: This method noticeably reduces the size of the binary.
- Disadvantage: It may be very difficult to debug a stripped application.

Use options:

Linux

`Wl, --strip-all`

Windows

None

Dynamically Link Intel-provided Libraries

By default, some of the Intel support and performance libraries are linked statically into an executable. As a result, the library codes are linked into every executable being built. This means that codes are duplicated.

It may be more profitable to link them dynamically.

- Advantage: Performance of the resulting executable is normally not significantly affected. Library codes that are otherwise linked in statically into every executable will not contribute to the code size of each executable with this option. These codes will be shared between all executables using them, and they will be available independent of those executables.
- Disadvantage: The libraries on which the resulting executable depends must be re-distributed with the executable for it to work properly. When libraries are linked statically, only library content that is actually used is linked into the executable. Dynamic libraries contain all the library content. Therefore, it may not be beneficial to use this option if you only need to build and/or distribute a single executable. The executable itself may be much smaller when linked dynamically, compared to a statically linked executable. However, the total size of the executable plus shared libraries or DLLs may be much larger than the size of the statically linked executable.

Use Options:

Linux

`shared-intel`

Windows

`MD`

NOTE Option MD affects all libraries, not only the Intel-provided ones.

Exclude Unused Code and Data from the Executable

Programs often contain dead code or data that is not used during their execution. Even if no expensive whole-program inter-procedural analysis is made at compile time to identify dead code, there are compiler options you can specify to eliminate unused functions and data at link time.

This method is often referred to as function-level or data-level linking.

- Advantage: Only the code that is referenced remains in an executable. Dead functions and data are stripped from the executable. For the options passed to the linker, they also enable the linker to reorder the sections for other possible optimization.
- Disadvantage: The object codes may become slightly larger because each function or datum is put into a separate section. The overhead is eliminated at the linking stage. This method requires linker support to strip unused sections and may increase linking time.

Use Options:

Linux

`-fdata-sections -ffunction-sections -Wl,--gc-sections`

Windows

`/Gw /Gy /link /OPT:REF`

NOTE Option MD affects all libraries, not only the Intel-provided ones.

These options (from the use options example above) are passed to the linker:

Linux

`Wl, --gc-sections`

Windows

`link /OPT:REF`

Disable Recognition and Expansion of Intrinsic Functions

When recognized, intrinsic functions can get expanded inline or their faster implementation in a library may be assumed and linked in. By default, Inline expansion of intrinsic functions is enabled.

In some cases, disabling this behavior may noticeably improve the size of the produced object or binary.

- Advantage: Both the size of the object files and the size of library codes brought into an executable can be reduced.
- Disadvantage: This method can prevent various performance optimizations from happening. Slower standard library implementation will be used. The size of the final executable can be increased in cases when code pulled in statically from a library for an otherwise inlined intrinsic is large.

Use Options:

Linux

`fno-builtin`

Windows

`Oi-`

Additional information:

- This option is already the default if you specify option `O1`.
- For C++, you can specify Linux option `nolib-inline` to disable inline expansion of standard library or intrinsic functions.

- Depending on code characteristics, this option can sometimes increase binary size.

Optimize Exception Handling Data

For SYCL, enabling and disabling of exception handling is supported for host compilation.

If a program requires support for exception handling, the compiler creates a special section containing DWARF directives that are used by the Linux runtime to unwind and catch an exception.

This information is found in the `.eh_frame` section and may be shrunk using the compiler options listed below.

- Advantage:

  These options may shrink the size of the object or binary file by up to 15%, though the amount of the reduction depends on the target platform. These options control whether unwind information is precise at an instruction boundary or at a call boundary. For example, option `fno-asynchronous-unwind-tables` can be used for programs that may *only* throw or catch exceptions.
- Disadvantage: Both options may change the program's behavior. Do not use option `fno-exceptions` for programs that require standard C++ handling for objects of classes with destructors. Do not use option `fno-asynchronous-unwind-tables` for functions compiled with option `-fexceptions` that contain calls to other functions that might throw exceptions or for C++ functions that declare objects with destructors.

Use Options:

Linux

`fno-exceptions` or `fno-asynchronous-unwind-tables`

Windows

None

Read the compiler option descriptions, which explain what the defaults and behavior are for each target platform.

Disable Loop Unrolling

Unrolling a loop increases the size of the loop proportionally to the unroll factor.

Disabling (or limiting) this optimization may help reduce code size at the expense of performance.

- Advantage: Code size is reduced.
- Disadvantage: Performance of otherwise unrolled loops may noticeably degrade because this limits other possible loop optimizations.

Use Options:

Linux

`unroll=0`

Windows

`Qunroll:0`

> **NOTE** This Windows option is not available for SYCL.

Additional information:

This option is already the default if you specify option `Os` or option `O1`.

Disable Automatic Vectorization

The compiler finds possibilities to use SIMD (Intel® Streaming SIMD Extensions (Intel® SSE)/Intel® Advanced Vector Extensions (Intel® AVX)) instructions to improve performance of applications. This optimization is called automatic vectorization.

In most cases, this optimization involves transformation of loops and increases code size, in some cases significantly.

Disabling this optimization may help reduce code size at the expense of performance.

- Advantage: Compile-time is also improved significantly.
- Disadvantage: Performance of otherwise vectorized loops may suffer significantly. If you care about the performance of your application, you should use this option selectively to suppress vectorization on everything except performance-critical parts.

Use Options:

Linux

`no-vec`

Windows

`Qvec-`

Additional information:

Depending on code characteristics, this option can sometimes increase binary size.

Avoid References to Compiler-specific Libraries

While compiler-specific libraries are intended to improve the performance of your application, they increase the size of your binaries.

Certain compiler options may improve the code size.

- Advantage: The compiler will not assume the presence of compiler-specific libraries. It will generate only calls that appear in the source code.
- Disadvantage: This method may sacrifice performance if the library codes were in hotspots. Also, because we cannot assume any libraries, some compiler optimizations will be suppressed.

Use Options:

Linux

`ffreestanding`

Windows

`Qfreestanding-`

NOTE This Windows option is not available for SYCL.

Additional information:

- This option implies option `fno-builtin`. You can override that default by explicitly specifying option `fbuiltin`.
- Depending on code characteristics, this option can sometimes increase binary size.

Use Interprocedural Optimization

Using interprocedural optimization (IPO) may reduce code size. It enables dead code elimination and suppresses generation of code for functions that are always inlined or proven that they are never to be called during execution.

- Advantage: Depending on the code characteristics, this optimization can reduce executable size and improve performance.
- Disadvantage: Binary size can increase depending on code/application..

Use Options:

Linux

`ipo`

Windows

`Qipo`

> **NOTE** This method is not recommended if you plan to ship object files as part of a final product.

Intel® oneAPI DPC++/C++ Compiler Math Library

The Intel® oneAPI DPC++/C++ Compiler includes a mathematical software library containing highly optimized and very accurate mathematical functions. These functions are commonly used in scientific or graphic applications, as well as other programs that rely heavily on floating-point computations. To include support for C99 `_Complex` data types, use the `[Q]std=c99` compiler option.

Many routines in the Intel® oneAPI DPC++/C++ Compiler Math Library are more optimized for Intel® microprocessors than for non-Intel microprocessors.

The `mathimf.h` header file includes prototypes for Intel® oneAPI DPC++/C++ Compiler Math Library functions.

> **NOTE**
> Intel's `math.h` header file is compatible with the GCC Math Library `libm`, but it does not cause the GCC Math Library to be linked. The source can be built with `gcc` or icx. The header file for the math library, `mathimf.h`, contains additional functions that are found only in the math library. The source can only be built using the compiler and libraries.
>
> The long double functions, such as `expl` or `logl`, in the math library are ABI incompatible with the Microsoft libraries. The Intel compiler and libraries support the 80-bit long double data type (see the description of the Qlong-double option). For maximum compatibility, use `math.h` or `mathimf.h` header files along with the math library.

Compiler Math Libraries for Linux

The math library linked to an application depends on the compilation or linkage options specified.

Library	Description
`libimf.a`	Default static math library.

Library	Description
`libimf.so`	Default shared math library.

NOTE The math libraries contain performance-optimized implementations for various Intel platforms. By default, the best implementation for the underlying hardware is selected at runtime. The library dispatch of multi-threaded code may lead to apparent data races, which may be detected by certain software analysis tools. However, as long as the threads are running on cores with the same CPUID, these data races are harmless and not a cause for concern.

Compiler Math Libraries for Windows

The math library linked to an application depends on the compilation or linkage options specified.

Library	Option	Description
`libm.lib`		Default static math library.
`libmmt.lib`	`/MT`	Multi-threaded static math library.
`libmmd.lib`	`/MD`	Dynamically linked math library.
`libmmdd.lib`	`/MDd`	Dynamically linked debug math library.
`libmmds.lib`		Static version compiled with `/MD` option.

oneAPI and OpenCL™ Considerations

Currently, oneAPI uses the OpenCL Specification to determine the ULP accuracy for OpenCL mathematical functions. Details about their precision and accuracy, including tables for single and double precision functions, are available from the Khronos OpenCL Specification's section, Relative Error as ULPs.

Mathematical functions have different accuracy levels on different devices. The OpenCL specification sets a limit on the maximum ULP error (where applicable), but individual devices may provide a more accurate implementation. If the OpenCL implementation is optimized for CPU usage, using the same code may not work on a GPU device.

See Also

Math Function List
Qlong-double compiler option
MD compiler option
MT compiler option
std, Qstd compiler option

Use the Intel® oneAPI DPC++/C++ Compiler Math Library

Many routines in the Intel® oneAPI DPC++/C++ Compiler Math Library are more optimized for Intel® microprocessors than for non-Intel microprocessors.

The `mathimf.h` header file includes prototypes for Intel® oneAPI DPC++/C++ Compiler Math Library functions.

To use the Intel® oneAPI DPC++/C++ Compiler math library, include the header file, `mathimf.h`, in your program. If the compiler is used for linking, then the math library is used by default.

Use Real Functions

The following examples demonstrate how to use the math library with the compiler. After you compile this example and run the program, the program will display the sine value of `x`.

Linux

```
// real_math.c
#include <stdio.h>
#include <mathimf.h>

int main() {
 float fp32bits;
 double fp64bits;
 long double fp80bits;
 long double pi_by_four = 3.141592653589793238/4.0;

// pi/4 radians is about 45 degrees
 fp32bits = (float) pi_by_four; // float approximation to pi/4
 fp64bits = (double) pi_by_four; // double approximation to pi/4
 fp80bits = pi_by_four; // long double (extended) approximation to pi/4

// The sin(pi/4) is known to be 1/sqrt(2) or approximately .7071067
 printf("When x = %8.8f, sinf(x) = %8.8f \n", fp32bits, sinf(fp32bits));
 printf("When x = %16.16f, sin(x) = %16.16f \n", fp64bits, sin(fp64bits));
 printf("When x = %20.20Lf, sinl(x) = %20.20Lf \n", fp80bits, sinl(fp80bits));

 return 0;
}
```

Use the following command to compile the example code on Linux platforms:

```
icx real_math.c
```

Windows

```
// real_math.c
#include <stdio.h>
#include <mathimf.h>

int main() {
  float fp32bits;
  double fp64bits;

// /Qlong-double compiler option required because, without it,
// long double types are mapped to doubles.
  long double fp80bits;
  long double pi_by_four = 3.141592653589793238/4.0;

// pi/4 radians is about 45 degrees
  fp32bits = (float) pi_by_four;

// float approximation to pi/4
  fp64bits = (double) pi_by_four;

// double approximation to pi/4
  fp80bits = pi_by_four;

// long double (extended) approximation to pi/4
// The sin(pi/4) is known to be 1/sqrt(2) or approximately .7071067
  printf("When x = %8.8f, sinf(x) = %8.8f \n",
```

```
  fp32bits, sinf(fp32bits));

  printf("When x = %16.16f, sin(x) = %16.16f \n",
  fp64bits, sin(fp64bits));

  printf("When x = %20.20f, sinl(x) = %20.20f \n",
  (double) fp80bits, (double) sinl(fp80bits));

// printf() does not support the printing of long doubles
// on Microsoft Windows, so fp80bits is cast to double in this example.
  return 0;
}
```

Since the `real_math.c` program includes the `long double` data type, use the `/Qlong-double` and `/Qpc80` compiler options in the command line:

Use the following command to compile the example code on Windows platforms:

```
icx /Qlong-double /Qpc80 real_math.c
```

Use Complex Functions

After you compile this example and run the program, you should get the following results:

`When z = 1.0000000 + 0.7853982 i, cexpf(z) = 1.9221154 + 1.9221156 i`

`When z = 1.000000000000 + 0.785398163397 i, cexp(z) = 1.922115514080 + 1.922115514080 i`

Linux and Windows

```
// complex_math.c
#include <stdio.h>
#include <complex.h>

int main() {
  float _Complex c32in,c32out;
  double _Complex c64in,c64out;
  double pi_by_four= 3.141592653589793238/4.0;
  c64in = 1.0 + I pi_by_four;

// Create the double precision complex number 1 + (pi/4)  i
// where I is the imaginary unit.
  c32in = (float _Complex) c64in;

// Create the float complex value from the double complex value.
  c64out = cexp(c64in);
  c32out = cexpf(c32in);

// Call the complex exponential,
// cexp(z) = cexp(x+iy) = e^ (x + i y) = e^x  (cos(y) + i sin(y))
 printf("When z = %7.7f + %7.7f i, cexpf(z) = %7.7f + %7.7f i \n"
 ,crealf(c32in),cimagf(c32in),crealf(c32out),cimagf(c32out));
 printf("When z = %12.12f + %12.12f i, cexp(z) = %12.12f + %12.12f i \n"
 ,creal(c64in),cimag(c64in),creal(c64out),cimagf(c64out));

  return 0;
}
```

Since this example program includes the `_Complex` data type, be sure to include the `[Q]std=c99` compiler option in the command line. For example:

Linux

```
icx -std=c99 complex_math.c
```

Windows

```
icx Qstd=c99 complex_math.c
```

> **NOTE** `_Complex` data types are supported in C but not in C++ programs.

Exception Conditions

If you call a math function using argument(s) that may produce undefined results, an error number is assigned to the system variable `errno`. Math function errors are usually domain errors or range errors.

Domain errors result from arguments that are outside the domain of the function. For example, `acos` is defined only for arguments between -1 and +1 inclusive. Attempting to evaluate `acos(-2)` or `acos(3)` results in a domain error, where the return value is `QNaN`.

Range errors occur when a mathematically valid argument results in a function value that exceeds the range of representable values for the floating-point data type. Attempting to evaluate `exp(1000)` results in a range error, where the return value is `INF`.

When domain or range error occurs, the following values are assigned to `errno`:

- domain error (`EDOM`): `errno = 33`
- range error (`ERANGE`): `errno = 34`

The following example shows how to read the `errno` value for an `EDOM` and `ERANGE` error.

```
// errno.c
#include <errno.h>
#include <mathimf.h>
#include <stdio.h>

int main(void) {
  double neg_one=-1.0;
  double zero=0.0;

// The natural log of a negative number is considered a domain error - EDOM
  printf("log(%e) = %e and errno(EDOM) = %d \n",neg_one,log(neg_one),errno);

// The natural log of zero is considered a range error - ERANGE
  printf("log(%e) = %e and errno(ERANGE) = %d \n",zero,log(zero),errno);
}
```

The output of `errno.c` will look like this:

`log(-1.000000e+00) = nan and errno(EDOM) = 33`

`log(0.000000e+00) = -inf and errno(ERANGE) = 34`

For the math functions in this section, a corresponding value for `errno` is listed when applicable.

Other Considerations

Some math functions are inlined automatically by the compiler. The functions actually inlined may vary and may depend on any vectorization or processor-specific compilation options used. You can disable automatic inline expansion of all functions by compiling your program with the `-fno-builtin` option (Linux) or the `/Oi-` option (Windows).

It is strongly recommended to use the default rounding mode (round-to-nearest-even) when calling math library transcendental functions and compiling with default optimization or higher. Faster implementations—in terms of latency and/or throughput— of these functions are validated under the default round-to-nearest-even mode. Using other rounding modes may make results generated by these faster implementations less accurate, or set unexpected floating-point status flags. This behavior may be avoided by using the `-fp-model strict` option (Linux) or `/fp: strict` option (Windows). This option warns the compiler not to assume default settings for the floating-point environment.

NOTE 64-bit decimal transcendental functions rely on binary double extended precision arithmetic. To obtain accurate results, user applications that call 64-bit decimal transcendentals should ensure that the x87 unit is operating in 80-bit precision (64-bit binary significands). In an environment where the default x87 precision is not 80 bits, such as Windows, it can be set to 80 bits by compiling the application source files with the `/Qpc80` option.

A change of the default precision control or rounding mode may affect the results returned by some of the mathematical functions.

The following are important compiler options when using certain data types in IA-32 and Intel® 64 architectures running Windows operating systems:

- `/Qlong-double`: Use this option when compiling programs that require support for the `long double` data type (80-bit floating-point). Without this option, compilation will be successful, but `long double` data types will be mapped to `double` data types.
- `/Qstd=c99`: Use this option when compiling programs that require support for `_Complex` data types.

See Also

fbuiltin, Oi compiler option
Overview: Tuning Performance
Qlong-double compiler option
std, Qstd compiler option

Math Function List

Many routines in the Intel® oneAPI DPC++/C++ Compiler Math Library are more optimized for Intel® microprocessors than for non-Intel microprocessors.

The `mathimf.h` header file includes prototypes for Intel® oneAPI DPC++/C++ Compiler Math Library functions.

The math functions are listed here by function type.

NOTE
FP16 Math Functions have the following requirements:

- Version 2021.4 or higher of the Intel® oneAPI DPC++/C++ Compiler.
- A next-generation Intel® Xeon® Scalable processor, code name Sapphire Rapids.

Function Type	Name
Trigonometric Functions	`acos`
	`acosd`
	`acospi`

Function Type	Name
	asin
	asind
	asinpi
	atan
	atan2
	atan2pi
	atand
	atan2d
	atand2
	atanpi
	cos
	cosd
	cospi
	cot
	cotd
	sin
	sincos
	sincosd
	sind
	sinpi
	tan
	tand
	tanpi
Hyperbolic Functions	acosh
	asinh
	atanh
	cosh
	sinh
	sinhcosh
	tanh
Exponential Functions	cbrt

Function Type	Name
	exp
	exp10
	exp2
	expm1
	frexp
	hypot
	invsqrt
	ilogb
	ldexp
	log
	log10
	log1p
	log2
	logb
	pow
	pow2o3
	pow3o2
	powr
	scalb
	scalbln
	scalbn
	sqrt
Special Functions	annuity
	cdfnorm
	cdfnorminv
	compound
	erf
	erfcx
	erfc
	erfcinv
	erfinv

Function Type	Name
	gamma
	gamma_r
	j0
	j1
	jn
	lgamma
	lgamma_r
	tgamma
	y0
	y1
	yn
Nearest Integer Functions	ceil
	floor
	llrint
	llround
	lrint
	lround
	modf
	nearbyint
	rint
	round
	trunc
Remainder Functions	fmod
	remainder
	remquo
Miscellaneous Functions	copysign
	fabs
	fdim
	finite
	fma
	fmax

Function Type	Name
	fmin
	fpclassify
	isfinite
	isgreater
	isgreaterequal
	isinf
	isless
	islessequal
	islessgreater
	isnan
	isnormal
	isunordered
	maxmag
	minmag
	nan
	nextafter
	nexttoward
	signbit
	significand
Complex Functions	cabs
	cacos
	cacosh
	carg
	casin
	casinh
	catan
	catanh
	ccos
	cexp
	cexp2
	cimag

Function Type	Name
	cis
	clog
	clog10
	conj
	ccosh
	cpow
	cproj
	creal
	csin
	csinh
	csqrt
	ctan
	ctanh

Trigonometric Functions

Many routines in the Intel® oneAPI DPC++/C++ Compiler Math Library are more optimized for Intel® microprocessors than for non-Intel microprocessors.

The `mathimf.h` header file includes prototypes for Intel® oneAPI DPC++/C++ Compiler Math Library functions.

The math library supports the following trigonometric functions:

> **NOTE**
> FP16 Math Functions have the following requirements:
>
> - Version 2021.4 or higher of the Intel® oneAPI DPC++/C++ Compiler.
> - A next-generation Intel® Xeon® Scalable processor, code name Sapphire Rapids.

acos

Description: The `acos` function returns the principal value of the inverse cosine of `x` in the range [0, pi] radians for `x` in the interval [-1,1].

errno: EDOM, for |x| > 1

Calling interface:

```
double acos(double x);
long double acosl(long double x);
float acosf(float x);
_Float16 acosf16(_Float16 x);
```

acosd

Description: The `acosd` function returns the principal value of the inverse cosine of `x` in the range [0,180] degrees for `x` in the interval [-1,1].

errno: EDOM, for |x| > 1

Calling interface:

```
double acosd(double x);
long double acosdl(long double x);
float acosdf(float x);
_Float16 acosdf16(_Float16 x);
```

acospi

Description: The `acospi` function returns the principal value of the inverse cosine of `x`, divided by pi, in the range [0,1] for `x` in the interval [-1,1].

errno: EDOM, for |x| > 1

Calling interface:

```
double acospi(double x);
float acospif(float x);
_Float16 acospif16(_Float16 x);
```

asin

Description: The `asin` function returns the principal value of the inverse sine of `x` in the range [-pi/2, +pi/2] radians for `x` in the interval [-1,1].

errno: EDOM, for |x| > 1

Calling interface:

```
double asin(double x);
long double asinl(long double x);
float asinf(float x);
_Float16 asinf16(_Float16 x);
```

asind

Description: The `asind` function returns the principal value of the inverse sine of `x` in the range [-90,90] degrees for `x` in the interval [-1,1].

errno: EDOM, for |x| > 1

Calling interface:

```
double asind(double x);
long double asindl(long double x);
float asindf(float x);
_Float16 asindf16(_Float16 x);
```

asinpi

Description: The `asinpi` function returns the principal value of the inverse sine of `x`, divided by pi, in the range [-1/2,1/2] degrees for `x` in the interval [-1,1].

errno: EDOM, for |x| > 1 divided by pi

Calling interface:

```
double asinpi(double x);
```

```
float asinpif(float x);
_Float16 asinpif16(_Float16 x);
```

atan

Description: The `atan` function returns the principal value of the inverse tangent of `x` in the range [-pi/2, +pi/2] radians.

Calling interface:

```
double atan(double x);
long double atanl(long double x);
float atanf(float x);
_Float16 atanf16(_Float16 x);
```

atan2

Description: The `atan2` function returns the principal value of the inverse tangent of `y/x` in the range [-pi, +pi] radians.

errno: EDOM, for x = 0 and y = 0

Calling interface:

```
double atan2(double y, double x);
long double atan2l(long double y, long double x);
float atan2f(float y, float x);
_Float16 atan2f16(_Float16 y, _Float16 x);
```

atan2pi

Description: The `atan2pi` function returns the principal value of the inverse tangent of `y/x`, divided by pi, in the range [-1, +1].

errno: EDOM, for x = 0 and y = 0

Calling interface:

```
double atan2pi(double y, double x);
float atan2pif(float y, float x);
_Float16 atan2pif16(_Float16 y, _Float16 x);
```

atand

Description: The `atand` function returns the principal value of the inverse tangent of `x` in the range [-90,90] degrees.

Calling interface:

```
double atand(double x);
long double atandl(long double x);
float atandf(float x);
_Float16 atandf16(_Float16 x);
```

atan2d

Description: The `atan2d` function returns the principal value of the inverse tangent of `y/x` in the range [-180, +180] degrees.

errno: EDOM, for x = 0 and y = 0.

Calling interface:

```
double atan2d(double x, double y);
```

```
long double atan2dl(long double x, long double y);
float atan2df(float x, float y);
_Float16 atan2df16(_Float16 x, _Float16 y);
```

atand2

Description: The `atand2` function returns the principal value of the inverse tangent of `y/x` in the range [-180, +180] degrees.

errno: EDOM, for x = 0 and y = 0.

Calling interface:

```
double atand2(double x, double y);
long double atand2l(long double x, long double y);
float atand2f(float x, float y);
_Float16 atand2f16(_Float16 x, _Float16 y);
```

atanpi

Description: The `atanpi` function returns the principal value of the inverse tangent of `x`, divided by pi, in the range [-1/2, +1/2].

Calling interface:

```
double atanpi(double x);
float atanpif(float x);
_Float16 atanpif16(_Float16 x);
```

cos

Description: The `cos` function returns the cosine of `x` measured in radians.

Calling interface:

```
double cos(double x);
long double cosl(long double x);
float cosf(float x);
_Float16 float cosf16(_Float16 x);
```

cosd

Description: The `cosd` function returns the cosine of `x` measured in degrees.

Calling interface:

```
double cosd(double x);
long double cosdl(long double x);
float cosdf(float x);
_Float16 cosdf16(_Float16 x);
```

cospi

Description: The `cospi` function returns the cosine of `x` multiplied by pi, `cos(x*pi)`.

Calling interface:

```
double cospi(double x);
float cospif(float x);
_Float16 cospif16(_Float16);
```

cot

Description: The `cot` function returns the cotangent of `x` measured in radians.

errno: ERANGE, for overflow conditions at x = 0.

Calling interface:

```
double cot(double x);
long double cotl(long double x);
float cotf(float x);
_Float16 cotf16(_Float16 x);
```

cotd

Description: The `cotd` function returns the cotangent of `x` measured in degrees.

errno: ERANGE, for overflow conditions at x = 0.

Calling interface:

```
double cotd(double x);
long double cotdl(long double x);
float cotdf(float x);
_Float16 cotdf16(_Float16 x);
```

sin

Description: The `sin` function returns the sine of `x` measured in radians.

Calling interface:

```
double sin(double x);
long double sinl(long double x);
float sinf(float x);
_Float16 sinf16(_Float16 x);
```

sincos

Description: The `sincos` function returns both the sine and cosine of `x` measured in radians.

Calling interface:

```
void sincos(double x, double *sinval, double *cosval);
void sincosl(long double x, long double *sinval, long double *cosval);
void sincosf(float x, float *sinval, float *cosval);
void sincosf16(_Float16 x, _Float16 *sinval, _Float16 *cosval);
```

sincosd

Description: The `sincosd` function returns both the sine and cosine of `x` measured in degrees.

Calling interface:

```
void sincosd(double x, double *sinval, double *cosval);
void sincosdl(long double x, long double *sinval, long double *cosval);
void sincosdf(float x, float *sinval, float *cosval);
void sincosdf16(_Float16 x, _Float16 *sinval, _Float16 *cosval);
```

sind

Description: The `sind` function computes the sine of `x` measured in degrees.

Calling interface:

```
double sind(double x);
```

```
long double sindl(long double x);
float sindf(float x);
_Float16 sindf16((_Float16 x);
```

sinpi

Description: The `sinpi` function returns the sine of `x` multiplied by pi, `sin(x*pi)`.

Calling interface:

```
double sinpi(double x);
float sinpif(float x);
_Float16 sinpif16(_Float16 x);
```

tan

Description: The `tan` function returns the tangent of `x` measured in radians.

Calling interface:

```
double tan(double x);
long double tanl(long double x);
float tanf(float x);
_Float16 tanf16(_Float16 x);
```

tand

Description: The `tand` function returns the tangent of `x` measured in degrees.

errno: ERANGE, for overflow conditions

Calling interface:

```
double tand(double x);
long double tandl(long double x);
float tandf(float x);
_Float16 tandf16(_Float16 x);
```

tanpi

Description: The `tanpi` function returns the tangent of `x` multiplied by pi, `tan(x*pi)`.

Calling interface:

```
double tanpi(double x);
float tanpif(float x);
_Float16 tanpif16(_Float16 x);
```

Hyperbolic Functions

Many routines in the Intel® oneAPI DPC++/C++ Compiler Math Library are more optimized for Intel® microprocessors than for non-Intel microprocessors.

The `mathimf.h` header file includes prototypes for Intel® oneAPI DPC++/C++ Compiler Math Library functions.

The math library supports the following `hyperbolic` functions:

NOTE
FP16 Math Functions have the following requirements:

- Version 2021.4 or higher of the Intel® oneAPI DPC++/C++ Compiler.
- A next-generation Intel® Xeon® Scalable processor, code name Sapphire Rapids.

acosh

Description: The `acosh` function returns the inverse hyperbolic cosine of `x`.

errno: EDOM, for x < 1

Calling interface:
```
double acosh(double x);
long double acoshl(long double x);
float acoshf(float x);
_Float16 acoshf16(_Float16 x);
```

asinh

Description: The `asinh` function returns the inverse hyperbolic sine of `x`.

Calling interface:
```
double asinh(double x);
long double asinhl(long double x);
float asinhf(float x);
_Float16 asinhf16(_Float16 x);
```

atanh

Description: The `atanh` function returns the inverse hyperbolic tangent of `x`.

errno:

EDOM, for |x| > 1

ERANGE, for x = 1

Calling interface:
```
double atanh(double x);
long double atanhl(long double x);
float atanhf(float x);
_Float16 atanhf16(_Float16 x);
```

cosh

Description: The `cosh` function returns the hyperbolic cosine of `x`, $(e^x + e^{-x})/2$.

errno: ERANGE, for overflow conditions

Calling interface:
```
double cosh(double x);
long double coshl(long double x);
float coshf(float x);
_Float16 coshf16(_Float16 x);
```

sinh

Description: The `sinh` function returns the hyperbolic sine of `x`, $(e^x - e^{-x})/2$.

errno: ERANGE, for overflow conditions

Calling interface:

```
double sinh(double x);
long double sinhl(long double x);
float sinhf(float x);
_Float16 sinhf16(_Float16 x);
```

sinhcosh

Description: The `sinhcosh` function returns both the hyperbolic sine and hyperbolic cosine of `x`.

errno: ERANGE, for overflow conditions

Calling interface:

```
void sinhcosh(double x, double *sinval, double *cosval);
void sinhcoshl(long double x, long double *sinval, long double *cosval);
void sinhcoshf(float x, float *sinval, float *cosval);
void sinhcoshf16(_Float16 x, _Float16 *sinval, _Float16 *cosval);
```

tanh

Description: The `tanh` function returns the hyperbolic tangent of `x`, $(e^{x} - e^{-x}) / (e^{x} + e^{-x})$.

Calling interface:

```
double tanh(double x);
long double tanhl(long double x);
float tanhf(float x);
_Float16 tanhf16(_Float16 x);
```

Exponential Functions

Many routines in the Intel® oneAPI DPC++/C++ Compiler Math Library are more optimized for Intel® microprocessors than for non-Intel microprocessors.

The `mathimf.h` header file includes prototypes for Intel® oneAPI DPC++/C++ Compiler Math Library functions.

The math library supports the following exponential functions:

NOTE
FP16 Math Functions have the following requirements:

- Version 2021.4 or higher of the Intel® oneAPI DPC++/C++ Compiler.
- A next-generation Intel® Xeon® Scalable processor, code name Sapphire Rapids.

cbrt

Description: The `cbrt` function returns the cube root of `x`.

Calling interface:

```
double cbrt(double x);
long double cbrtl(long double x);
float cbrtf(float x);
_Float16 cbrtf16(_Float16 x);
```

exp

Description: The `exp` function returns `e` raised to the `x` power, e^x.

errno: ERANGE, for underflow and overflow conditions

Calling interface:

```
double exp(double x);
long double expl(long double x);
float expf(float x);
_Float16 expf16(_Float16 x);
```

exp10

Description: The `exp10` function returns 10 raised to the `x` power, 10^x.

errno: ERANGE, for underflow and overflow conditions

Calling interface:

```
double exp10(double x);
long double exp10l(long double x);
float exp10f(float x);
_Float16 exp10f16(_Float16 x);
```

exp2

Description: The `exp2` function returns 2 raised to the `x` power, 2^x.

errno: ERANGE, for underflow and overflow conditions

Calling interface:

```
double exp2(double x);
long double exp2l(long double x);
float exp2f(float x);
_Float16 exp2f16(_Float16 x);
```

expm1

Description: The `expm1` function returns `e` raised to the `x` power, minus 1, $e^x - 1$.

errno: ERANGE, for overflow conditions

Calling interface:

```
double expm1(double x);
long double expm1l(long double x);
float expm1f(float x);
_Float16 expm1f16(_Float16 x);
```

frexp

Description: The `frexp` function converts a floating-point number `x` into signed normalized fraction in [1/2, 1) multiplied by an integral power of two. The signed normalized fraction is returned, and the integer exponent stored at location `exp`.

Calling interface:

```
double frexp(double x, int *exp);
long double frexpl(long double x, int *exp);
float frexpf(float x, int *exp);
_Float16 frexpf16(_Float16 x, int *exp);
```

hypot

Description: The `hypot` function returns the square root of $(x^2 + y^2)$.

errno: ERANGE, for overflow conditions

Calling interface:

```
double hypot(double x, double y);
long double hypotl(long double x, long double y);
float hypotf(float x, float y);
_Float16 hypotf16(_Float16 x, _Float16 y);
```

ilogb

Description: The `ilogb` function returns the exponent of `x` base two as a `signed int` value.

errno: ERANGE, for x = 0

Calling interface:

```
int ilogb(double x);
int ilogbl(long double x);
int ilogbf(float x);
int ilogbf16(_Float16 x);
```

invsqrt

Description: The `invsqrt` function returns the inverse square root.

Calling interface:

```
double invsqrt(double x);
long double invsqrtl(long double x);
float invsqrtf(float x);
_Float16 invsqrtf16(_Float16 x);
```

ldexp

Description: The `ldexp` function returns $x*2^{exp}$, where `exp` is an integer value.

errno: ERANGE, for underflow and overflow conditions

Calling interface:

```
double ldexp(double x, int exp);
long double ldexpl(long double x, int exp);
float ldexpf(float x, int exp);
_Float16 ldexpf16(_Float16 x, int exp);
```

log

Description: The `log` function returns the natural log of `x`, ln(`x`).

errno: EDOM, for x < 0
errno: ERANGE, for x = 0

Calling interface:

```
double log(double x);
long double logl(long double x);
float logf(float x);
_Float16 logf16(_Float16 x);
```

log10

Description: The `log10` function returns the base-10 log of `x`, $\log_{10}(x)$.

errno: EDOM, for x < 0
errno: ERANGE, for x = 0

Calling interface:

```
double log10(double x);
long double log10l(long double x);
float log10f(float x);
_Float16 log10f16(_Float16 x);
```

log1p

Description: The `log1p` function returns the natural log of `(x+1), ln(x + 1)`.

errno: EDOM, for x < -1
errno: ERANGE, for x = -1

Calling interface:

```
double log1p(double x);
long double log1pl(long double x);
float log1pf(float x);
_Float16 log1pf16(_Float16 x);
```

log2

Description: The `log2` function returns the base-2 log of `x`, $\log_2(x)$.

errno: EDOM, for x < 0
errno: ERANGE, for x = 0

Calling interface:

```
double log2(double x);
long double log2l(long double x);
float log2f(float x);
_Float16 log2f16(_Float16 x);
```

logb

Description: The `logb` function returns the signed exponent of `x`.

errno: EDOM, for x = 0

Calling interface:

```
double logb(double x);
long double logbl(long double x);
float logbf(float x);
_Float16 logbf16(_Float16 x);
```

pow

Description: The `pow` function returns `x` raised to the power of `y`, x^y.

errno: EDOM, for x = 0 and y < 0
errno: EDOM, for x < 0 and y is a non-integer
errno: ERANGE, for overflow and underflow conditions

Calling interface:

```
double pow(double x, double y);
```

```
long double powl(double x, double y);
float powf(float x, float y);
_Float16 powf16(_Float16 x, _Float16 y);
```

pow2o3

Description: The `pow2o3` function returns the cube root of `x` squared, `cbrt(x`^{2}`)`.

Calling interface:

```
double pow2o3(double x);
float pow2o3f(float x);
_Float16 pow2o3f16(_Float16 x);
```

pow3o2

Description: The `pow3o2` function returns the square root of the cube of `x`, `sqrt(x`^{3}`)`.

errno: EDOM, for x < 0
errno: ERANGE, for overflow and underflow conditions

Calling interface:

```
double pow3o2(double x);
float pow3o2f(float x);
_Float16 pow3o2f16(_Float16 x);
```

powr

Description: The `powr` function returns `x` raised to the power of `y`, x^y, where x ≥ 0.

errno: EDOM, for x < 0
errno: ERANGE, for overflow and underflow conditions

Calling interface:

```
double powr(double x, double y);
float powrf(float x, float y);
_Float16 powrf16(_Float16 x, _Float16 y);
```

scalb

Description: The `scalb` function returns $x*2^y$, where `y` is a floating-point value.

errno: ERANGE, for underflow and overflow conditions

Calling interface:

```
double scalb(double x, double y);
long double scalbl(long double x, long double y);
float scalbf(float x, float y);
_Float16 scalbf16(_Float16 x, _Float16 y);
```

scalbn

Description: The `scalbn` function returns $x*2^n$, where `n` is an integer value.

errno: ERANGE, for underflow and overflow conditions

Calling interface:

```
double scalbn(double x, int n);
long double scalbnl (long double x, int n);
float scalbnf(float x, int n);
_Float16 scalbnf16(_Float16 x, int n);
```

scalbln

Description: The `scalbln` function returns $x*2^n$, where `n` is a long integer value.

errno: ERANGE, for underflow and overflow conditions

Calling interface:
```
double scalbln(double x, long int n);
long double scalblnl (long double x, long int n);
float scalblnf(float x, long int n);
_Float16 scalblnf16(_Float16 x, long int n);
```

sqrt

Description: The `sqrt` function returns the correctly rounded square root.

errno: EDOM, for x < 0

Calling interface:
```
double sqrt(double x);
long double sqrtl(long double x);
float sqrtf(float x);
_Float16 sqrtf16(_Float16 x);
```

Special Functions

Many routines in the Intel® oneAPI DPC++/C++ Compiler Math Library are more optimized for Intel® microprocessors than for non-Intel microprocessors.

The `mathimf.h` header file includes prototypes for Intel® oneAPI DPC++/C++ Compiler Math Library functions.

The math library supports the following special functions:

NOTE
FP16 Math Functions have the following requirements:

- Version 2021.4 or higher of the Intel® oneAPI DPC++/C++ Compiler.
- A next-generation Intel® Xeon® Scalable processor, code name Sapphire Rapids.

annuity

Description: The `annuity` function computes the present value factor for an annuity, $(1 - (1+x)^{(-y)}) / x$, where `x` is a rate and `y` is a period.

errno: ERANGE, for underflow and overflow conditions

Calling interface:
```
double annuity(double x, double y);
long double annuityl(long double x, long double y);
float annuityf(float x, float y);
_Float16 annuityf16(_Float16 x, _Float16 y);
```

cdfnorm

Description: The `cdfnorm` function returns the cumulative normal distribution function value.

Calling interface:
```
double cdfnorm(double x);
```

```
float cdfnormf(float x);
_Float16 cdfnormf16 (_Float16 x);
```

cdfnorminv

Description: The `cdfnorminv` function returns the inverse cumulative normal distribution function value.

errno:
EDOM, for finite or infinite `(x > 1) || (x < 0)`
ERANGE, for `x = 0 or x = 1`

Calling interface:

```
double cdfnorminv(double x);
float cdfnorminvf (float x);
_Float16 cdfnorminvf16 (_Float16 x);
```

compound

Description: The `compound` function computes the compound interest factor, $(1+x)^y$, where `x` is a rate and `y` is a period.

errno: ERANGE, for underflow and overflow conditions

Calling interface:

```
double compound(double x, double y);
long double compoundl(long double x, long double y);
float compoundf(float x, float y);
_Float16 compoundf16(_Float16 x, _Float16 y);
```

erf

Description: The `erf` function returns the error function value.

Calling interface:

```
double erf(double x);
long double erfl(long double x);
float erff(float x);
_Float16 erff16(_Float16 x);
```

erfc

Description: The `erfc` function returns the complementary error function value.

errno: ERANGE, for underflow conditions

Calling interface:

```
double erfc(double x);
long double erfcl(long double x);
float erfcf(float x);
_Float16 erfcf16(_Float16 x);
```

erfcx

Description: The `erfcx` function returns the scaled complementary error function value.

errno: ERANGE, for overflow conditions

Calling interface:

```
double erfcx(double x);
float erfcxf(float x);
```

erfcinv

Description: The `erfcinv` function returns the value of the inverse complementary error function of `x`.

errno: EDOM, for finite or infinite (x > 2) || (x < 0)

Calling interface:
```
double erfcinv(double x);
float erfcinvf(float x);
_Float16 erfcinvf16(_Float16 x);
```

erfinv

Description: The `erfinv` function returns the value of the inverse error function of `x`.

errno: EDOM, for finite or infinite |x| > 1

Calling interface:
```
double erfinv(double x);
long double erfinvl(long double x);
float erfinvf(float x);
_Float16 erfinvf16(_Float16 x);
```

gamma

Description: The `gamma` function returns the value of the logarithm of the absolute value of gamma.

errno: ERANGE, for overflow conditions when x is a negative integer.

Calling interface:
```
double gamma(double x);
long double gammal(long double x);
float gammaf(float x);
_Float16 gammaf16(_Float16 x);
```

gamma_r

Description: The `gamma_r` function returns the value of the logarithm of the absolute value of gamma. The sign of the gamma function is returned in the integer `signgam`.

Calling interface:
```
double gamma_r(double x, int *signgam);
long double gammal_r(long double x, int *signgam);
float gammaf_r(float x, int *signgam);
_Float16 gammaf16_r(_Float16 x, int *signgam);
```

j0

Description: Computes the Bessel function (of the first kind) of `x` with order 0.

Calling interface:
```
double j0(double x);
long double j0l(long double x);
float j0f(float x);
_Float16 j0f16(_Float16 x);
```

j1

Description: Computes the Bessel function (of the first kind) of `x` with order 1.

Calling interface:

```
double j1(double x);
long double j1l(long double x);
float j1f(float x);
_Float16 j1f16(_Float16 x);
```

jn

Description: Computes the Bessel function (of the first kind) of `x` with order `n`.

Calling interface:

```
double jn(int n, double x);
long double jnl(int n, long double x);
float jnf(int n, float x);
_Float16 jnf16(int n, _Float16 x);
```

lgamma

Description: The `lgamma` function returns the value of the logarithm of the absolute value of gamma.

errno: ERANGE, for overflow conditions, x=0 or negative integers.

Calling interface:

```
double lgamma(double x);
long double lgammal(long double x);
float lgammaf(float x);
_Float16 lgammaf16(_Float16 x);
```

lgamma_r

Description: The `lgamma_r` function returns the value of the logarithm of the absolute value of gamma. The sign of the gamma function is returned in the integer `signgam`.

errno: ERANGE, for overflow conditions, x=0 or negative integers.

Calling interface:

```
double lgamma_r(double x, int *signgam);
long double lgammal_r(long double x, int *signgam);
float lgammaf_r(float x, int *signgam);
_Float16 lgammaf16_r(_Float16 x, int *signgam);
```

tgamma

Description: The `tgamma` function computes the gamma function of `x`.

errno:

EDOM, for x=0 or negative integers.

ERANGE, for overflow conditions.

Calling interface:

```
double tgamma(double x);
long double tgammal(long double x);
float tgammaf(float x);
_Float16 tgammaf16(_Float16 x);
```

y0

Description: Computes the Bessel function (of the second kind) of `x` with order 0.

errno: EDOM, for x <= 0

Calling interface:

```
double y0(double x);
long double y0l(long double x);
float y0f(float x);
_Float16 y0f16(_Float16 x);
```

y1

Description: Computes the Bessel function (of the second kind) of `x` with order 1.

errno: EDOM, for x <= 0

Calling interface:

```
double y1(double x);
long double y1l(long double x);
float y1f(float x);
_Float16 y1f16(_Float16 x);
```

yn

Description: Computes the Bessel function (of the second kind) of `x` with order `n`.

errno: EDOM, for x <= 0

Calling interface:

```
double yn(int n, double x);
long double ynl(int n, long double x);
float ynf(int n, float x);
_Float16 ynf16(int n, _Float16 x);
```

Nearest Integer Functions

Many routines in the Intel® oneAPI DPC++/C++ Compiler Math Library are more optimized for Intel® microprocessors than for non-Intel microprocessors.

The `mathimf.h` header file includes prototypes for Intel® oneAPI DPC++/C++ Compiler Math Library functions.

The math library supports the following nearest integer functions:

NOTE
FP16 Math Functions have the following requirements:

- Version 2021.4 or higher of the Intel® oneAPI DPC++/C++ Compiler.
- A next-generation Intel® Xeon® Scalable processor, code name Sapphire Rapids.

ceil

Description: The `ceil` function returns the smallest integral value not less than `x` as a floating-point number.

Calling interface:

```
double ceil(double x);
long double ceill(long double x);
float ceilf(float x);
_Float16 ceilf16(_Float16 x);
```

floor

Description: The `floor` function returns the largest integral value not greater than `x` as a floating-point value.

Calling interface:

```
double floor(double x);
long double floorl(long double x);
float floorf(float x);
_Float16 floorf16(_Float16 x);
```

llrint

Description: The `llrint` function returns the rounded integer value (according to the current rounding direction) as a `long long int`.

errno: ERANGE, for values too large

Calling interface:

```
long long int llrint(double x);
long long int llrintl(long double x);
long long int llrintf(float x);
long long int llrintf16(_Float16 x);
```

llround

Description: The `llround` function returns the rounded integer value as a `long long int`.

errno: ERANGE, for values too large

Calling interface:

```
long long int llround(double x);
long long int llroundl(long double x);
long long int llroundf(float x);
long long int llroundf16(_Float16 x);
```

lrint

Description: The `lrint` function returns the rounded integer value (according to the current rounding direction) as a `long int`.

errno: ERANGE, for values too large

Calling interface:

```
long int lrint(double x);
long int lrintl(long double x);
long int lrintf(float x);
long int lrintf16(_Float16 x);
```

lround

Description: The `lround` function returns the rounded integer value as a `long int`. Halfway cases are rounded away from zero.

errno: ERANGE, for values too large

Calling interface:

```
long int lround(double x);
long int lroundl(long double x);
long int lroundf(float x);
```

```
long int lroundf16(_Float16 x);
```

modf

Description: The `modf` function returns the value of the signed fractional part of `x` and stores the integral part at `*iptr` as a floating-point number.

Calling interface:

```
double modf(double x, double *iptr);
long double modfl(long double x, long double *iptr);
float modff(float x, float *iptr);
_Float16 modff16(_Float16 x, _Float16 *iptr);
```

nearbyint

Description: The `nearbyint` function returns the rounded integral value as a floating-point number, using the current rounding direction.

Calling interface:

```
double nearbyint(double x);
long double nearbyintl(long double x);
float nearbyintf(float x);
_Float16 nearbyintf16(_Float16 x);
```

rint

Description: The `rint` function returns the rounded integral value as a floating-point number, using the current rounding direction.

Calling interface:

```
double rint(double x);
long double rintl(long double x);
float rintf(float x);
_Float16 rintf16(_Float16 x);
```

round

Description: The `round` function returns the nearest integral value as a floating-point number. Halfway cases are rounded away from zero.

Calling interface:

```
double round(double x);
long double roundl(long double x);
float roundf(float x);
_Float16 roundf16(_Float16 x);
```

trunc

Description: The `trunc` function returns the truncated integral value as a floating-point number.

Calling interface:

```
double trunc(double x);
long double truncl(long double x);
float truncf(float x);
_Float16 truncf16(_Float16 x);
```

Remainder Functions

Many routines in the Intel® oneAPI DPC++/C++ Compiler Math Library are more optimized for Intel® microprocessors than for non-Intel microprocessors.

The `mathimf.h` header file includes prototypes for Intel® oneAPI DPC++/C++ Compiler Math Library functions.

The math library supports the following remainder functions:

> **NOTE**
> FP16 Math Functions have the following requirements:
>
> - Version 2021.4 or higher of the Intel® oneAPI DPC++/C++ Compiler.
> - A next-generation Intel® Xeon® Scalable processor, code name Sapphire Rapids.

fmod

Description: The `fmod` function returns the value `x-n*y` for integer `n` such that if `y` is nonzero, the result has the same sign as `x` and magnitude less than the magnitude of `y`.

errno: EDOM, for y = 0

Calling interface:

```
double fmod(double x, double y);
long double fmodl(long double x, long double y);
float fmodf(float x, float y);
_Float16 fmodf16(_Float16 x, _Float16 y);
```

remainder

Description: The `remainder` function returns the value of `x REM y` as required by the IEEE standard.

errno: EDOM, for y = 0

Calling interface:

```
double remainder(double x, double y);
long double remainderl(long double x, long double y);
float remainderf(float x, float y);
_Float16 remainderf16(_Float16 x, _Float16 y);
```

remquo

Description: The `remquo` function returns the value of `x REM y`. In the object pointed to by `quo` the function stores a value whose sign is the sign of `x/y` and whose magnitude is congruent modulo 2^n of the integral quotient of `x/y`. `N` is an implementation-defined integer. For all systems, `N` is equal to 31.

errno: EDOM, for y = 0

Calling interface:

```
double remquo(double x, double y, int *quo);
long double remquol(long double x, long double y, int *quo);
float remquof(float x, float y, int *quo);
_Float16 remquof16(_Float16 x, _Float16 y, int *quo);
```

Miscellaneous Functions

Many routines in the Intel® oneAPI DPC++/C++ Compiler Math Library are more optimized for Intel® microprocessors than for non-Intel microprocessors.

The `mathimf.h` header file includes prototypes for Intel® oneAPI DPC++/C++ Compiler Math Library functions.

The math library supports the following miscellaneous functions:

> **NOTE**
> FP16 Math Functions have the following requirements:
>
> - Version 2021.4 or higher of the Intel® oneAPI DPC++/C++ Compiler.
> - A next-generation Intel® Xeon® Scalable processor, code name Sapphire Rapids.

copysign

Description: The `copysign` function returns the value with the magnitude of `x` and the sign of `y`.

Calling interface:

```
double copysign(double x, double y);
long double copysignl(long double x, long double y);
float copysignf(float x, float y);
_Float16 copysignf16(_Float16 x, _Float16 y);
```

fabs

Description: The `fabs` function returns the absolute value of `x`.

Calling interface:

```
double fabs(double x);
long double fabsl(long double x);
float fabsf(float x);
_Float16 fabsf16(_Float16 x);
```

fdim

Description: The `fdim` function returns the positive difference value, `x-y` (for `x > y`) or +0 (for `x <= to y`).

errno: ERANGE, for overflow conditions

Calling interface:

```
double fdim(double x, double y);
long double fdiml(long double x, long double y);
float fdimf(float x, float y);
_Float16 fdimf16(_Float16 x, _Float16 y);
```

finite

Description: The `finite` function returns 1 if `x` is not a `NaN` or +/- infinity. Otherwise 0 is returned.

Calling interface:

```
int finite(double x);
int finitel(long double x);
int finitef(float x);
```

```
int finitef16(_Float16 x);
```

fma

Description: The `fma` functions return `(x*y)+z`.

Calling interface:

```
double fma(double x, double y, double z);
long double fmal(long double x, long double y, long double z);
float fmaf(float x, float y, float z);
_Float16 fmaf16(_Float16 x, _Float16 y, _Float16 z);
```

fmax

Description: The `fmax` function returns the maximum numeric value of its arguments.

Calling interface:

```
double fmax(double x, double y);
long double fmaxl(long double x, long double y);
float fmaxf(float x, float y);
_Float16 fmaxf16(_Float16 x, _Float16 y);
```

fmin

Description: The `fmin` function returns the minimum numeric value of its arguments.

Calling interface:

```
double fmin(double x, double y);
long double fminl(long double x, long double y);
float fminf(float x, float y);
_Float16 fminf16(_Float16 x, _Float16 y);
```

fpclassify

Description: The `fpclassify` function returns the value of the number classification macro appropriate to the value of its argument.

Return Value
0 (NaN)
1 (Infinity)
2 (Zero)
3 (Subnormal)
4 (Finite)

Calling interface:

```
int fpclassify(double x);
int fpclassifyl(long double x);
int fpclassifyf(float x);
int fpclassifyf16(_Float16 x);
```

isfinite

Description: The `isfinite` function returns 1 if `x` is not a NaN or +/- infinity. Otherwise 0 is returned.

Calling interface:

```
int isfinite(double x);
int isfinitel(long double x);
int isfinitef(float x);
int isfinitef16(_Float16 x);
```

isgreater

Description: The `isgreater` function returns 1 if `x` is greater than `y`. This function does not raise the invalid floating-point exception.

Calling interface:

```
int isgreater(double x, double y);
int isgreaterl(long double x, long double y);
int isgreaterf(float x, float y);
int isgreaterf16(_Float16 x, _Float16 y);
```

isgreaterequal

Description: The `isgreaterequal` function returns 1 if `x` is greater than or equal to `y`. This function does not raise the invalid floating-point exception.

Calling interface:

```
int isgreaterequal(double x, double y);
int isgreaterequall(long double x, long double y);
int isgreaterequalf(float x, float y);
int isgreaterequalf16(_Float16 x, _Float16 y);
```

isinf

Description: The `isinf` function returns a non-zero value if and only if its argument has an infinite value.

Calling interface:

```
int isinf(double x);
int isinfl(long double x);
int isinff(float x);
int isinff16(_Float16 x);
```

isless

Description: The `isless` function returns 1 if `x` is less than `y`. This function does not raise the invalid floating-point exception.

Calling interface:

```
int isless(double x, double y);
int islessl(long double x, long double y);
int islessf(float x, float y);
int islessf16(_Float16 x, _Float16 y);
```

islessequal

Description: The `islessequal` function returns 1 if `x` is less than or equal to `y`. This function does not raise the invalid floating-point exception.

Calling interface:

```
int islessequal(double x, double y);
int islessequall(long double x, long double y);
```

```
int islessequalf(float x, float y);
int islessequalf16(_Float16 x, _Float16 y);
```

islessgreater

Description: The `islessgreater` function returns 1 if `x` is less than or greater than `y`. This function does not raise the invalid floating-point exception.

Calling interface:

```
int islessgreater(double x, double y);
int islessgreaterl(long double x, long double y);
int islessgreaterf(float x, float y);
int islessgreaterf16(_Float16 x, _Float16 y);
```

isnan

Description: The `isnan` function returns a non-zero value, if and only if `x` has a NaN value.

Calling interface:

```
int isnan(double x);
int isnanl(long double x);
int isnanf(float x);
int isnanf16(_Float16 x);
```

isnormal

Description: The `isnormal` function returns a non-zero value, if and only if `x` is normal.

Calling interface:

```
int isnormal(double x);
int isnormall(long double x);
int isnormalf(float x);
int isnormalf16(_Float16 x);
```

isunordered

Description: The `isunordered` function returns 1 if either `x` or `y` is a NaN. This function does not raise the invalid floating-point exception.

Calling interface:

```
int isunordered(double x, double y);
int isunorderedl(long double x, long double y);
int isunorderedf(float x, float y);
int isunorderedf16(_Float16 x, _Float16 y);
```

maxmag

Description: The `maxmag` function returns the value of larger magnitude from among its two arguments, `x` and `y`. If `|x| > |y|` it returns `x`; if `|y| > |x|` it returns `y`; otherwise it behaves like `fmax(x,y)`.

Calling interface:

```
double maxmag(double x, double y);
float maxmagf(float x, float y);
_Float16 maxmagf16(_Float16 x, _Float16 y);
```

minmag

Description: The `minmag` function returns the value of smaller magnitude from among its two arguments, `x` and `y`. If `|x| < |y|` it returns `x`; if `|y| < |x|` it returns `y`; otherwise it behaves like `fmin(x,y)`.

Calling interface:

```
double minmag(double x, double y);
float minmagf(float x, float y);
_Float16 maxmagf16(_Float16 x, _Float16 y);
```

nan

Description: The `nan` function returns a quiet NaN, with content indicated through `tagp`.

Calling interface:

```
double nan(const char *tagp);
long double nanl(const char *tagp);
float nanf(const char *tagp);
_Float16 nanf16(const char *tagp);
```

nextafter

Description: The `nextafter` function returns the next representable value in the specified format after `x` in the direction of `y`.

errno: ERANGE, for overflow and underflow conditions

Calling interface:

```
double nextafter(double x, double y);
long double nextafterl(long double x, long double y);
float nextafterf(float x, float y);
_Float16 nextafterf16(_Float16 x, _Float16 y);
```

nexttoward

Description: The `nexttoward` function returns the next representable value in the specified format after `x` in the direction of `y`. If `x` equals `y`, then the function returns `y` converted to the type of the function. Use the `Qlong-double` option on Windows operating systems for accurate results.

errno: ERANGE, for overflow and underflow conditions

Calling interface:

```
double nexttoward(double x, long double y);
long double nexttowardl(long double x, long double y);
float nexttowardf(float x, long double y);
_Float16 nexttowardf16(_Float16 x, long double y);
```

signbit

Description: The `signbit` function returns a non-zero value, if and only if the sign of `x` is negative.

Calling interface:

```
int signbit(double x);
int signbitl(long double x);
int signbitf(float x);
```

significand

Description: The `significand` function returns the significand of `x` in the interval [1,2). For `x` equal to zero, NaN, or +/- infinity, the original `x` is returned.

Calling interface:
```
double significand(double x);
long double significandl(long double x);
float significandf(float x);
_Float16 significandf16(_Float16 x);
```

Complex Functions

Many routines in the Intel® oneAPI DPC++/C++ Compiler Math Library are more optimized for Intel® microprocessors than for non-Intel microprocessors.

The `mathimf.h` header file includes prototypes for Intel® oneAPI DPC++/C++ Compiler Math Library functions.

The math library supports the following complex functions:

cabs

Description: The `cabs` function returns the complex absolute value of `z`.

Calling interface:
```
double cabs(double _Complex z);
long double cabsl(long double _Complex z);
float cabsf(float _Complex z);
```

cacos

Description: The `cacos` function returns the complex inverse cosine of `z`.

Calling interface:
```
double _Complex cacos(double _Complex z);
long double _Complex cacosl(long double _Complex z);
float _Complex cacosf(float _Complex z);
```

cacosh

Description: The `cacosh` function returns the complex inverse hyperbolic cosine of `z`.

Calling interface:
```
double _Complex cacosh(double _Complex z);
long double _Complex cacoshl(long double _Complex z);
float _Complex cacoshf(float _Complex z);
```

carg

Description: The `carg` function returns the value of the argument in the interval [-pi, +pi].

Calling interface:
```
double carg(double _Complex z);
long double cargl(long double _Complex z);
float cargf(float _Complex z);
```

casin

Description: The `casin` function returns the complex inverse sine of `z`.

Calling interface:
```
double _Complex casin(double _Complex z);
long double _Complex casinl(long double _Complex z);
float _Complex casinf(float _Complex z);
```

casinh

Description: The `casinh` function returns the complex inverse hyperbolic sine of `z`.

Calling interface:
```
double _Complex casinh(double _Complex z);
long double _Complex casinhl(long double _Complex z);
float _Complex casinhf(float _Complex z);
```

catan

Description: The `catan` function returns the complex inverse tangent of `z`.

Calling interface:
```
double _Complex catan(double _Complex z);
long double _Complex catanl(long double _Complex z);
float _Complex catanf(float _Complex z);
```

catanh

Description: The `catanh` function returns the complex inverse hyperbolic tangent of `z`.

Calling interface:
```
double _Complex catanh(double _Complex z);
long double _Complex catanhl(long double _Complex z);
float _Complex catanhf(float _Complex z);
```

ccos

Description: The `ccos` function returns the complex cosine of `z`.

Calling interface:
```
double _Complex ccos(double _Complex z);
long double _Complex ccosl(long double _Complex z);
float _Complex ccosf(float _Complex z);
```

ccosh

Description: The `ccosh` function returns the complex hyperbolic cosine of `z`.

Calling interface:
```
double _Complex ccosh(double _Complex z);
long double _Complex ccoshl(long double _Complex z);
float _Complex ccoshf(float _Complex z);
```

cexp

Description: The `cexp` function returns e^z (`e` raised to the power `z`).

Calling interface:
```
double _Complex cexp(double _Complex z);
long double _Complex cexpl(long double _Complex z);
float _Complex cexpf(float _Complex z);
```

cexp2

Description: The `cexp` function returns 2^z (2 raised to the power `z`).

Calling interface:

```
double _Complex cexp2(double _Complex z);
long double _Complex cexp2l(long double _Complex z);
float _Complex cexp2f(float _Complex z);
```

cexp10

Description: The `cexp10` function returns 10^z (10 raised to the power `z`).

Calling interface:

```
double _Complex cexp10(double _Complex z);
long double _Complex cexp10l(long double _Complex z);
float _Complex cexp10f(float _Complex z);
```

cimag

Description: The `cimag` function returns the imaginary part value of `z`.

Calling interface:

```
double cimag(double _Complex z);
long double cimagl(long double _Complex z);
float cimagf(float _Complex z);
```

cis

Description: The `cis` function returns the cosine and sine (as a complex value) of `z` measured in radians.

Calling interface:

```
double _Complex cis(double x);
long double _Complex cisl(long double z);
float _Complex cisf(float z);
```

cisd

Description: The `cisd` function returns the cosine and sine (as a complex value) of `z` measured in degrees.

Calling interface:

```
double _Complex cisd(double x);
long double _Complex cisdl(long double z);
float _Complex cisdf(float z);
```

clog

Description: The `clog` function returns the complex natural logarithm of `z`.

Calling interface:

```
double _Complex clog(double _Complex z);
long double _Complex clogl(long double _Complex z);
float _Complex clogf(float _Complex z);
```

clog2

Description: The `clog2` function returns the complex logarithm base 2 of `z`.

Calling interface:

```
double _Complex clog2(double _Complex z);
```

```
long double _Complex clog2l(long double _Complex z);
float _Complex clog2f(float _Complex z);
```

clog10

Description: The `clog10` function returns the complex logarithm base 10 of `z`.

Calling interface:

```
double _Complex clog10(double _Complex z);
long double _Complex clog10l(long double _Complex z);
float _Complex clog10f(float _Complex z);
```

conj

Description: The `conj` function returns the complex conjugate of `z` by reversing the sign of its imaginary part.

Calling interface:

```
double _Complex conj(double _Complex z);
long double _Complex conjl(long double _Complex z);
float _Complex conjf(float _Complex z);
```

cpow

Description: The `cpow` function returns the complex power function, x^y.

Calling interface:

```
double _Complex cpow(double _Complex x, double _Complex y);
long double _Complex cpowl(long double _Complex x, long double _Complex y);
float _Complex cpowf(float _Complex x, float _Complex y);
```

cproj

Description: The `cproj` function returns a projection of `z` onto the Riemann sphere.

Calling interface:

```
double _Complex cproj(double _Complex z);
long double _Complex cprojl(long double _Complex z);
float _Complex cprojf(float _Complex z);
```

creal

Description: The `creal` function returns the real part of `z`.

Calling interface:

```
double creal(double _Complex z);
long double creall(long double _Complex z);
float crealf(float _Complex z);
```

csin

Description: The `csin` function returns the complex sine of `z`.

Calling interface:

```
double _Complex csin(double _Complex z);
long double _Complex csinl(long double _Complex z);
float _Complex csinf(float _Complex z);
```

csinh

Description: The `csinh` function returns the complex hyperbolic sine of `z`.

Calling interface:

```
double _Complex csinh(double _Complex z);
long double _Complex csinhl(long double _Complex z);
float _Complex csinhf(float _Complex z);
```

csqrt

Description: The `csqrt` function returns the complex square root of `z`.

Calling interface:

```
double _Complex csqrt(double _Complex z);
long double _Complex csqrtl(long double _Complex z);
float _Complex csqrtf(float _Complex z);
```

ctan

Description: The `ctan` function returns the complex tangent of `z`.

Calling interface:

```
double _Complex ctan(double _Complex z);
long double _Complex ctanl(long double _Complex z);
float _Complex ctanf(float _Complex z);
```

ctanh

Description: The `ctanh` function returns the complex hyperbolic tangent of `z`.

Calling interface:

```
double _Complex ctanh(double _Complex z);
long double _Complex ctanhl(long double _Complex z);
float _Complex ctanhf(float _Complex z);
```

C99 Macros

Many routines in the Intel® oneAPI DPC++/C++ Compiler Math Library are more optimized for Intel® microprocessors than for non-Intel microprocessors.

The `mathimf.h` header file includes prototypes for Intel® oneAPI DPC++/C++ Compiler Math Library functions.

The math library and `mathimf.h` header file support the following C99 macros:

- `int fpclassify(x)`
- `int isfinite(x)`
- `int isgreater(x, y)`
- `int isgreaterequal(x, y)`
- `int isinf(x)`
- `int isless(x, y)`
- `int islessequal(x, y)`
- `int islessgreater(x, y)`
- `int isnan(x)`
- `int isnormal(x)`
- `int isunordered(x, y)`
- `int signbit(x)`

See Also

Miscellaneous Functions

Part VI

Compatibility and Portability

This section contains information about conformance to language standards, language compatibility, and portability.

Standards Conformance

Conformance to C/C++ Standards

The Intel® oneAPI DPC++/C++ Compiler conforms to the following C/C++ standards:

- C++17 standard (ISO/IEC 14882:2017)
- C++14 standard (ISO/IEC 14882:2014)
- C++11 standard (ISO/IEC 14882:2011)
- C++98 standard (ISO/IEC 14882:1998)
- C11 standard (ISO/IEC 9899:2011)
- C99 standard (ISO/IEC 9899:1999)

Conformance to SYCL Standards

The Intel® oneAPI DPC++ Compiler supports the SYCL 2020 Specification and work is in progress towards SYCL 2020 conformance. The SYCL standard is based on the C++ standard and the Intel® oneAPI DPC++/C++ Compiler headers include some of the C++ standard headers. All of the current restrictions and limitations that apply to C/C++ standards, which relate to library headers, also apply to SYCL headers.

GCC Compatibility and Interoperability

GCC Compatibility

The Intel® oneAPI DPC++/C++ Compiler is compatible with most versions of the GNU Compiler Collection (GCC). The release notes contains a list of compatible versions.

C language object files created with the compiler are binary compatible with the GCC and C/C++ language library. You can use the Intel® oneAPI DPC++/C++ Compiler or the GCC compiler to pass object files to the linker.

NOTE When using an Intel software development product that includes a compiler with a Clang front-end, you can also use `icx` or `icpx`.

The Intel® oneAPI DPC++/C++ Compiler supports many of the language extensions provided by the GNU compilers.

Statement expressions are supported, except that the following are prohibited inside them:

- Dynamically-initialized local static variables
- Local non-POD class definitions
- Try/catch
- Variable length arrays

Branching out of a statement expression and statement expressions in constructor initializers are not allowed. Variable-length arrays are no longer allowed in statement expressions.

The Intel® oneAPI DPC++/C++ Compiler supports GCC-style inline ASM if the assembler code uses AT&T* System V/386 syntax.

GCC Interoperability

C++ compilers are interoperable if they can link object files and libraries generated by one compiler with object files and libraries generated by the second compiler, and the resulting executable runs successfully. The Intel® oneAPI DPC++/C++ Compiler is highly compatible with the GNU compilers.

The Intel® oneAPI DPC++/C++ Compiler and GCC support the following predefined macros:

- `__GNUC__`
- `__GNUG__`
- `__GNUC_MINOR__`
- `__GNUC_PATCHLEVEL__`

Caution Not defining these macros results in different paths through system header files. These alternate paths may be poorly tested or otherwise incompatible.

How the Compiler Uses GCC

The Intel® oneAPI DPC++/C++ Compiler uses the GNU tools on the system, such as the GNU header files, including `stdio.h`, and the GNU linker and libraries. So the compiler has to be compatible with the version of GCC or G++* you have on your system.

By default, the compiler determines which version of GCC or G++ you have installed from the `PATH` environment variable.

If you want use a version of GCC or G++ other than the default version on your system, you need to use the `--gcc-toolchain` compiler option to specify the location of the base toolchain. For example:

- You want to build something that cannot be compiled by the default version of the system compiler, so you need to use a legacy version for compatibility, such as if you want to use third party libraries that are not compatible with the default version of the system compiler.
- You want to use a later version of GCC or G++ than the default system compiler.

The Intel® oneAPI DPC++/C++ Compiler driver uses the default version of GCC/G++, or the version you specify, to extract the location of the headers and libraries.

Compatibility with Open Source Tools

The Intel® oneAPI DPC++/C++ Compiler includes improved support for the following open source tools:

- **GNU Libtool**: A script that allows package developers to provide generic shared library support.
- **Valgrind**: A flexible system for debugging and profiling executables running on x86 processors.
- **GNU Automake**: A tool for automatically generating `Makefile.ins` from files called `Makefile.am`.

Microsoft Compatibility

The Intel® oneAPI DPC++/C++ Compiler is fully source- and binary-compatible (native code only) with Microsoft Visual C++ (MSVC). You can debug binaries built with the Intel® oneAPI DPC++/C++ Compiler from within the Microsoft Visual Studio environment.

The compiler supports security checks with the /GS option. You can control this option in the Microsoft Visual Studio IDE by using **C/C++** > **Code Generation** > **Buffer Security Check**.

Microsoft Visual Studio Integration

The compiler is compatible with Microsoft Visual Studio 2017, 2019, and 2022 projects.

> **NOTE** Support for Microsoft Visual Studio 2017 is deprecated as of the Intel® oneAPI 2022.1 release, and will be removed in a future release.

Unsupported Features

Unsupported project types:

- .NET-based CLR C++ project types are not supported by the Intel® oneAPI DPC++/C++ Compiler. The specific project types will vary depending on your version of Visual Studio, for example:
  - CLR Class Library
  - CLR Console App
  - CLR Empty Project

Unsupported major features:

- COM Attributes
- C++ Accelerated Massive Parallelism (C++ AMP)
- Managed extensions for C++ (new pragmas, keywords, and command-line options)
- Event handling (new keywords)
- Select keywords:
  - `__abstract`
  - `__box`
  - `__delegate`
  - `__gc`
  - `__identifier`
  - `__nogc`
  - `__pin`
  - `__property`
  - `__sealed`
  - `__try_cast`
  - `__w64`

Unsupported preprocessor features:

- `#import` directive changes for attributed code
- `#using` directive

- managed, unmanaged pragmas
- _MANAGED macro
- runtime_checks pragma

Mixing Managed and Unmanaged Code

If you use the managed extensions to the C++ language in Microsoft Visual Studio .NET, you can use the compiler for your non-managed code for better application performance. Make sure managed keywords do not appear in your non-managed code.

For information on how to mix managed and unmanaged code, refer to the article, An Overview of Managed/Unmanaged Code Interoperability, on the Microsoft Web site.

Precompiled Header Support

There are some differences in how precompiled header (PCH) files are supported between the Intel® oneAPI DPC++/C++ Compiler and the Microsoft Visual C++ Compiler:

- The PCH information generated by the Intel oneAPI DPC++/C++ Compiler is not compatible with the PCH information generated by the Microsoft Visual Studio Compiler.
- The Intel oneAPI DPC++/C++ Compiler does not support PCH generation and use in the same translation unit.

Compilation and Execution Differences

While the Intel® oneAPI DPC++/C++ Compiler is compatible with the Microsoft Visual C++ Compiler, some differences can prevent successful compilation. There can also be some incompatible generated-code behavior of some source files with the Intel oneAPI DPC++/C++ Compiler. In most cases, a modification of the user source file enables successful compilation with both the Intel oneAPI DPC++/C++ Compiler and the Microsoft Visual C++ Compiler. The differences between the compilers are:

- **Inline Assembly Target Labels (IA-32 Architecture Only)**

  IA-32 applications do not apply for SYCL.

  For compilations targeted for IA-32 architecture, inline assembly target labels of goto statements are case sensitive. The Microsoft Visual C++ compiler treats these labels in a case insensitive manner. For example, the Intel oneAPI DPC++/C++ Compiler issues an error when compiling the following code:

```
int func(int x) {
   goto LAB2;
     // error: label "LAB2" was referenced but not defined
   __asm lab2: mov x, 1
   return x;
}
```

  However, the Microsoft Visual C++ Compiler accepts the preceding code. As a work-around for the Intel oneAPI DPC++/C++ Compiler, when a goto statement refers to a label defined in inline assembly, you must match the label reference with the label definition in both name and case.
- **Inlining Functions Marked for dllimport**

  The Intel oneAPI DPC++/C++ Compiler will attempt to inline any functions that are marked dllimport but Microsoft will not. Therefore, any calls or variables used inside a dllimport routine need to be available at link time or the result will be an unresolved symbol.

  The following example contains two files: header.h and bug.cpp.

  header.h:

```
#ifndef _HEADER_H
#define _HEADER_H
namespace Foo_NS {
```

```
        class Foo2 {
        public:
                Foo2(){};
                ~Foo2();
                static int test(int m_i);
        };
}
#endif
```

bug.cpp:

```
#include "header.h"
struct Foo2 {
  static void test();
};

struct __declspec(dllimport) Foo
{
   void getI() { Foo2::test(); };
};

struct C  {
  virtual void test();
};

void C::test() { Foo p;  p->getI(); }

int main() {
   return 0;
}
```

Enum Bit-Field Signedness

The Intel® oneAPI DPC++/C++ Compiler and Microsoft Visual C++ differ in how they attribute signedness to bit fields declared with an `enum` type. Microsoft Visual C++ always considers `enum` bit fields to be signed, even if not all values of the `enum` type can be represented by the bit field.

The Intel oneAPI DPC++/C++ Compiler considers an `enum` bit field to be unsigned, unless the `enum` type has at least one `enum` constant with a negative value. In any case, the Intel oneAPI DPC++/C++ Compiler produces a warning if the bit field is declared with too few bits to represent all the values of the `enum` type.

See Also

/GS compiler option

Port from Microsoft Visual C++ to the Intel® oneAPI DPC++/C++ Compiler*

This section describes a basic approach to porting applications from Microsoft Visual C++* for Windows* to the Intel® oneAPI DPC++/C++ Compiler for Windows.

If you build your applications from the Windows command line, you can port applications from Microsoft Visual C++ to the Intel® oneAPI DPC++/C++ Compiler by modifying your makefile to invoke the Intel® oneAPI DPC++/C++ Compiler instead of Microsoft Visual C++.

The Intel® oneAPI DPC++/C++ Compiler integration with Microsoft Visual Studio provides a conversion path to the Intel® oneAPI DPC++/C++ Compiler that allows you to build your Visual C++ projects with the Intel® oneAPI DPC++/C++ Compiler. This version of the Intel® oneAPI DPC++/C++ Compiler supports:

- Microsoft Visual Studio 2022
- Microsoft Visual Studio 2019
- Microsoft Visual Studio 2017

> **NOTE** Support for Microsoft Visual Studio 2017 is deprecated as of the Intel® oneAPI 2022.1 release, and will be removed in a future release.

See the appropriate section in this documentation for details on using the Intel® oneAPI DPC++/C++ Compiler with Microsoft Visual Studio.

The Intel® oneAPI DPC++/C++ Compiler also supports many of the same compiler options, macros, and environment variables you already use in your Microsoft work.

One challenge in porting applications from one compiler to another is making sure there is support for the compiler options you use to build your application. The *Compiler Options* reference lists compiler options that are supported by both the Intel® oneAPI DPC++/C++ Compiler and Microsoft C++.

See Also

Other Considerations
Modify Your Makefile

Modify Your makefile

If you use makefiles to build your Microsoft* application, you need to change the value for the compiler variable to use the Intel® oneAPI DPC++/C++ Compiler. You may also want to review the options specified by `CPPFLAGS`. For example, a sample Microsoft makefile:

```
# name of the program
  PROGRAM = area.exe

# names of source files
  CPPSOURCES = area_main.cpp area_functions.cpp

# names of object files
  CPPOBJECTS = area_main.obj area_functions.obj

# Microsoft(R) compiler options
  CPPFLAGS = /RTC1 /EHsc

# Use Microsoft C++(R)
  CPP = cl

# link objects
  $(PROGRAM): $(CPPOBJECTS)
     link.exe /out:$@ $(CPPOBJECTS)

# build objects
  area_main.obj: area_main.cpp area_headers.h
  area_functions.obj: area_functions.cpp area_headers.h
```

```
# clean
  clean:  del  $(CPPOBJECTS) $(PROGRAM)
```

Modified makefile for the Intel® oneAPI DPC++/C++ Compiler

Before you can run `nmake` with the Intel® oneAPI DPC++/C++ Compiler, you need to set the proper environment. In this example, only the name of the compiler changed to use icx:

```
# name of the program
  PROGRAM = area.exe

# names of source files
  CPPSOURCES = area_main.cpp area_functions.cpp

# names of object files
  CPPOBJECTS = area_main.obj area_functions.obj

# Intel(R) C/C++/DPC++ Compiler options
  CPPFLAGS = /RTC1 /EHsc

# Use the Intel C/C++/DPC++ Compiler
  CPP = icx

# link objects
  $(PROGRAM): $(CPPOBJECTS)
     link.exe /out:$@ $(CPPOBJECTS)

# build objects
  area_main.obj: area_main.cpp area_headers.h
  area_functions.obj: area_functions.cpp area_headers.h

# clean
  clean:  del  $(CPPOBJECTS) $(PROGRAM)
```

With the modified makefile, the output of `nmake` is similar to the following:

```
Microsoft (R) Program Maintenance Utility Version 8.00.50727.42
Copyright (C) Microsoft Corporation. All rights reserved.

        icx /RTC1 /EHsc  /c area_main.cpp area_functions.cpp

Intel(R) Compiler for applications running on IA-32 or IA-64
Copyright (C) 1985-2006 Intel Corporation. All rights reserved.

area_main.cpp
area_functions.cpp
        link.exe /out:area.exe area_main.obj area_functions.obj

Microsoft (R) Incremental Linker Version 8.00.50727.42
Copyright (C) Microsoft Corporation. All rights reserved.
```

Use IPO in makefiles

By default, IPO generates dummy object files containing interprocedural information used by the compiler. To link or create static libraries with these object files requires specific Intel-provided tools. To use them in your makefile, replace references to `link` with `xilink` and references to `lib` with `xilib`. For example:

```
 # name of the program
  PROGRAM = area.exe

# names of source files
  CPPSOURCES = area_main.cpp area_functions.cpp

# names of object files
  CPPOBJECTS = area_main.obj area_functions.obj

# Intel C/C++/DPC++ Compiler options
  CPPFLAGS = /RTC1 /EHsc /Qipo

# Use the Intel C/C++/DPC++ Compiler
  CPP = icx

# link objects
  $(PROGRAM): $(CPPOBJECTS)
     xilink.exe /out:$@ $(CPPOBJECTS)

# build objects
  area_main.obj: area_main.cpp area_headers.h
  area_functions.obj: area_functions.cpp area_headers.h

# clean
  clean:  del  $(CPPOBJECTS) $(PROGRAM)
```

Other Considerations

There are some notable differences between the Intel® oneAPI DPC++/C++ Compiler and the Microsoft* Compiler. Consider the following as you begin compiling your code with the Intel® oneAPI DPC++/C++ Compiler.

Set the Environment

The compiler installation provides a batch file, `setvars.bat`, that sets the proper environment for the Intel® oneAPI DPC++/C++ Compiler. For information on running `setvars.bat`, see Specifying the Location of Compiler Components.

Use Optimization

The Intel® oneAPI DPC++/C++ Compiler is an optimizing compiler that begins with the assumption that you want improved performance from your application when it is executed on Intel® architecture. Consequently, certain optimizations, such as option `O2`, are part of the default invocation of the compiler. By default, Microsoft turns off optimization, which is the equivalent of compiling with options `Od` or `O0`. Other forms of the `O`[*n*] option compare as follows:

Option	Intel® oneAPI DPC++/C++ Compiler	Microsoft Compiler
`/Od`	Turns off all optimization. Same as `O0`.	Default. Turns off all optimization.

Option	Intel® oneAPI DPC++/C++ Compiler	Microsoft Compiler
`/O1`	Decreases code size with some increase in speed.	Optimizes code for minimum size.
`/O2`	Default. Favors speed optimization with some increase in code size. Intrinsics, loop unrolling, and inlining are performed.	Optimizes code for maximum speed.
`/O3`	Enables `-O2` optimizations plus more aggressive optimizations, such as prefetching, scalar replacement, and loop and memory access transformations.	Not supported.

Modify Your Configuration

The Intel® oneAPI DPC++/C++ Compiler lets you maintain configuration and response files that are part of compilation. Options stored in the configuration file apply to every compilation, while options stored in response files apply only where they are added on the command line. If you have several options in your makefile that apply to every build, you may find it easier to move these options to the configuration file (`..\bin\icx.cfg`).

In a multi-user, networked environment, options listed in the `icx.cfg` file are generally intended for everyone who uses the compiler. If you need a separate configuration, you can use the `ICXCFG` environment variable to specify the name and location of your own `.cfg` file, such as `\my_code\my_config.cfg`. Anytime you instruct the compiler to use a different configuration file, the `icx.cfg` system configuration file is ignored.

Use the Intel Libraries

The Intel® oneAPI DPC++/C++ Compiler supplies additional libraries that contain optimized implementations of many commonly used functions. Some of these functions are implemented using CPU dispatch. This means that different code may be executed when run on different processors.

Supplied libraries include the Intel® oneAPI DPC++/C++ Compiler (*libm*), the Short Vector Math Library (*svml_disp*), *libirc*, as well as others. These libraries are linked in by default when the compiler sees that references to them have been generated. Some library functions, such as `sin` or `memset`, may not require a call to the library, since the compiler may inline the code for the function.

Intel® oneAPI DPC++/C++ Compiler Math Library (*libm*)

With the Intel® oneAPI DPC++/C++ Compiler, the math library, *libm*, is linked by default when calling math functions that require the library. Some functions, such as `sin`, may not require a call to the library, since the compiler already knows how to compute the `sin` function. The math library also includes some functions not found in the standard math library.

> **NOTE**
> You cannot make calls to the math library with the Microsoft Compiler.

Many routines in the *libimf* library are more optimized for Intel® microprocessors than for non-Intel microprocessors.

Short Vector Math Library (*svml_disp*)

When vectorization is in progress, the compiler may translate some calls to the *libm* math library functions into calls to *svml_disp* functions. These functions implement the same basic operations as the math library, but operate on short vectors of operands. This results in greater efficiency. In some cases, the *svml_disp* functions are slightly less precise than the equivalent *libm* functions.

Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

libirc

libirc contains optimized implementations of some commonly used string and memory functions. For example, it contains functions that are optimized versions of `memcpy` and `memset`. The compiler will automatically generate calls to these functions when it sees calls to `memcpy` and `memset`. The compiler may also transform loops that are equivalent to `memcpy` or `memset` into calls to these functions.

Many routines in the *libirc* library are more optimized for Intel® microprocessors than for non-Intel microprocessors.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

See Also

O compiler option
Using Configuration Files
Using Response Files
Specifying the Location of Compiler Components

Port from GCC* to the Intel® oneAPI DPC++/C++ Compiler

This section describes a basic approach to porting applications from the (GNU Compiler Collection*) GCC C/C++ compilers to the Intel® oneAPI DPC++/C++ Compiler. These compilers correspond to each other as follows:

Language	Intel® Compiler	GCC Compiler
C	`icx` for C++ or `dpcpp` for SYCL.	`gcc`
C++	`icpx` for C++ or `dpcpp` for SYCL.	`g++`

NOTE Unless otherwise indicated, the term "`gcc`" refers to both GCC and G++* compilers from the GCC.

Advantages to Using the Intel® oneAPI DPC++/C++ Compiler

In many cases, porting applications from `gcc` to the Intel® oneAPI DPC++/C++ Compiler can be as easy as modifying your makefile to invoke the Intel® oneAPI DPC++/C++ Compiler (`icx` for C++ or `dpcpp` for SYCL) instead of `gcc`. Using the Intel® oneAPI DPC++/C++ Compiler typically improves the performance of your application, especially for those that run on Intel processors. In many cases, your application's performance may also show improvement when running on non-Intel processors. When you compile your application with the Intel® oneAPI DPC++/C++ Compiler, you have access to:

- Compiler options that optimize your code for the latest Intel® architecture processors.
- Advanced profiling tools (PGO) similar to the GNU profiler `gprof`.
- High-level optimizations (HLO).
- Interprocedural optimization (IPO).
- Intel intrinsic functions that the compiler uses to inline instructions, including various versions of Intel® Streaming SIMD Extensions and Intel® Advanced Vector Extensions.
- Highly-optimized Intel® oneAPI DPC++/C++ Compiler Math Library for improved accuracy.

Because the Intel® oneAPI DPC++/C++ Compiler is compatible and interoperable with `gcc`, porting your `gcc` application to the Intel® oneAPI DPC++/C++ Compiler includes the benefits of binary compatibility. As a result, you should not have to re-build libraries from your `gcc` applications. The Intel® oneAPI DPC++/C++ Compiler also supports many of the same compiler options, macros, and environment variables you already use in your `gcc` work.

Equivalent Macros

The Intel® oneAPI DPC++/C++ Compiler is compatible with the predefined GNU* macros.

See http://gcc.gnu.org for a list of compatible predefined macros.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

See Also

Modify Your makefile
Supported Environment Variables
Additional Predefined Macros

Modify Your makefile

If you use makefiles to build your GCC* application, you need to change the value for the GCC compiler variable to use the Intel® oneAPI DPC++/C++ Compiler. You may also want to review the options specified by `CFLAGS`. For example, a sample GCC makefile:

```
# Use gcc compiler
  CC = gcc

# Compile-time flags
  CFLAGS = -O2 -std=c99
all: area_app

area_app: area_main.o area_functions.o
    $(CC) area_main.o area_functions.o -o area

area_main.o: area_main.c
    $(CC) -c $(CFLAGS) area_main.c

area_functions.o: area_functions.c
    $(CC) -c -fno-asm $(CFLAGS) area_functions.c

clean: rm -rf *o area
```

Modified makefile for the Intel® oneAPI DPC++/C++ Compiler

In this example, the name of the compiler is changed to use icpx

```
# Use Intel C/C++/DPC++ Compiler
  CC = icpx

# Compile-time flags
  CFLAGS = -std=c99
all: area_app

area_app: area_main.o area_functions.o
    $(CC) area_main.o area_functions.o -o area

area_main.o: area_main.c
    $(CC) -c $(CFLAGS) area_main.c

area_functions.o: area_functions.c
    $(CC) -c -fno-asm $(CFLAGS) area_functions.c

clean: rm -rf *o area
```

If your GCC code includes features that are not supported with the Intel® oneAPI DPC++/C++ Compiler (compiler options, language extensions, macros, pragmas, and so on), you can compile those sources separately with GCC if necessary.

In the above makefile, `area_functions.c` is an example of a source file that includes features unique to GCC. Because the Intel® oneAPI DPC++/C++ Compiler uses the `O2` option by default and GCC uses option `O0` as the default, we instruct GCC to compile at option `O2`. We also include the `-fno-asm` switch from the original makefile because this switch is not supported with the Intel® oneAPI DPC++/C++ Compiler. The following sample makefile is modified for using the Intel® oneAPI DPC++/C++ Compiler and GCC together:

```
# Use Intel C/C++/DPC++ Compiler
  CC = icpx
# Use gcc for files that cannot be compiled by icpx
  GCC = gcc
# Compile-time flags
  CFLAGS = -std=c99
all: area_app

area_app: area_main.o area_functions.o
    $(CC) area_main.o area_functions.o -o area

area_main.o: area_main.c
    $(CC) -c $(CFLAGS) area_main.c

area_functions.o: area_functions.c
    $(GCC) -c -O2 -fno-asm $(CFLAGS) area_functions.c

clean: rm -rf *o area
```

Output of `make` using a modified makefile:

```
icpx -c -std=c99 area_main.c
gcc -c -O2 -fno-asm -std=c99 area_functions.c
icpx area_main.o area_functions.o -o area
```

Use IPO in Makefiles

By default, IPO generates "dummy" object files containing Interprocedural information used by the compiler. To link or create static libraries with these object files requires special Intel-provided tools. To use them in your makefile, simply replace references to "ld" with "xild" and references to "ar" with "xiar", or use `icx` or `icpx` (for C++) or `dpcpp` (for SYCL*) to link as shown in the example:

```
# Use Intel C/C++/DPC++ Compiler
  CC = icpx
# Compile-time flags
  CFLAGS = -std=c99 -ipo
all: area_app

area_app: area_main.o area_functions.o
    $(CC) area_main.o area_functions.o -o area

area_main.o: area_main.c
    $(CC) -c $(CFLAGS) area_main.c

area_functions.o: area_functions.c
    $(CC) -c $(CFLAGS) area_functions.c

clean: rm -rf *o area
```

Other Considerations

There are some notable differences between the Intel® oneAPI DPC++/C++ Compiler and GCC*. Consider the following as you begin compiling your source code with the Intel® oneAPI DPC++/C++ Compiler.

Set the Environment

The Intel® oneAPI DPC++/C++ Compiler relies on environment variables for the location of compiler binaries, libraries, man pages, and license files. In some cases these are different from the environment variables that GCC uses. Another difference is that these variables are not set by default after installing the Intel® oneAPI DPC++/C++ Compiler. The following environment variables can be set prior to running the Intel® oneAPI DPC++/C++ Compiler:

- PATH: Adds the location of the compiler binaries to PATH.
- LD_LIBRARY_PATH: Sets the location where the generated executable picks up the runtime libraries (*.so files).
- MANPATH : Adds the location of the compiler man pages (`icx or icpx` for C++ or `dpcpp` for SYCL) to MANPATH.

To set these environment variables, you can source the `setvars.sh` script (e.g. `source setvars.sh`).

> **NOTE**
> Setting these environment variables with `setvars.sh` does not impose a conflict with GCC. You should be able to use both compilers in the same shell.

Use Optimization

The Intel® oneAPI DPC++/C++ Compiler is an optimizing compiler that begins with the assumption that you want improved performance from your application when it is executed on Intel® architecture. Consequently, certain optimizations, such as option `O2`, are part of the default invocation of the Intel® oneAPI DPC++/C++ Compiler. Optimization is turned off in GCC by default, the equivalent of compiling with option `O0`. Other forms of the O<*n*> option compare as follows:

Option	Intel® oneAPI DPC++/C++ Compiler	GCC
`-O0`	Turns off optimization.	Default. Turns off optimization.
`-O1`	Decreases code size with some increase in speed.	Decreases code size with some increase in speed.
`-O2`	**Default.** Favors speed optimization with some increase in code size. Same as option `O`. Intrinsics, loop unrolling, and inlining are performed.	Optimizes for speed as long as there is not an increase in code size. Loop unrolling and function inlining, for example, are not performed.
`-O3`	Enables option `O2` optimizations plus more aggressive optimizations, such as prefetching, scalar replacement, and loop and memory access transformations.	Optimizes for speed while generating larger code size. Includes option `O2` optimizations plus loop unrolling and inlining.

Target Intel® Processors

While many of the same options that target specific processors are supported with both compilers, Intel includes options that utilize processor-specific instruction scheduling to target the latest Intel® processors.

Modify Your Configuration

The Intel® oneAPI DPC++/C++ Compiler lets you maintain configuration and response files that are part of compilation. Options stored in the configuration file apply to every compilation, while options stored in response files apply only where they are added on the command line. If you have several options in your makefile that apply to every build, you may find it easier to move these options to the configuration file (`icx.cfg or icpx.cfg` for C++ or `dpcpp.cfg` for SYCL).

In a multi-user, networked environment, options listed in the `icx.cfg or icpx.cfg` for C++ or `dpcpp.cfg` for SYCL files are generally intended for everyone who uses the compiler. If you need a separate configuration, you can use the `ICXCFG or ICPXCFG` for C++ or `DPCPPCFG` for SYCLenvironment variable to specify the name and location of your own `.cfg` file, such as `/my_code/my_config.cfg`. Anytime you instruct the compiler to use a different configuration file, the system configuration files (`icx.cfg or icpx.cfg` for C++ or `dpcpp.cfg` for SYCL) are ignored.

Use the Intel Libraries

The Intel® oneAPI DPC++/C++ Compiler supplies additional libraries that contain optimized implementations of many commonly used functions. Some of these functions are implemented using CPU dispatch. This means that different code may be executed when run on different processors.

Supplied libraries include the Intel® oneAPI DPC++/C++ Compiler Math Library (*libimf*), the Short Vector Math Library (*libsvml*), *libirc*, as well as others. These libraries are linked in by default. Some library functions, such as `sin` or `memset`, may not require a call to the library, since the compiler may inline the code for the function.

> **NOTE** The Intel Compiler Math Libraries contain performance-optimized implementations for various Intel platforms. By default, the best implementation for the underlying hardware is selected at runtime. The library dispatch of multi-threaded code may lead to apparent data races, which may be detected by certain software analysis tools. However, as long as the threads are running on cores with the same CPUID, these data races are harmless and are not a cause for concern.

Intel® oneAPI DPC++/C++ Compiler Math Library (*libimf*)

With the Intel® Compiler, the math library, *libimf*, is linked by default. Some functions, such as sin, may not require a call to the library, since the compiler already knows how to compute the `sin` function. The math library also includes some functions not found in the standard math library.

NOTE
You cannot make calls to the math library with GCC.

Many routines in the *libimf* library are more optimized for Intel® microprocessors than for non-Intel microprocessors.

Short Vector Math Library (*libsvml*)

When vectorization is being done, the compiler may translate some calls to the *libimf* math library functions into calls to *libsvml* functions. These functions implement the same basic operations as the math library, but operate on short vectors of operands. This results in greater efficiency. In some cases, the *libsvml* functions are slightly less precise than the equivalent *libimf* functions.

Many routines in the *libimf* library are more optimized for Intel® microprocessors than for non-Intel microprocessors.

libirc

libirc contains optimized implementations of some commonly used string and memory functions. For example, it contains functions that are optimized versions of `memcpy` and `memset`. The compiler will automatically generate calls to these functions when it sees calls to `memcpy` and `memset`. The compiler may also transform loops that are equivalent to `memcpy` or `memset` into calls to these functions.

Many routines in the *libirc* library are more optimized for Intel® microprocessors than for non-Intel microprocessors.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

See Also

Invoke the Compiler

march compiler option
o compiler option
Using Configuration Files
Using Response Files

Index

A

B

C

D

E

F

G

H

I

K

L

N

O

P

R

S

T

U

V

W

X