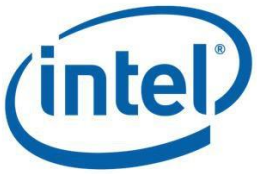
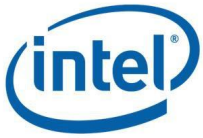


Intel® Media Developer's Guide

Deliver Hardware Accelerated Video Applications on Intel® Platforms

Revision: 2017

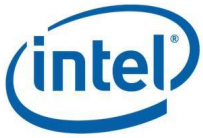
1.1 Legal Information



Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.
Notice revision #20110804

Contents



2 About this Document

2.1 Overview

This document provides architectural details, supplements to the API documentation in the manuals, and coding examples to help developers utilize the amazing capabilities of Intel GPUs for media applications.

The two main paths for application developers to access GPU media processing capabilities are

1. Intel® Media SDK
2. Intel® SDK for OpenCL™ applications

Both are available together in Intel® Media Server Studio for Linux and Windows, and as standalone client packages for Windows.

This guide starts with how to understand the sample applications bundled with installation and expands from there to cover working with the Intel Media SDK from a software developer's perspective. It is intended to accompany the reference manuals

- mediasdk-man.pdf
- mediasdkmvc-man.pdf
- mediasdkusr-man.pdf
- mediasdkjpeg-man.pdf
- mediasdkscreencap-man.pdf
- mediasdk-distrib.pdf
- Media_Samples_Guide.pdf
- mediasdkvp8-man.pdf

to support rapid development of high performance media applications.

2.2 Intended Audience

This document is intended for all developers interested in utilizing the amazing media capabilities of Intel processors with Quicksync video/Gen graphics GPUs. There is significant overlap between the audience for this document and for the series of documents titled "The Compute Architecture of Intel Processor Graphics". That series covers the compute architecture from a general OpenCL™ development perspective, this document focuses on media capabilities.

2.3 Conventions and Symbols

The following conventions are used in this document.

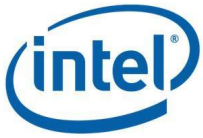
Conventions and Symbols used in this Document

`code`	`// Initialize DECODE`
	`decode->DecodeHeader(bitstream, InitParam_d);`
	`decode->Init(InitParam_d);`

`This type style`	Indicates the exact characters you type as input. Also used to highlight the elements of a graphical user interface such as buttons and menu names.
`[ items ]`	Indicates that the items enclosed in brackets are optional.
`{ item | item }`	Indicates to select only one of the items listed between braces. A vertical bar (|) separates the items.
`... (ellipses)`	Indicates that you can repeat the preceding item.

2.4 Terms Used in this Document

FFMPEG	Open Source video playback and conversion player
Gen Graphics	GPU technology in Intel processors
GPU / Processor Graphics	Graphics Processing Unit. Intel architecture integrates this on the same die with the CPUs. Contains execution units (shaders) and fixed function.
H.264	ISO*/IEC* 14496-10 and ITU-T* H.264, MPEG-4 Part 10, Advanced Video Coding (AVC), May 2005
H.265/HEVC	High Efficiency Video Coding. ISO/IEC 23008-2 MPEG-H Part 2 and ITU-T H.265



HRD	Hypothetical Reference Decoder
Quicksync Video	Presence of Quicksync video indicates Gen graphics GPU on the processor

2.5 Related Information

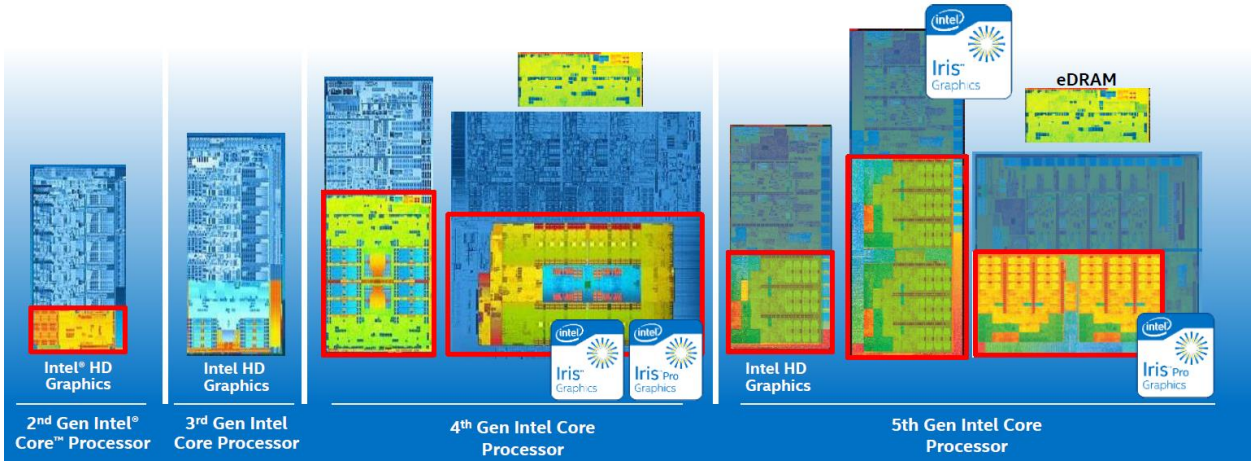
Intel Media SDK Support Forum:
http://software.intel.com/en-us/forums/intel-media
Intel Media SDK product website:
https://software.intel.com/en-us/intel-media-server-studio

https://software.intel.com/en-us/media-client-solutions

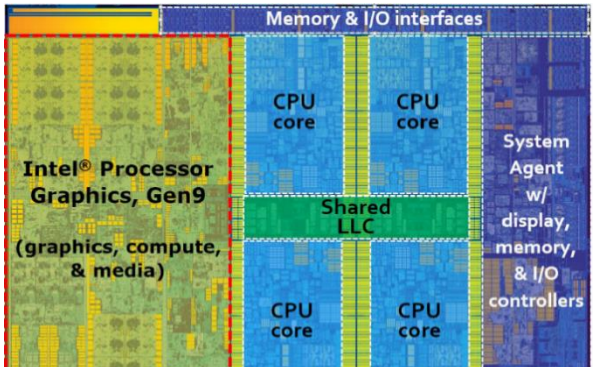
Intel Media SDK development tutorials:
https://software.intel.com/en-us/intel-media-server-studio-support/training

3 Overview of Intel GPU Performance Opportunities

Intel GPUs are an increasingly significant component of overall performance for many types of processors. They go by many different names – integrated graphics, processor graphics, Gen graphics, Intel(R) HD Graphics, Intel® Iris Graphics, etc. All of these mean one thing: heterogeneous performance opportunities for developers. These capabilities have been improving for several years.



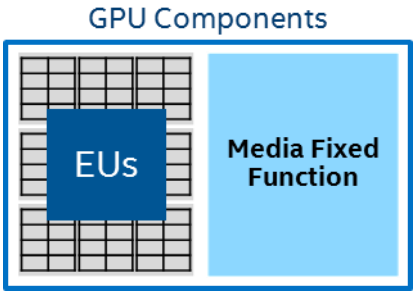
This document focuses on developing for media capabilities with Intel® 6th Generation Core™ and Intel® Xeon® E3 v5 processors (previously code named Skylake architecture). As the processor graphics generation for Skylake is Gen9, this guide extends the content in the document titled "The Compute Architecture of Intel Processor Graphics Gen9" for media.



Components layout for an Intel(R) Core(TM) i7 processor 6700K

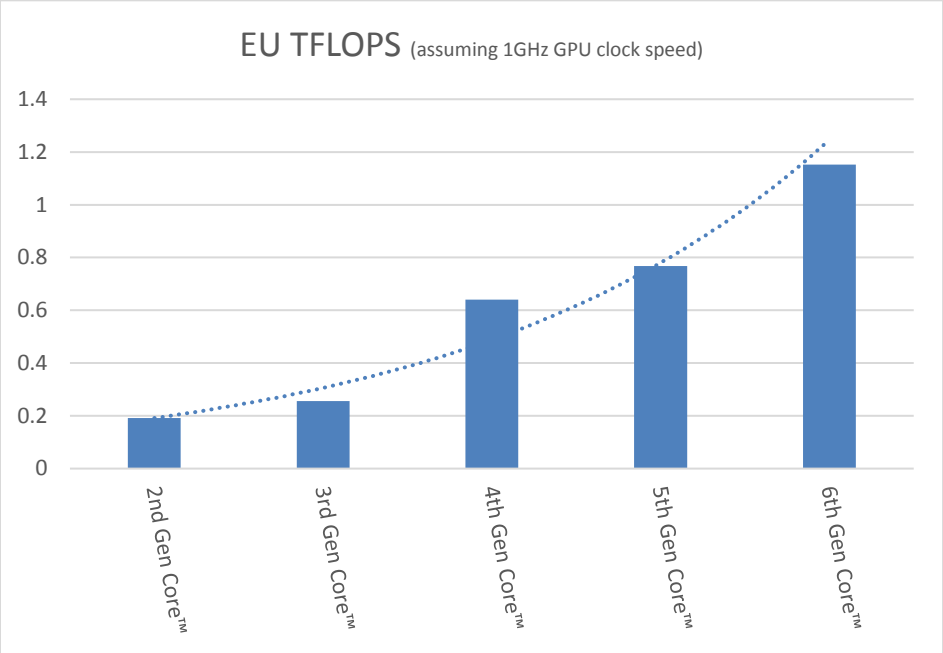
Intel processors are now heterogeneous. Most processors have traditional CPUs as well as very capable GPUs (and often other hardware blocks as well.) Intel processor graphics GPUs have two main groups of hardware of interest to media developers:

- **Execution Units/EUs**: General purpose execution units. These are used for graphics rendering, but they are also suited to a wide range of media processing tasks.
- **Media Fixed Function**: In addition, specialized fixed function hardware accelerates video codec and frame processing algorithms for fundamentally higher performance at lower power than the EUs or CPUs.



This combination means Intel GPUs have an ideal combination of features for video processing applications:

- Hardware accelerated decode and encode for many popular codecs, now including HEVC
- Direct access to hardware components of these codecs such as motion estimation is also available for custom algorithm development
- Hardware acceleration of many frequently used image processing algorithms (i.e. resize)
- Create custom pipelines mixing these already implemented components with your own algorithms
- High end GPUs now offer >1 TFLOPS of EU performance for custom filters, computer vision, encode quality improvements, etc.
- Codec operations which would require TFLOPS of CPU capability can be run simultaneously with EU and CPU tasks
- Low power, high density, low TCO



EU capabilities (here measured by peak TFLOPS) are rapidly increasing with each hardware generation

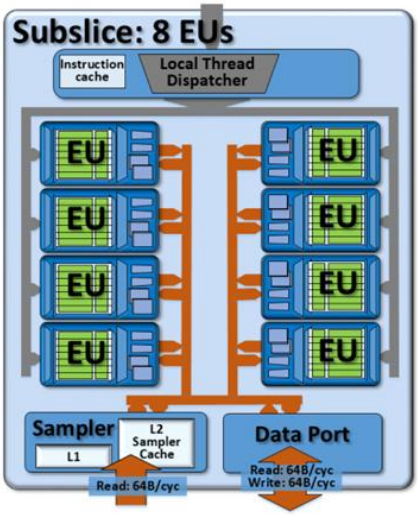
This is the lineup of codecs and filters available in Intel® Media SDK for Skylake processors. If your application makes use of one of these codecs or filters, performance/power is quite likely to be better if combined with GPU hardware acceleration than running only on CPUs.

HW Driver Codecs	AVC/H264, MPEG2, MJPEG HEVC/H265 (available in community/free editions)
Plugin Codecs	HEVC SW and GAcc/Hybrid screen capture (decode)
Filters	Resize/crop, Color conversion, deinterlace, Composition/ blend, FRC, Telecine/Interlace reversal,…

Intel® Media SDK capabilities summary for Skylake architecture processors

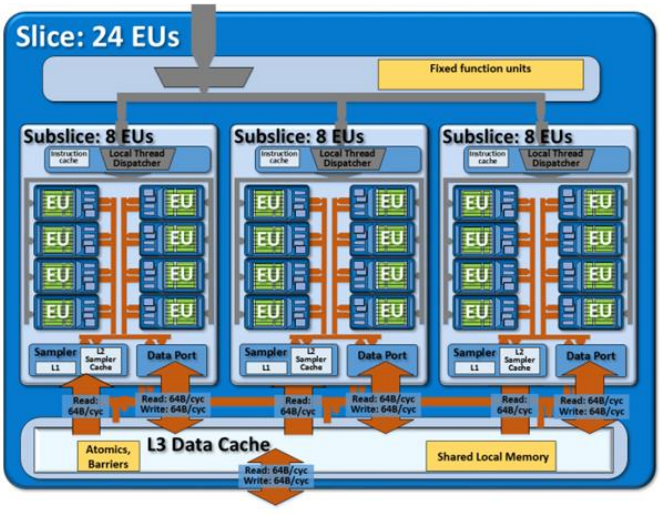
"The Compute Architecture of Intel Processor Graphics" series covers Intel Gen Graphics GPU architecture, and is a great reference for overall GPU architecture documentation. This guide adds additional information for media domain scenarios.

The basic unit of the GPU architecture is the subslice. This contains EUs and memory components. They are assembled into slices.



Gen9 Subslice components

Subslices are assembled into slices.



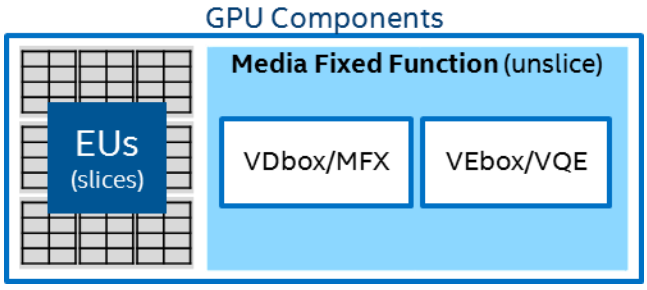
Gen9 Slice components

Execution Units are general purpose compute engines ideal for video processing uses. They are often used by encode for parts of the algorithm not run in fixed function like rate control and mode decisions.

The samplers are also highly important to media tasks. They are used by resize and motion estimation.

In addition to EU slices, there is an "unslice" with additional hardware engines individually schedulable for media tasks:

1. VDBox (also known as MFX) for codec operations
2. VEBox (also known as VQE) provides hardware acceleration for video enhancement/frame processing operations.



VDbox/MFX contains:

- Bitstream decoder (BSD).
- ENC (intra prediction, motion estimation)
- PAK (quantization, entropy coding, pixel reconstruction, motion compensation)

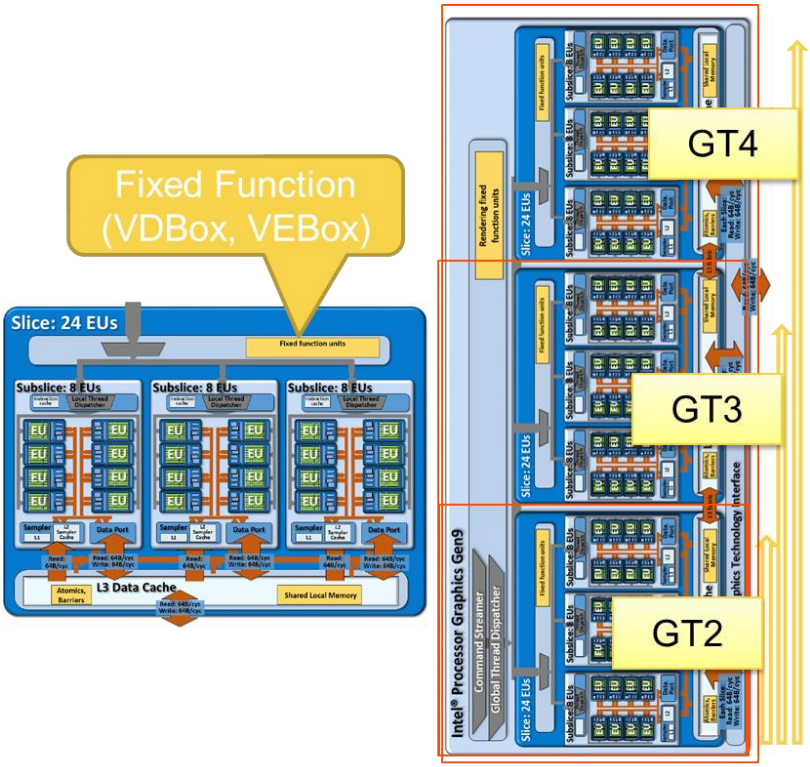
6th Generation Core / Skylake architecture introduces a new low power encode path entirely in VDBox called VDenc. However, it is lower quality.

VEbox/VQE contains:

- Denoise
- Advanced Deinterlace (ADI)
- Local Adaptive Contrast Enhancement (LACE)
- Camera processing features (skin tone enhancement, etc.)

To cover a wide range of performance, cost, and power requirements Intel GPUs have a scalable architecture. For 6th Generation Core/Gen9, the options start with 1 slice, sometimes referred to as a GT2 configuration. Each slice has 24 EUs. Adding a slice brings the GPU to the next level, GT3, with 48 EUs. For high end we also have a 3 slice configuration bringing the total to 72 EUs.

Each level adds more capability for Intel® Media SDK and Intel® SDK for OpenCL™ Applications developers to use.



Graphics Technology Highlights

- Slices contain execution Units (EUs) -- general purpose processing engines well suited to media tasks, shaders, and other memory hierarchy
- An "unslice" region contains VDbox/MFX for codec operations and VEbox/VQE for frame operations
- GT levels add more fixed function as well – for example GT2 has 1 VDbox/1 VEbox, GT4 has 2x.
- eDRAM adds cache, increases bandwidth

Naming conventions:

	adds	Other names	Summary
Intel® HD Graphics		GT2 "4+2"	Good

Intel® Iris™ Graphics	+slices +eDRAM	GT3 “2+3e”	Better
Intel® Iris™ Pro Graphics	+slices +eDRAM	GT3e,GT4e “4+4e”	Best

You can think of Intel’s GPU naming conventions as “good”, “better”, and “best”. This gives some structure to the multiple layers of names you might hear.

- “Good”=Intel HD Graphics. For a 4 CPU processor it might also be called a “4+2” configuration: 4 CPUs with GT2. Example: i7-6700K.
- “Better”=Intel Iris Graphics. Two processors with GT3 GPU + edram would be a “2+3e”. Example: i7-6567U
- “Best”=Intel Iris Pro Graphics. This is the best of everything: most slices, highest peak frequency, eDRAM. Example: i7-6770HQ.

With more hardware, each level represents a leap in capability.

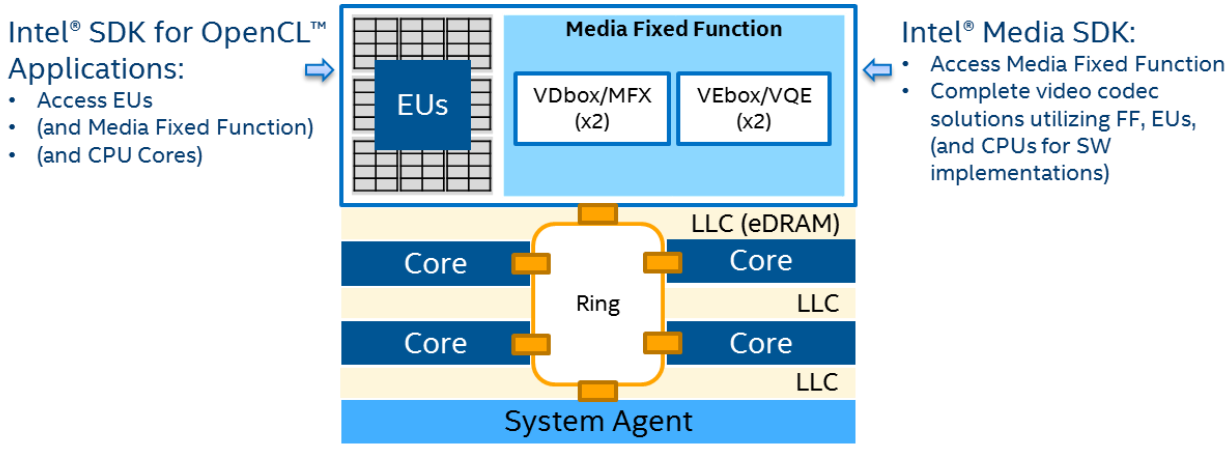
Within each level (for example, when comparing Intel® Iris™ Pro Graphics options) the main specification determining media performance capability is GPU peak frequency.

There is a lot of great information at ark.intel.com. “Processor Graphics” will tell you about the GPU capabilities available on the processor. But if you just want to know if the processor contains a Gen graphics GPU you can filter on Intel® Quick Sync video. This refers to the fixed function video codec capabilities but if they are available the rest of Gen graphics goodness is there too for Media SDK and OpenCL applications to access.

4 Developer Pathways to GPU Capabilities

There are two main ways that developers can access the amazing performance capabilities of Gen graphics GPUs:

1. **Intel® SDK for OpenCL™ applications**: provides low level access to EUs and memory hierarchy, with an increasing set of extensions to provide access to media fixed function capabilities.
2. **Intel® Media SDK** provides higher level access to complete codec implementations through an optimized asynchronous framework.



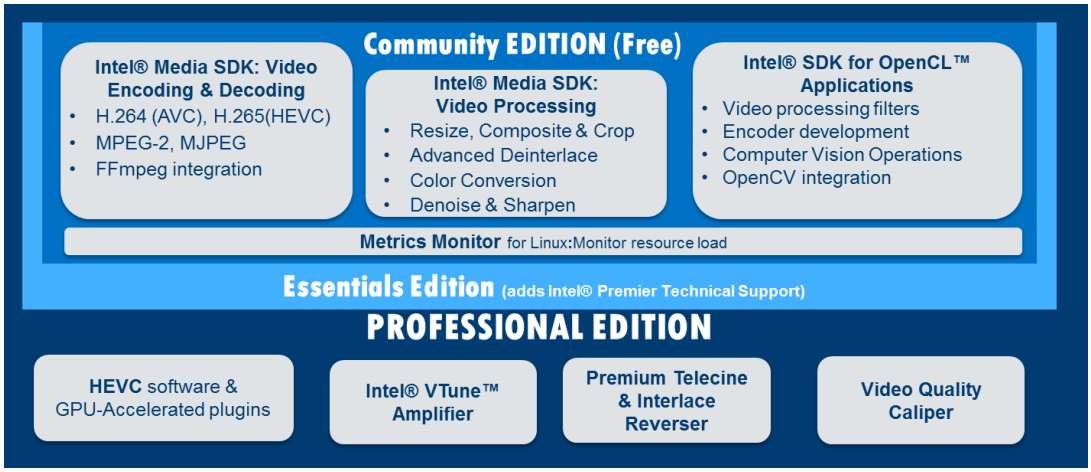
The Intel® Media Software Development Kit (Intel® Media SDK) abstracts the complex task of developing codec applications to use Intel GPUs, especially fixed function capabilities. It provides a high level view of video elementary stream processing operations. Since this enables highly optimized access to specialized hardware, it can be very fast and efficient. The high-level design also means that very good performance can be maintained as new generations of Intel® architectures emerge with few code changes. A software-only implementation is available with the same interface to provide compatibility with the widest range of processors. In this way the Intel Media SDK provides a portable and high-performance solution for the most demanding video processing tasks.

For operations not covered by Intel® Media SDK, Intel® SDK for OpenCL™ Applications provides a way to write custom algorithms. Traditionally the focus was low level programmability for EUs. An increasing amount of fixed function capabilities are also available via Intel extensions to the Khronos OpenCL standard.

Used together, these two SDKs provide developers with the opportunity to create fully customizable GPU pipelines. They can extend each other's capabilities in many ways, allowing developers to differentiate and innovate:

- OpenCL™ custom frame processing for transcode pipelines using Intel® Media SDK encode/decode.
- OpenCL™ and other computer vision algorithms receive input frames via with Intel® Media SDK hardware accelerated decode, and stream out using hardware accelerated encode.
- OpenCL™ motion analysis and/or frame quality analysis for custom encoder bitrate control

Intel® Media Server Studio combines multiple components to provide a complete package for media developers.



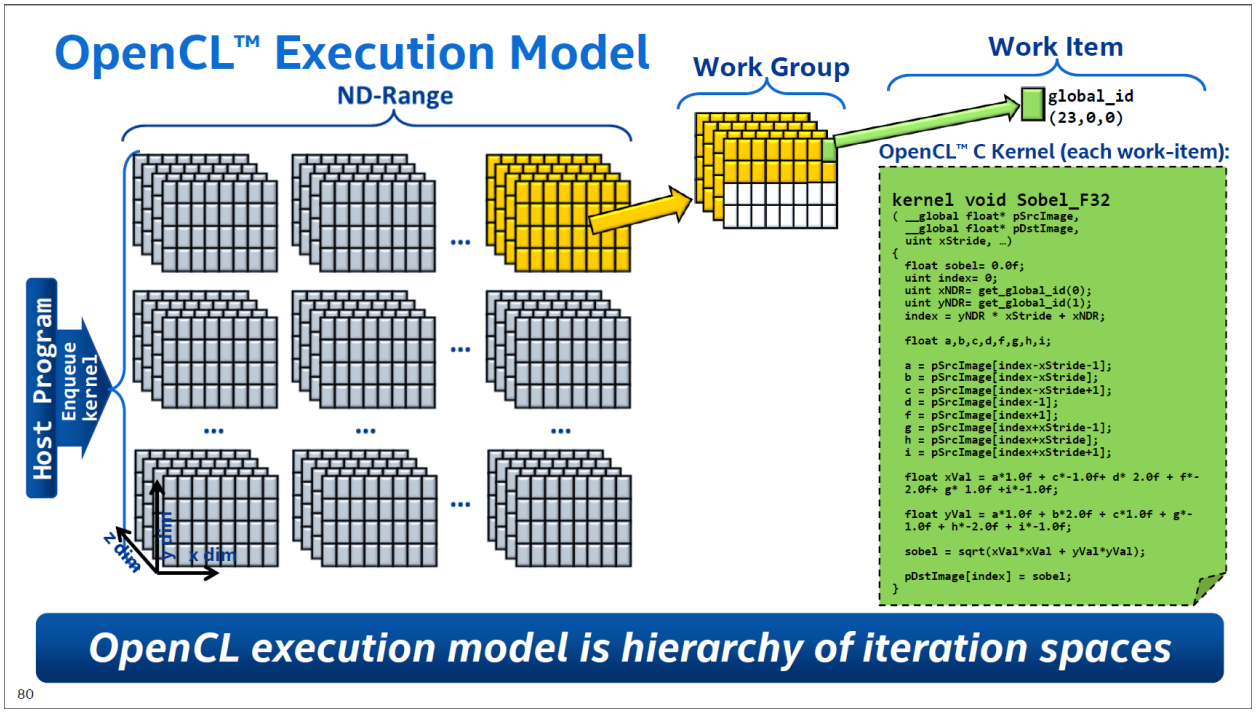
In summary, Intel® Media SDK and Intel® SDK for OpenCL™ Applications are the recommended pathways to access the capabilities of Gen Graphics GPUs. There are several ways to get these tools:

- Linux and Windows Server: Intel® Media Server Studio
- Windows client:
  - Intel® Media SDK standalone
  - Intel® SDK for OpenCL® applications standalone

4.1 Intel® SDK for OpenCL™ Applications Architecture overview (media focus)

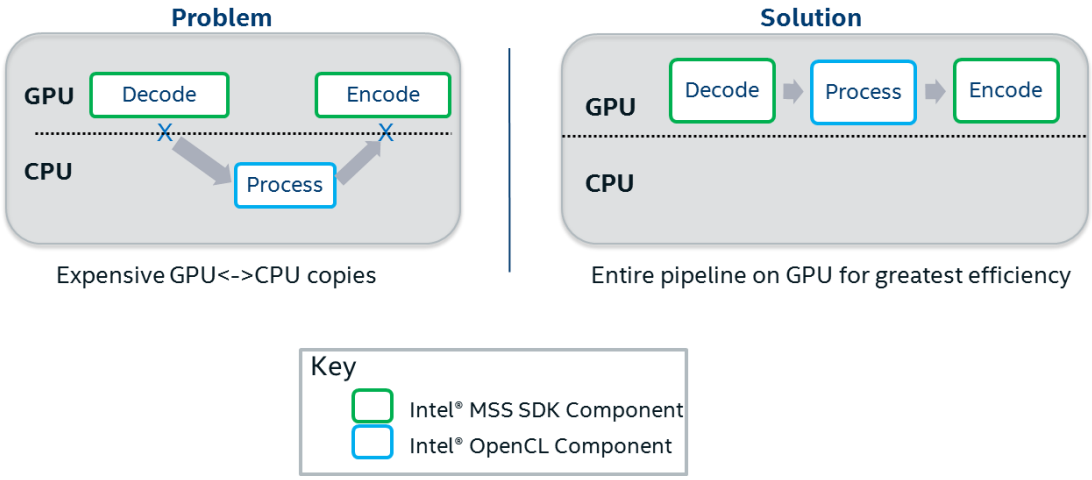
Many great references for OpenCL™ exist. What this section intends to add is a quick overview of use for media processing.

Media developers can write code with Intel® SDK for OpenCL™ Applications. In addition, an increasing number of algorithms in popular frameworks like OpenCV are accelerated with OpenCL.



OpenCL, an industry standard, provides an excellent framework for heterogeneous application development. It is based on executing kernels. These can be considered like the inner loop for an algorithm doing similar work across a range of data. Data can be arranged as 1D, 2D, or 3D (abstracted as ND). The kernel specifies the smallest unit of work, or "work item", where multiple concurrent threads are scheduled in "work groups" (the collection of work items executing on a single compute unit) across the many compute units available. These execute the kernel across the full ND range specified by the OpenCL program.

For media applications the work is often 2D. OpenCL is well suited for custom filters, quality metric feedback to bitrate control, and many other media tasks. When used with Media SDK, OpenCL can improve performance by keeping work on the GPU. This avoids extra CPU/GPU copies and synchronization.



Using Media SDK and OpenCL together to keep a video processing pipeline together on the GPU

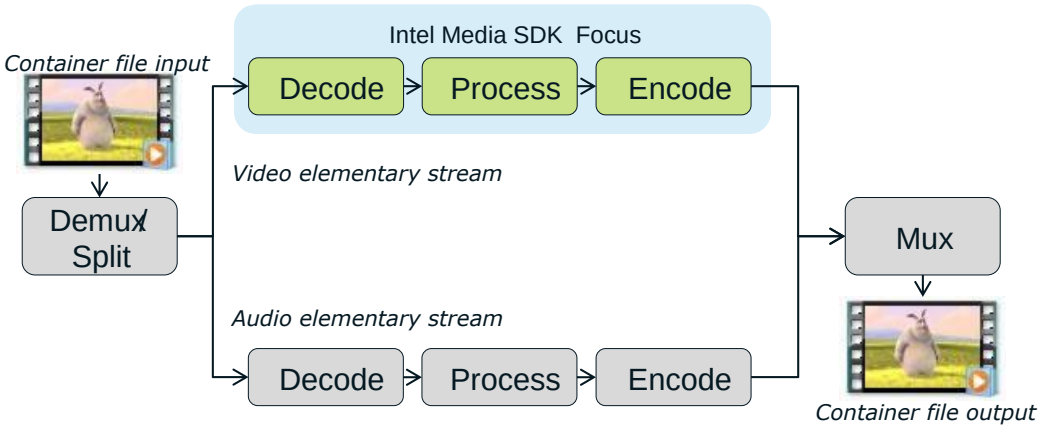
Traditional OpenCL kernels primarily execute on the EUs or CPUs. There are also builtin kernels available to access the fixed function capabilities of Intel hardware:

- https://software.intel.com/en-us/articles/video-motion-estimation-using-opencl
- https://software.intel.com/en-us/articles/advanced-video-motion-estimation-tutorial

4.2 Intel® Media SDK Application Overview

Media developers can write to the Intel® Media SDK API directly. However, quite often development will start in an established full featured framework like FFmpeg.

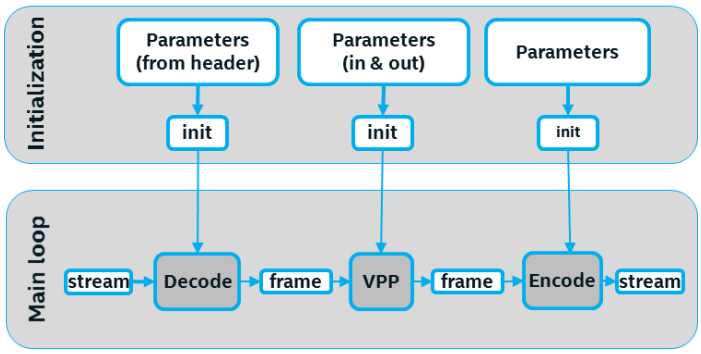
One of the main differences to keep in mind when developing directly to the Intel® Media SDK API is that it focuses on video elementary streams. This is where hardware acceleration is needed most. As media applications usually involve other components like containers and network protocols these must be integrated from elsewhere.



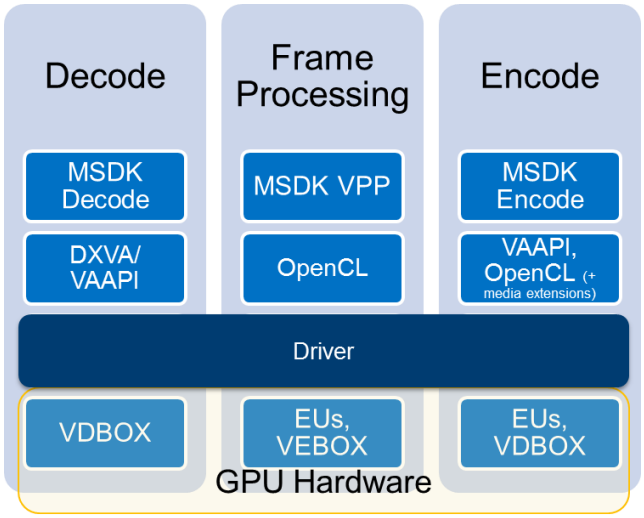
Some things to keep in mind as you start to use Media SDK:

- You're working at a high level on top of a full featured asynchronous framework, not writing directly to the hardware as you would with CPU code or OpenCL on the EUs.
- There are three basic operations: decode, video processing (VPP), and encode.
- Applications control behavior by providing parameters for these 3 basic pipeline stages.
- After initialization all the app needs to worry about is guiding data pointers through the pipeline.
- It's an accelerator model – you're scheduling work, not directly implementing it.

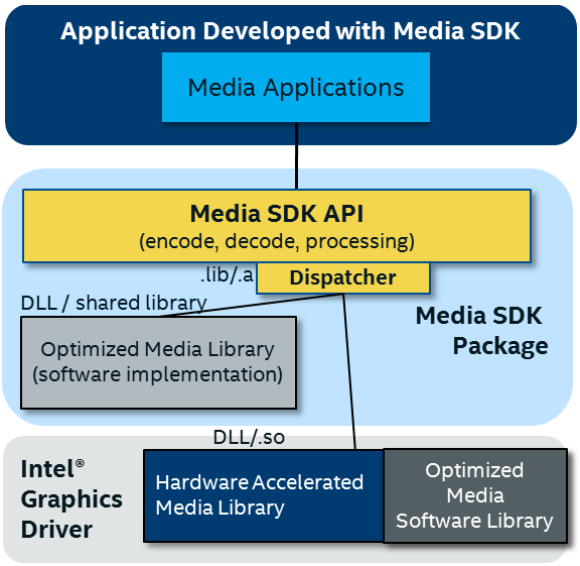
The library (and usually driver) take care of all of the details of running optimally for you. Usually all you have to do is remove barriers to Intel® Media SDK performance in your application code. This method does not require in-depth optimizations convoluting code readability like lower level approaches. A big advantage of working with a high level framework like this is that your code is "future proofed." As you move to new architectures you don't need to re-architect. Intel takes care of the optimizations for you so you can get as close-to-peak performance with the same application code across many hardware generations.



Intel® Media SDK and Intel® SDK for OpenCL™ Applications take care of most of the details of getting work to the GPU. The main remaining development hurdle for utilizing heterogeneous capabilities is that the component stack is more complex. A GPU driver is an intermediary to accelerate work on GPU hardware. For media there are additional graphics stack components as well.

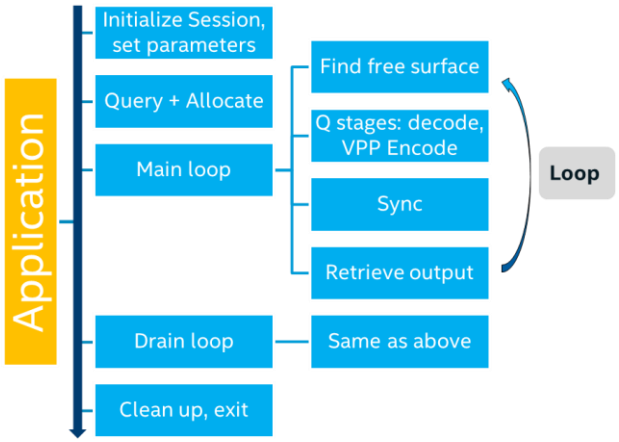


As with Intel® SDK for OpenCL™ Applications, the heterogeneous aspects of the programming model are handled by the application at runtime. Media SDK work can target CPUs or GPU.



Also, as with OpenCL code, there is a standard "boilerplate" of steps for each application.

- Set up session(s), set parameters for session stages
- Query number of frame surfaces required and allocate them
- Main loop (steps are covered in more detail in sections below)
- A drain loop is required to retrieve the final buffered frames
- cleanup



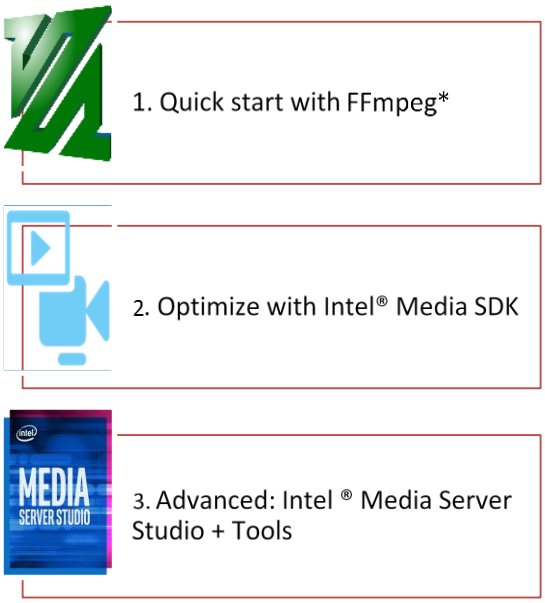
This is not like standard CPU implementations, with code that may potentially need to be optimized and obfuscated. These steps initialize Media SDK's optimized asynchronous framework and guide surfaces through pipelines, but the main work is done by other stack layers. This lets you focus on your algorithm development, not maintaining multiple pathways to cover hardware generations separately.

Once implemented, applications are automatically optimized for each hardware generation. There is just a short checklist to make it possible for Media Applications to avoid potential bottlenecks.

An ideal Application should	
Use video memory	☑
Avoid CPU<->GPU raw frame copies	☑
Run asynchronously	☑
Minimize waits for non-GPU tasks	☑

4.3 FFMpeg Integration

Getting started with Intel® Media SDK doesn't require a big investment. A possible strategy for developing media applications is to start with a full featured framework like FFmpeg*, then optimize where needed.

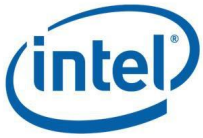


Hardware acceleration can be added to FFmpeg with a simple compile step. For applications written to use FFmpeg command line or libav* APIs they can then be hardware enabled by changing the codec name – for example from libx264 to h264_qsv.

The FFmpeg approach trades performance for ease of use. This means additional performance gains are possible above what you will see from simply recompiling the Intel® Media SDK wrappers in FFMpeg with additional optimization efforts. Future documentation will have more on this topic.

This article describing how to create hybrid applications with Intel® Media SDK pipelines fed and FFmpeg APIs is dated, but the concepts are still useful:

- https://software.intel.com/en-us/articles/integrating-intel-media-sdk-with-ffmpeg-for-muxdemuxing-and-audio-encodedecode-usages



Getting started is easy…

1. Download ffmpeg >= 2.8
2. Configure with –enable-libmfx; make; make install
3. Use HW accelerated codecs

>***ffmpeg*** *-I INPUT -c:v h264_qsv -preset:v faster out.qsv.mp4*

After compiling FFmpeg with libmfx support you should see h264_qsv,mpeg2_qsv, and hevc_qsv in the codecs list.

Parameter similarities/differences:

	x264	**h264_qsv**
Presets	Similar names, each codec has own perf/quality tradeoffs	
GOP structure	Same	
BRC	CRF (default), VBR/CBR	LA VBR (default), CBR

The *_qsv codecs have similar preset names for quality/performance tradeoffs as other commonly used software codecs in FFmpeg such as libx264. Please keep in mind that the Media SDK codecs are very different implementations and the presets are likely to behave differently.

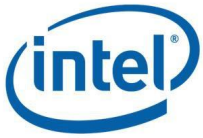
Many of the most commonly used settings such as for GOP structure are the same across codecs. However, as with other codecs in FFmpeg, beyond the basic settings common to most video codecs there are no guarantees that there will be an equivalent parameter when moving between codec implementations.

One of the biggest differences between the FFmpeg default codecs and the Media SDK wrappers is default bitrate control. The libx264 and libx265 codecs use constant rate factor (CRF) BRC. The Media SDK codecs in FFmpeg use lookahead VBR unless otherwise specified. CRF and VBR have very different

goals. VBR varies quality to maintain a bitrate, CRF varies bitrate to maintain a specified quality level. The easiest way to do an apples-to-apples comparison is by specifying CBR or VBR for the SW codecs.

More info on Quick sync video in FFmpeg:

- http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/cloud-computing-quicksync-video-ffmpeg-white-paper.pdf
- http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/quicksync-video-ffmpeg-install-valid.pdf



5 Installation

Unlocking the full capabilities of the Intel® Media SDK and Intel® SDK for OpenCL™ Applications requires

- Intel processors with integrated Gen graphics GPUs
- BIOS and motherboard enablement of integrated graphics
- Graphics driver components (user mode and kernel mode)
- (For Linux) graphics stack updates
- Additional libraries

These must be available for applications to use hardware accelerated media features.

For more information please see the articles below.

Hardware options:
https://software.intel.com/en-us/articles/mss-supported-hw

Graphics drivers:
https://software.intel.com/en-us/articles/driver-support-matrix-for-media-sdk-and-opencl

5.1 Windows Client Installation

All components needed to run GPU accelerated applications are distributed with the Windows graphics driver. Please make sure to stay updated to the latest available driver from downloadcenter.intel.com.

Note: some OEMs require their own validated driver versions. In this case please work with the driver update procedure for your OEM and platform.

Windows standalone Intel® Media SDK and Intel® SDK for OpenCL™ applications packages contain components for application developers, but in general these components are not required by end users who only want to run applications.

5.2 Linux Installation

The Linux installation model is significantly different from Windows client. Driver, library, and developer components are distributed together. It is recommended that all users (developers and end users) install all components.

Please see the Getting Started Guide for Linux installation details:
https://software.intel.com/sites/default/files/media_server_studio_getting_started_guide.pdf

This white paper has more information on troubleshooting your installation:
http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/quicksync-video-ffmpeg-install-valid.pdf

5.3 Windows Server Installation

The Windows Server install model combines components of Linux and Windows client.

A snapshot of Intel® Media SDK and Intel® SDK for OpenCL™ applications components from Windows client is validated with a specific driver version. Driver and SDK components are shipped together. Install and full package are validated for Intel® Xeon® processor use.

While it may be possible to upgrade to newer SDK or driver components, these hybrid client/server configurations are not validated.

5.4 Verifying Successful Installation

Windows:

- Installer does pre and post checks to ensure install success
- Check for an Intel graphics adapter in device manager, and ensure that the driver is the latest one available
- Check the contents of the %INTELMEDIASDKROOT% directory
- Run the mediasdk_sys_analyzer from %INTELMEDIASDKROOT%\tools\mediasdk_sys_analyzer, which should show Media SDK API availability and an Intel graphics device

Linux:

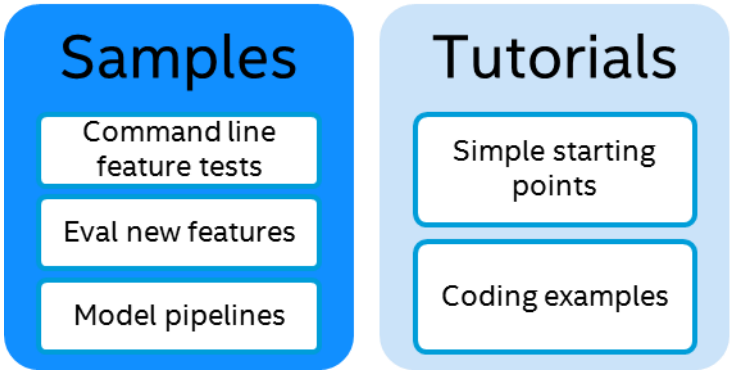
- Install is much more minimal than Windows, manual checking is required
- Check /opt/intel for mediasdk, opencl, and common folders
- Run vainfo
- please refer to the “Verifying correct installation” section in the Getting Started Guide

All: test with FFmpeg and/or see the section on Evaluating Intel® Media SDK via Samples and Tutorials.

6 Evaluating Intel® Media SDK via Samples and Tutorials

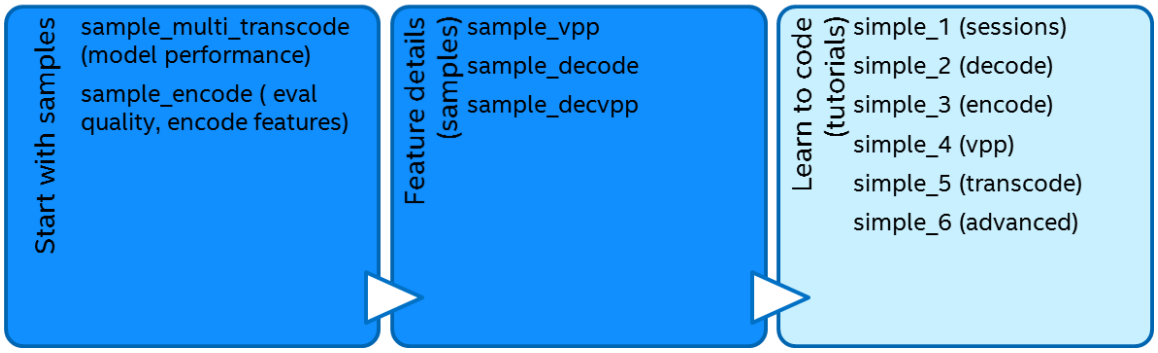
6.1 Sample/Tutorial Overview

A rich set of samples and tutorials are available to help understand Intel® Media SDK's capabilities and how to get started with your own implementations.



Samples are recommended for use as command line utilities to explore Intel® Media SDK capabilities. The **sample_multi_transcode** tool is especially valuable as a way to model the effects of parameters and pipeline architecture on transcode performance. **Sample_encode** is very useful for quality comparisons, has a wider variety of encode features, and is an easier base to add new parameters than the more complex transcode tool.

In addition to the samples, the tutorials provide simple starting points.

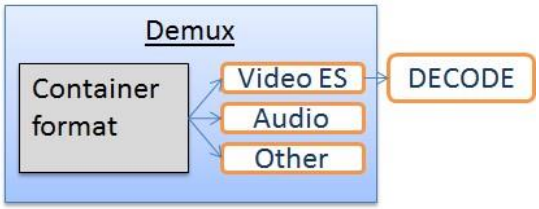


Here is a suggested order for evaluating Intel hardware accelerated codec capabilities:

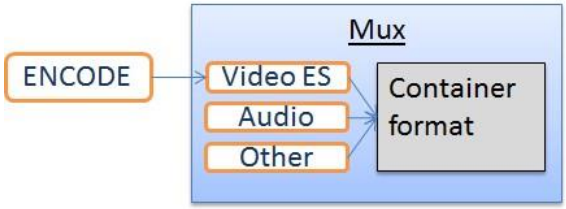
1. If you're already using FFmpeg, try out the simple recompile to access h264_qsv, hevc_qsv, and mpeg2_qsv.
2. **Evaluate peak "native" performance with sample_multi_transcode**. Transcode is a better representation of hardware performance capabilities than encode or decode separately. Unlike CPU implementations where transcode time is roughly decode+frame proc+encode, benchmarking these stages separately distorts timings with significant disk and memory I/O overhead for raw frames. The sample_multi_transcode tool is a great place to start understanding how fast your transcode applications could be.
3. **Evaluate quality with sample_encode**. This example is designed for quality comparisons, not performance comparisons (see above). It reads uncompressed inputs as expected for standard quality analysis. It also has a wider range of encode features pre-enabled than sample_multi_transcode, and is easier to extend.
4. **Use sample_decode to test decode performance**. Please note: YUV output is expensive and not optimized. Run without output for hardware decode times that will better represent decode work in pipelines that do not need to write each frame to disk.
5. **Test VPP features with sample_vpp**. See the VPP section for more info on command lines.
6. **Write your own code**. For many cases the existing FFMpeg integration can provide a quick path to a working application (while not quite as fast as native code). However, for non-FFmpeg scenarios or when additional performance is necessary, use the tutorials to learn how to code to the Intel® Media SDK API.

6.2 Working with containers

Most source content is not in an elementary stream format. Rather, it is a combination of both video and audio "muxed" together into a single file via a container format. It must be demuxed to a video elementary stream before the Intel Media SDK decode can begin to work.



Many video players read only container formats and do not directly support the H.264 or MPEG-2 elementary streams output by Intel Media SDK encode. For these players, an additional muxing stage to a container format is required.



6.3 Creating Elementary Streams from Existing Content

The Intel® Media SDK decoder and encoder work with elementary streams only. The Media SDK samples and tutorials require an additional demuxing step as described above.

Some sources for test content:

- Big Buck Bunny: https://peach.blender.org/
- Tears of Steel: https://mango.blender.org/
- Xiph Derf collection: https://media.xiph.org/video/derf/

The Derf collection is in uncompressed YUV4MPEG format (.y4m). This has minimal metadata for uncompressed frames. You can prepare it for transcode or decode use by encoding to a supported elementary stream format with ffmpeg, as below:

```
ffmpeg –i in.y4m out.h264
```

or for encode use by dumping to raw YV12/I420 YUV:

```
ffmpeg –i in.y4m out.yuv
```

It can be helpful to check what types of streams have been muxed together with a tool compatible with a wide variety of containers, such as MediaInfo. Here is MediaInfo's view of the streams in the Big Buck Bunny trailer (trailer_480p.mov):

In this case the MPEG-4 container file holds 1 AVC/H.264 video stream and 1 audio stream.

You can also use ffprobe to examine container file contents

```
ffprobe –i in.mp4
```

FFmpeg is an ideal demuxing tool:

```
ffmpeg -i in.mp4 -vcodec copy -bsf h264_mp4toannexb -an  out.h264
```

The "-vcodec copy" parameter above ensures that video packets are passed to output and not re-encoded. Audio is removed with "-an".

Note that for h.264 elementary streams it is necessary to enforce "Annex B" format. There are many possible formats for h.264 but this is the only one understood by Intel® Media SDK.

6.4 Creating Container Files from Elementary Streams

This step is necessary to view video elementary streams in most players.

```
ffmpeg -i in.h264 –vcodec copy out.mp4
```

The command line above reads in an elementary stream and wraps the packets in an mpeg4 container without re-encoding.

6.5 Playing elementary streams directly

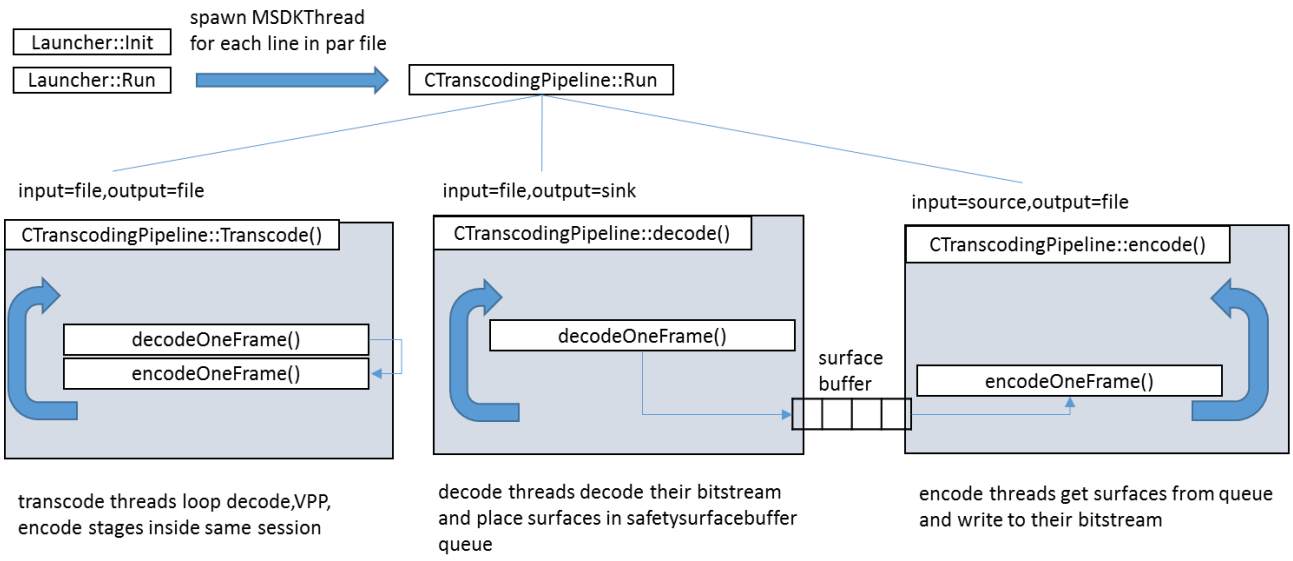
There are a few players which are able to view elementary streams outside of a container format. The ffplay utility in FFmpeg is ideal. Creation of this tool can be enabled when FFMpeg is compiled if the Simple DirectMedia Layer (SDL) libraries are installed.

```
ffplay out.h264
```

The Media SDK decode sample can also be used as an elementary stream player.

6.6 Evaluating performance with sample_multi_transcode

The sample_multi_transcode example is provided as a command line utility to allow quick evaluation of Intel QuickSync capabilities. It allows testing of a wide range of video processing capabilities.



The main program reads the command line or par (parameter) file. Each line in the par file specifies a thread. Threads come in three flavors:

1. **Transcode**: decode and encode are in the same thread, along with any other frame processing. These are self-contained operations which can be implemented within a single session. Input is a bitstream, output is a transcoded bitstream.
2. **Decode**: decode (and sometimes VPP) only. Input is a bitstream, output is a raw frame which is placed in a surface buffer, from decode perspective called the "sink".
3. **Encode**: encode (if specified, preceded by frame processing/VPP operations). Waits for inputs from the surface buffer, from encode perspective called the "source".

There are several ways the video pipeline stages can be arranged. These can all be modeled with sample_multi_transcode.

Name	Description
1:1	Simplest case: 1 input, one output. Except for cases like framerate conversion and deinterlacing, one frame is encoded for each frame decoded.
N:N	Several 1:1 pipelines running simultaneously and independently. Decoded frames are not available to other pipelines, and no synchronization between pipelines is required.
1:N	One decode, multiple encodes. For example, when a single input needs to be encoded to multiple formats.
N:1	Multiple decodes, one encode. Usually this involves a composition stage, where multiple inputs are overlayed either as picture-in-picture or a video wall.

The sample_multi_transcode tool can be used from command line or, for modeling more complex scenarios, with parameter(par) files. You can see the parameters available with

```
$ sample_multi_transcode.exe -?
```

1:1 transcodes

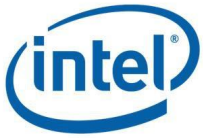
This is the simplest case to start testing with sample_multi_transcode. The command line below is a single decode, and encode with no frame processing.

Command line example:

```
sample_multi_transcode.exe -i::h264 in.h264 -o::h264 out.h264 -hw -b 2000
```

The same in Par file syntax:

```
$ cat test.par
-i::h264 in.h264 -o::h264 out.h264 -hw -b 2000
```



```
$ sample_multi_transcode.exe -par test.par
```

N:N transcodes
Multiple concurrent transcodes can also be specified from a par file. In this scenario there is a separate decode for each pipeline.

```
$ cat test.par
-i::h264 in.h264 -o::h264 out0.h264 -hw -b 2000 -join
-i::h264 in.h264 -o::h264 out1.h264 -hw -b 4000 -join
-i::h264 in.h264 -o::h264 out2.h264 -hw -b 6000 -join
-i::h264 in.h264 -o::h264 out3.h264 -hw -b 8000 -join

$ sample_multi_transcode.exe -par test.par
```

With the par file above the same input is transcoded to multiple outputs sweeping a bitrate range. This starts multiple sessions, which are joined. The joining step has minimal performance impact for the hardware implementation, but joining sessions can be more memory efficient.

Since the same file is decoded multiple times, an obvious optimization is to do only one decode and share with the encodes using a 1:N pipeline instead.

1:N transcodes
1:N is usually the fastest option in terms of total FPS output for the processor, as it minimizes overhead from decode and requires coordination of work on fewer surfaces.

The pipeline below has the same output as the N:N scenario above, but without the “extra” decodes.

```
$ cat test.par
-i::h264 in.h264 -o::sink -hw -b 2000 -join
-i::source -o::h264 out1.h264 -hw -b 4000 -join
-i::source -o::h264 out2.h264 -hw -b 6000 -join
-i::source -o::h264 out3.h264 -hw -b 8000 -join

$ sample_multi_transcode.exe -par test.par
```

Here the decoded frames are passed to a surface queue. (See the architecture diagram above.) The decode side is called the “sink” and the output side is the “source”. Frames are read from the source instead of waiting for separate decodes, then passed to encode stages with the parameters specified.

N:1 transcodes

Sample_multi_transcode supports composition with vpp_comp* parameters. For more information please see readme-multi-transcode.pdf in the sample_multi_transcode directory of the samples package. Watch for more updates on this topic in future versions of this guide.

Additional parameters to consider for performance:

```
-u (targetusage)  1=best quality/slowest, 7=best speed/fastest.
```

2-6 are most practical for general use. Quality gains from going from –u 2 to –u 1 may not offset additional processing time. Similarly, the additional speedup from –u 7 from –u 6 is minimal compared to the large quality drop, especially for h264.

```
-async (asynchronous depth)  The number of frames which can be processed
simultaneously without synchronization.
```

For h.264 best performance is usually in 1-4 range. Trend toward 1 as number of simultaneous transcodes increases, use 4 for lower #s of concurrent transcodes.

```
-l (# of slices)
```

slices increase parallelism and performance but tradeoff is quality/bitrate. More bits are required for slice headers, and motion search cannot cross slice boundaries.

6.7 Building the sample applications

For Windows, sample projects are created for Microsoft* Visual Studio* 2010. You will see a warning as part of the conversion with more recent versions of Microsoft Visual Studio. This is not a problem.

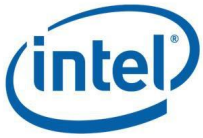
On Linux, sample projects built by running the perl script “build.pl” which runs cmake to do the compilation.

The Media SDK samples are distributed as separate downloadable packages. For Intel®

Media SDK, use this download page, for Intel® Media Server Studio, download from here.

All of the samples share some common design characteristics.

- Pipelines: the output from one stage provides the input to the next. There can be multiple pipelines (such as in the multi transcode sample), but, as with the underlying Intel Media SDK implementation, there is an assumption of a linear series of “media building block” operations.



- Utilities: the sample_common directory contains utility classes for file I/O, surface/buffer allocation, etc. These are used by all of the samples. Additional functionality added here will be available to all samples.
- Opaque memory: Opaque memory is enabled for transcode pipelines, simplifying surface memory management.

6.8 Intel Media SDK Tutorials

In addition to the samples, developers new to the Intel Media SDK are encouraged to use the Intel Media SDK tutorials to understand how the API works. This simplified introduction is divided into six distinct sections with increasing levels of complexity:

- Section 1: Introduces the Intel Media SDK session concept via a very simple sample.
- Section 2-4: Illustrates how to utilize the three core SDK components: Decode, Encode, and VPP.
- Section 5: Showcases transcode workloads, utilizing the components described in earlier sections.
- Section 6: Describes more advanced and compound usages of the SDK.

In addition to demonstrating how to implement the most common workloads, the tutorials also explain some key concepts to optimal performance via a step-by-step approach. They can be downloaded here: For Media Server Studio, and Media SDK for Client.

7 Initialization: Session, Queries, and Allocations

7.1 Intel® Media SDK application design fundamentals

The Intel® Media SDK is designed to represent media operations as easy-to-use highlevel building blocks. For example, decode takes a bit stream as input and produces surfaces as output. However, there are also some fundamental design characteristics to keep in mind to fully utilize the performance benefits of Intel Media SDK.

The list below outlines some basic architectural concepts for any Intel® Media SDK project. The program's entire architecture does not need to fit these principles, but the section working with Intel Media SDK should have these characteristics:

Session/pipeline-based: A session is created for a pipeline of steps. Surfaces may be shared internally between pipeline stages. Sessions use the dispatcher to map function calls to their DLL implementation.

Asynchronous: each stage can have multiple frames "in flight", meaning that they are in some stage of processing on the CPU or GPU. In general, Intel Media SDK functions do not work like a traditional function call with a single input and output. Frame surface synchronization status needs to be checked. Frames may be locked while a session is working on them.

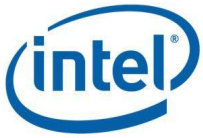
Based on the NV12 color format: Decode/encode and VPP operations use NV12 because this provides better performance than other formats such as YUV. While it is possible to include color conversion filters it is best if pipelines can be arranged to minimize conversion steps by doing as much consecutive work in NV12 as possible.

Designed to minimize copies: as with NV12 conversions, arrange pipeline steps to reuse surfaces in the same location instead of copying them between CPU and GPU.

7.2 Intel® Media SDK function groups

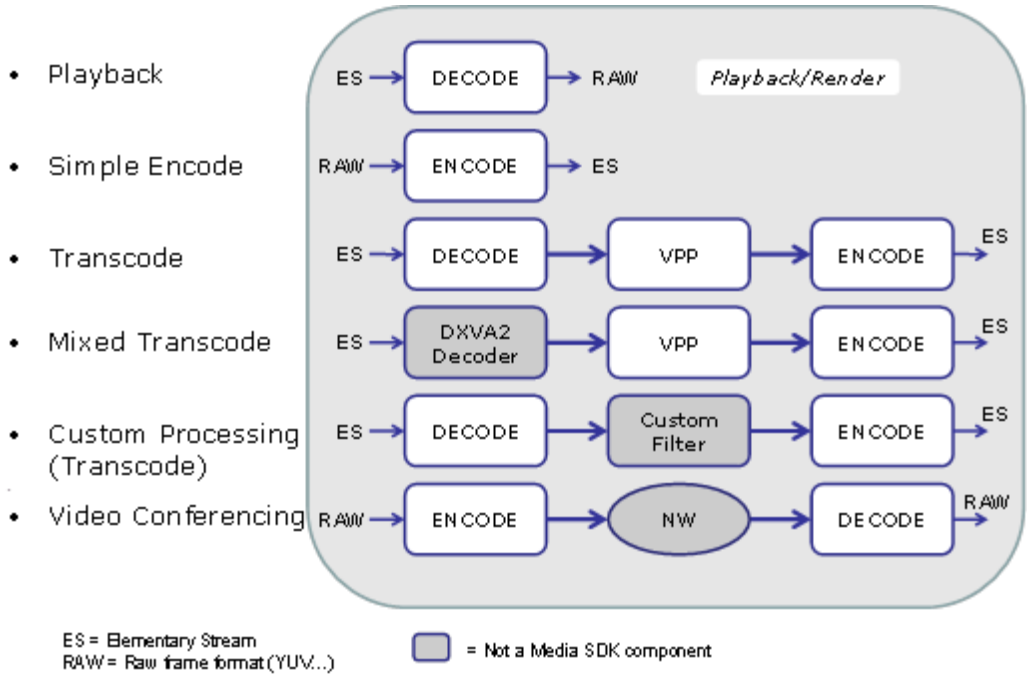
At its core, the Intel® Media SDK consists of five function groups accessed via the dispatcher. Syntax is identical in the software and hardware versions, though since they are different libraries, results can be different.

CORE	Auxiliary functions such as synchronization



DECODE	Decodes bitstreams into raw video frames
VPP	Processes raw video frames
ENCODE	Encodes raw video frames into bitstreams
USER	Performs user-defined algorithms as plugins

The Decode, VPP, Encode, and User function groups can be used as building blocks to provide the foundation for many different usage scenarios, such as:



While the manual is based on the C functions in mfxvideo.h, the samples use the C++ interface in mfxvideo++.h. In general the C++ functions are simply a wrapper around the corresponding C functions (see the appendix). The same function groups can be seen in both sets of functions.

7.3 The dispatcher, software implementation, and software fallback

The media dispatching library, or dispatcher, is the gateway to Intel® Media SDK functions. This is statically linked to all Intel Media SDK-enabled programs. The dispatcher is responsible for exposing the entry points for the encoding, decoding, and video preprocessing routines. The dispatcher is also responsible for detecting and loading the appropriate implementation library for the client machine.

If the machine has compatible hardware and the dispatcher can find a way to access it (implying an appropriate graphics driver has been installed), the API calls will use the hardware implementation.

If software mode is requested, or Intel® Media SDK is initialized in auto mode and no hardware implementation can be found, the software implementation will start.

Implementation	Description
MFX_IMPL_HARDWARE_ANY	Find the platform-specific implementation on any acceleration device
MFX_IMPL_HARDWARE(2-4)	Use the platform specific implementation of specific acceleration device
MFX_IMPL_SOFTWARE	Same interface as hardware implementation, but CPU only.
MFX_IMPL_AUTO	Dispatcher chooses hardware or software at runtime based on system capabilities.
MFX_IMPL_AUTO_ANY	Recommended to ensure processing on any hardware device with software fall back if needed.

Use the following flags to specify the OS infrastructure that hardware acceleration should be based on:

MFX_IMPL_VIA_D3D9	Hardware acceleration via the Microsoft* DirectX 9 infrastructure
MFX_IMPL_VIA_D3D11	Hardware acceleration via the Microsoft* DirectX 11 infrastructure
MFX_IMPL_VIA_VAAPI	Hardware acceleration goes through the Linux* VA API infrastructure.

MFX_IMPL_VIA_ANY	Hardware acceleration via any supported OS supported OS infrastructure

Please refer to the Intel® Media SDK reference manual and header files for details about additional implementation targets, primarily used to support systems with multiple GPUs.

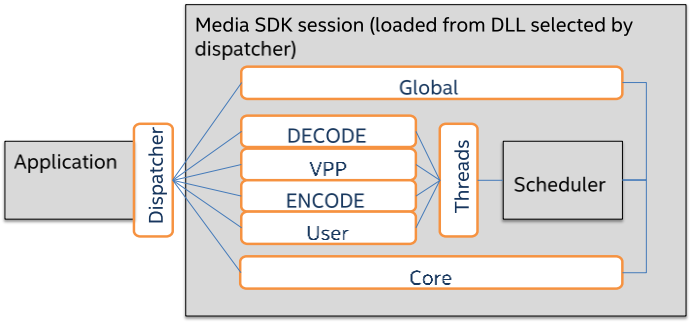
The dispatcher determines which pathway to use at session startup. If the hardware mode is chosen and the application cannot complete the operations requested by the parameters, Intel® Media SDK goes into "software fallback" mode, which can work with a wider variety of inputs. Operations should be checked for the warning MFX_WRN_PARTIAL_ACCELERATION, which indicates that this has occurred.

Results from the software and hardware implementations can be different, in more ways than just speed. While the software implementation is a reference and functional preview for the hardware accelerated version, in some cases different algorithms are used.

The Media SDK dispatcher is also available as source code in the "opensource/mfx_dispatch" which is part of the Media SDK installer package.

7.4 Under the hood of the dispatcher

The Intel® Media SDK dispatcher library provides the function interfaces. The dispatcher loads the appropriate library at runtime and sets up an SDK context called a "session" to the application. The session delivers the API's functionality and infrastructure.



The Dispatcher (libmfx.lib for Windows and libmfx.a for Linux*) must be statically linked to your application. It is responsible for locating and loading the correct library.

When an application initializes an Intel® Media SDK session, the dispatcher will load either the software library (MFX_IMPL_SOFTWARE) or the hardware library appropriate for the platform (MFX_IMPL_HARDWARE).

To enable a solution capable of running on systems with or without hardware acceleration, the application can instruct the SDK to automatically choose the library with MFX_IMPL_AUTO_ANY. This is the recommended usage.

Windows DLL search strategies:

Implementation Version	Search strategy
Hardware	"Common Files" locations for 32- and 64-bit DLLs specified in the registry by the graphics driver installer. DLL name includes codes for target platform. If a driver with the appropriate name is not found in the registry location used by the dispatcher, hardware session initialization will fail.
Software	Located by standard DLL search rules (i.e. system path). Intel Media SDK installer updates path with install target bin directory. If libmfxsw64.dll (libmfxsw32.dll for MFXInit run in 32 bit mode) cannot be found, the software session initialization will fail.

Problems loading the hardware library could be related to version (e.g., a higher API version is available for software than hardware, which can happen in beta releases or if the graphics driver is not updated with a new Intel Media SDK install). Hardware acceleration problems can also often be traced to the driver. With MFX_IMPL_AUTO_ANY the software fallback happens silently, with no error. In many cases this is the desired functionality, though logging the version loaded can be helpful. MFXQueryIMPL and MFXQueryVersion can be useful to provide diagnostic feedback.

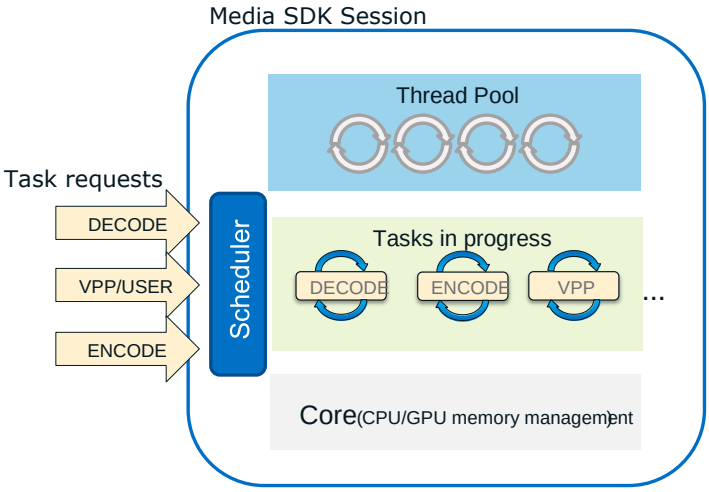
7.5 Intel® Media SDK sessions under the hood

Here are some fundamental ideas behind Intel® Media SDK sessions. Implementation details are subject to change, but there are several essential architectural components to be aware of:

Scheduler: Manages subtask dependencies and priorities as multiple requests are handled asynchronously. The scheduler is responsible for synchronization, and reports the status code back when the sync point is queried.

Thread pool: Started with session initialization to avoid thread creation overhead. The scheduler manages thread assignment to tasks.

Memory management core: Intended to enable the highest possible performance via CPU and GPU allocation management, copy minimization, fast copy operations, atomic lock/unlock operations, opaque memory, etc.



All of these components work together to maximize asynchronous throughput of media tasks. Since Intel Media SDK automatically creates several threads per session, this may interfere with other threading approaches, especially in software mode.

7.6 Creating Sessions

All applications must first create an Intel® Media SDK session to initialize the library and create a context for work to be done. Only one instance of the Decoder, Encoder, and VPP (preprocessing module) can run per session. Applications can create more than one session to accommodate multiple instances or pipelines. Software and hardware pipelines can exist simultaneously—this can be a way to balance GPU and CPU use for higher throughput.

To create a session, use the function:
mfxStatus sts = MFXInit(mfxIMPL impl, mfxVersion *ver, *session) Parameters:

Parameter	description

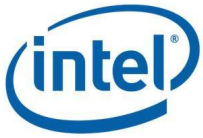
Impl(in)	Implementation requested. The most commonly used implementations are: • MFX_IMPL_AUTO_ANY • MFX_IMPL_HARDWARE_ANY • MFX_IMPL_SOFTWARE (for software HEVC plugin) Others are for special purposes
Ver(in)	API version requested (recommended: minimum API version for required functionality.)
Session (out)	Media SDK session created

Return status is either MFX_ERR_NONE (session is created and ready to use) or MFX_ERR_UNSUPPORTED (could not find the version/implementation requested). This status is very important to check, since most of the Intel Media SDK functions will not work without a valid session. Later return codes of MFX_ERR_INVALID_HANDLE may indicate an unusable session.

For most cases we recommend specifying a version instead of using the null version feature. The minimum version providing all functions used by the program is best, since this will allow the greatest portability. For example, even if API version 1.19 is available, it is probably not required if a session is only doing a simple encode—this can be accomplished by specifying version 1.0, which would enable portability even to machines with very old drivers.

7.7 Query version and implementation

The following program is a minimal Intel® Media SDK "hello world". It attempts to start an Intel Media SDK software and hardware session. Once the session is active, its version and implementation can be queried.



```
#include "mfxvideo.h"
#include <stdio.h>

int main()
{     mfxVersion SWversion = {0,1}, HWversion = {0,1}, version;     mfxSession
SWsession, HWsession;                                  mfxStatus sts;

    sts = MFXInit(MFX_IMPL_SOFTWARE, &SWversion, &SWsession);     if (MFX_ERR_NONE ==
sts)
    {
        MFXQueryVersion(SWsession,&version);
        printf("SW version:%d.%d  API level: %d.%d\n",
            SWversion.Major,SWversion.Minor,              version.Major,
version.Minor);
    }

    Sts = MFXInit(MFX_IMPL_HARDWARE, &HWversion, &HWsession);     if (MFX_ERR_NONE ==
sts)
    {
        MFXQueryVersion(HWsession,&version);
        printf("HW version:%d.%d  API Level: %d.%d\n",
            HWversion.Major,HWversion.Minor,             version.Major,
version.Minor);
    }

    MFXClose(SWsession);     MFXClose(HWSession);
}
```

Comparing the software and hardware API level can help determine if the graphics driver is out of sync and needs to be updated. Note: beta releases can offer a software "preview" of hardware capabilities, so the software API level may sometimes be higher.

Also note that there is a "mediasdk_system_analyzer" tool for Windows operating systems distributed with the SDK package. The tool analyzes the developer platform and reports back Media SDK related capabilities.

7.8 Join/clone/disjoin session

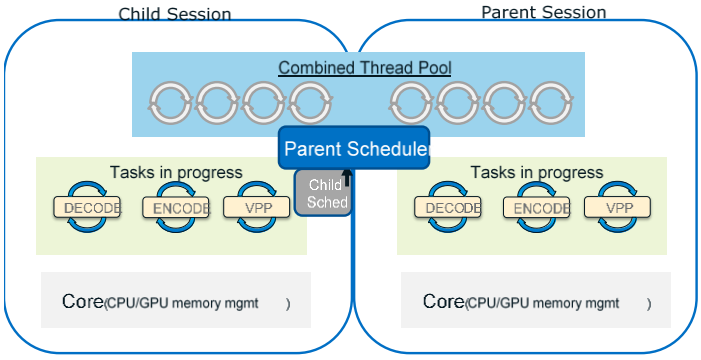
Intel® Media SDK provides functionality to handle multiple simultaneous sessions/pipelines. This is designed to facilitate accurate system resource sharing to boost the performance of advanced workloads such as transcodes of multiple inputs at the same time.

An Intel® Media SDK session holds the context of execution for a given task, and may contain only a single instance of **DECODE**, **VPP**, and **ENCODE**. Intel Media SDK's intrinsic parallelism, in terms of thread pool and scheduler, is well optimized for individual pipelines. For workloads with multiple simultaneous pipelines additional sessions are required. To avoid duplicating the resources needed for each session they can be "joined".

mfxCloneSession	After the initial session is created the application can create a child of the initial session. The application must complete all necessary pipeline initializations, including setting the allocator, initializing DECODE, VPP, and ENCODE using this new session. When created the cloned session has a separate task scheduler.
mfxJoinSession	Share the task scheduler (thread pool) between parent and child sessions. .
mfxDisjoinSession	After the batch conversion is complete, the application must disjoin all child sessions prior to closing the root session.
mfxClosesession	The application must also close the child session during deinitialization.

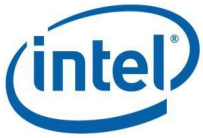
The performance impact with joined sessions is most noticeable when using the Intel Media SDK with software based sessions. This is because the CPU is prone to oversubscription. Joining sessions reduces the impact of this oversubscription. The benefits for hardware sessions are more subtle, but in some cases joining can reduce overall memory footprint.

With shared sessions, child session threads are placed under the control of the parent scheduler, and all child scheduler requests are forwarded. In addition to the performance benefits described above, this scheme enables coherent subtask dependency checking across the joined sessions.



A joined session, illustrating control forwarding from child scheduler.

The relationship between parent and child sessions can be fine-tuned with priority settings, but usually this is not important for performance.



7.9 Core Functions: Interacting with the asynchronous acceleration infrastructure

The Intel® Media SDK hides most of the complexity of efficiently pushing data through the session infrastructure, but awareness of the internal mechanisms can be helpful for developing applications.

- Surfaces need to be set up where the work will be done (CPU or GPU) to avoid extra copies.
- The session needs to buffer frames internally, as well as reorder them, to maximize performance.
- The session will estimate how many surfaces it needs. This happens before work starts so the application can query how many surfaces to allocate.

As work is happening, some frame surfaces will be locked. The program controlling the session will frequently need to find an unlocked frame surface as it interacts with the session.

These steps are shared for all function groups.

7.10 IOPattern: System, Video, and Opaque memory

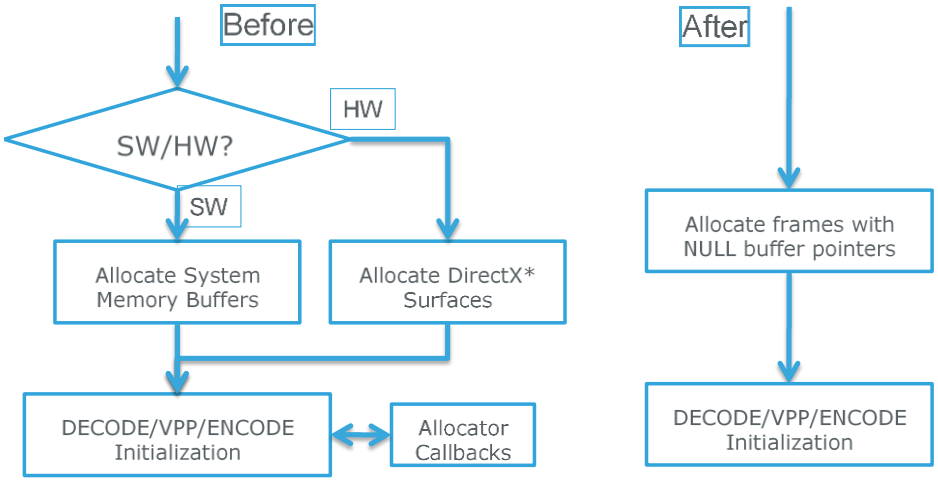
IOPattern specifies the kind of surface memory expected by Intel® Media SDK stages.

Decode requires an IOPATTERN_OUT setting, encode requires IOPATTERN_IN. VPP has both.

SYSTEM_MEMORY specifies that CPU memory will be used. This is best for the software implementation, but can simplify integration for applications using hardware acceleration.

VIDEO_MEMORY specifies that GPU memory will be used. This is best for the hardware implementation.

For Media SDK only transcode sessions, OPAQUE_MEMORY allows some simplification. Instead of requiring separate surface allocations to handle hardware and software implementation scenarios, the SDK handles surface allocation. However, this comes with the tradeoff that the surfaces will not be available to the application.



Code simplification with opaque surfaces

This flowchart applies to Linux* VAAPI as well.
For an example of opaque surface use, please see sample_multi_transcode.

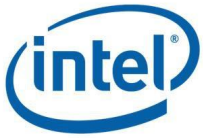
7.11 Surface reordering and locking

Intel® Media SDK reorders frames and builds reference lists to maximize performance. Extra surfaces are required to enable these internal optimizations. Since the SDK framework is operating concurrently with the main CPU program controlling the session, the session must make some surfaces unusable while it is working with them. The controlling program must search for unlocked surfaces when interacting with a session.

The lock mechanism works much like other lock implementations. Internally, the Intel Media SDK session increases a lock count when a surface is used and decreases the count when done. Applications should consider the frame off limits when locked. Also, other than checking the lock state, **applications should not use the SDK's lock field**. If additional locking oversight is required, such as for multi-threaded use, the application must add its own additional lock state management.

Pseudo code:

```
Allocate a pool of frame surfaces decode->DecodeHeader() decode->Init()
while (bitstream not finished)
{
    Locate an unlocked frame by checking lock state
```



```
    Fill the frame with encode input
    Send the frame to the SDK encoding function
    Sync
    Write any output bitstream
}
De-allocate frame surfaces
```

7.12 Surface Allocation with QueryIOSurf

All surfaces used should be allocated at session creation. While SDK operations are thread safe, allocations are not. **The SDK assumes all allocations will be done in a single threaded initialization step before launching other threads for the main execution loops**.

Each session needs enough surfaces to implement everything in its pipeline, as well as a few extra to accommodate asynchronous reordering and optimized data sharing between stages. To accomplish this, each pipeline stage (encode, decode, etc.) needs to call QueryIOSurf to determine how many surfaces will be needed for the specified parameters.

The application should query each stage to determine how many surfaces are necessary. The total number of surfaces to allocate is **the sum of all stages**.

The following code is from simple_2_decode_vmem, showing allocations for video memory.

The QueryIOSurf function is called after decode parameters have been set by decoding the input stream header to see how many surfaces are necessary.

```
    // Query number of required surfaces for decoder
    mfxFrameAllocRequest Request;
    sts = mfxDEC.QueryIOSurf(&mfxVideoParams, &Request);
```

Next step is to run the allocator. The application is responsible for actually creating all of the surfaces used, except for opaque memory. Please watch for more documentation on the allocator interface in future developer's guide versions.

```
    mfxU16 numSurfaces = Request.NumFrameSuggested;

    // Allocate surfaces for decoder
    mfxFrameAllocResponse mfxResponse;
    sts = mfxAllocator.Alloc(mfxAllocator.pthis, &Request, &mfxResponse);
    MSDK_CHECK_RESULT(sts, MFX_ERR_NONE, sts);
```

In the next step, the pointers used to guide surfaces through the main work loops are allocated. Note that the memory ID (.MemID) for the surface pointer comes from the allocator Alloc call above.

```
    // Allocate surface headers (mfxFrameSurface1) for decoder
    mfxFrameSurface1** pmfxSurfaces = new mfxFrameSurface1 *[numSurfaces];
    MSDK_CHECK_POINTER(pmfxSurfaces, MFX_ERR_MEMORY_ALLOC);
    for (int i = 0; i < numSurfaces; i++) {
        pmfxSurfaces[i] = new mfxFrameSurface1;
        memset(pmfxSurfaces[i], 0, sizeof(mfxFrameSurface1));
        memcpy(&(pmfxSurfaces[i]->Info), &(mfxVideoParams.mfx.FrameInfo),
sizeof(mfxFrameInfo));
        pmfxSurfaces[i]->Data.MemId = mfxResponse.mids[i];
    }
```

7.13 Finding unlocked surfaces

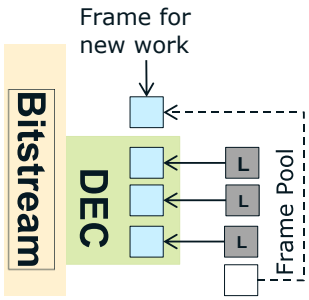
Since the application manages the surface pool, it must also be aware of which surfaces are currently in use by asynchronous operations. Locked surfaces cannot be used so a search must be done for an unused surface whenever a new one starts its way through the pipeline.

```
while (MFX_ERR_NONE <= sts ||       MFX_ERR_MORE_DATA == sts ||
       MFX_ERR_MORE_SURFACE == sts)
{
   // . . .
   else if (MFX_ERR_MORE_SURFACE == sts || MFX_ERR_NONE == sts)    {
      // find new working surface
      nIndex  =  GetFreeSurfaceIndex(m_pmfxSurfaces,  m_mfxResponse.NumFrameActual);
}
   sts = m_pmfxDEC->DecodeFrameAsync(&m_mfxBS,
                                     &(m_pmfxSurfaces[nIndex]),
                                     &pmfxOutSurface,
&syncp);    // . . .
}
```

The surface search can be straightforward, as below.

```
mfxU16 GetFreeSurfaceIndex(mfxFrameSurface1* pSurfacesPool, mfxU16 nPoolSize)
{
    if (pSurfacesPool)
    {
        for (mfxU16 i = 0; i < nPoolSize; i++)
        {
            if (0 == pSurfacesPool[i].Data.Locked)
            {
                return i;
            }
        }
    }

    return MSDK_INVALID_SURF_IDX;
}
```

The following diagram illustrates the buffer search process. Individual members of the frame/surface pool are managed externally. Those currently in use are flagged as locked, so finding an unlocked surface requires the application to search for one that is available.



L=locked

7.14 Using queries for parameter validation

The Encoder, Decoder, and VPP components validate the set of parameters supplied but depending on the operation the scope of the validation differs. Please refer to the table below for further details on the behavior of each function.

Function	Behavior
DecodeHeader	Parses the input bit stream and populates mfxVideoParam structure. Parameters are NOT validated. Decoder may or may not be able to decode the stream
Query	Validates the input parameters in mfxVideoParam structure. Returns the corrected parameters (if any) or MFX_ERR_UNSUPPORTED if parameters cannot be corrected. Parameters set as zero are not validated.
QueryIOSurf	Does NOT validate the mfxVideoParam input parameters except those used in calculating the number of input surfaces
Init	Validates the input parameters in mfxVideoParam structure completely. Some parameters may be corrected if selected configuration is not compatible. If compatibility cannot be resolved MFX_ERR_INVALID_VIDEO_PARAM is returned. Note: GetVideoParam() may be used to retrieve the corrected parameter set

8 Decode

Intel® Media SDK decode operations provide an OS and platform neutral way to decompress bitstreams to raw frames.

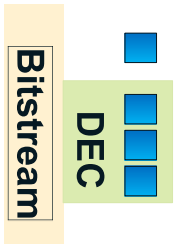
8.1 Decode overview

The following pseudo code illustrates the steps required for decode. Please note that this code is synchronous—better performance can be obtained from an asynchronous implementation.

```
set_up_params(&InitParam_d);
set_up_allocator(InitParam_d);

// Initialize DECODE
decode->DecodeHeader(bitstream, InitParam_d); decode->Init(InitParam_d);

// Decode frames from stream
while (bitstream) {
              read_from_file(&bitstream);
              find_free_work_frame(&frame_w);
              decode->DecodeFrameAsync(bitstream, frame_w, frame_d, sync_d);
              SyncOperation(sync_d);
              write_frame(frame_d);
}

// Decode frames still waiting to drain
while (MFX_ERR_MORE_SURFACE == sts) {
              find_free_work_frame(&frame_w);
              decode->DecodeFrameAsync(NULL, frame_w, frame_d, sync_d);
              SyncOperation(sync_d);
              write_frame(frame_d);
}

// Close components
decode->Close();
delete_surfaces_and_allocator();
```

Here the decoder is initialized from the stream via DecodeHeader. There is a drain loop at the end with a NULL bitstream pointer to access buffered frames.

The main asynchronous loop processes multiple frames at a time:



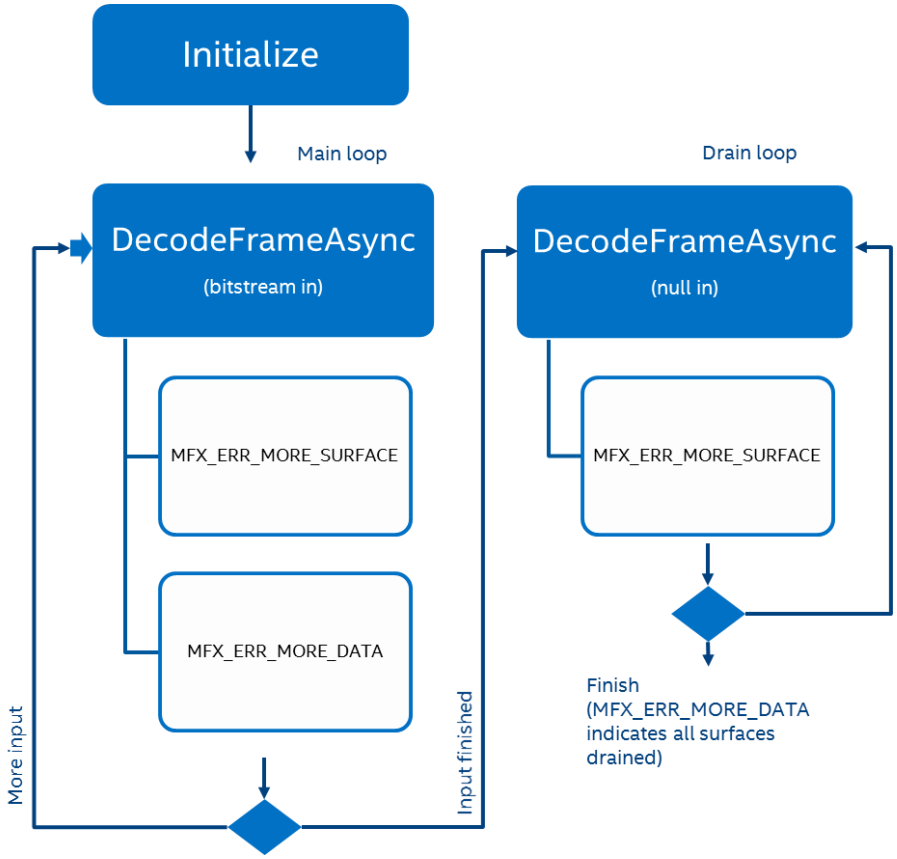
When the bitstream is done, frames may still be in the decoder, so an additional loop is required to drain them.

8.2 Decode states:

Intel® Media SDK stages are implemented as asynchronous state machines to maximize performance. The decoder will progress through multiple states. This requires a loop so that decode can be called multiple times to advance through those states. With each iteration decode informs the application what is needed next.

The simplified diagram below shows some of the most common states/return codes for decode. Note they are both "errors". This does not mean that anything is wrong – just that the application is required to provide something for decode to proceed.

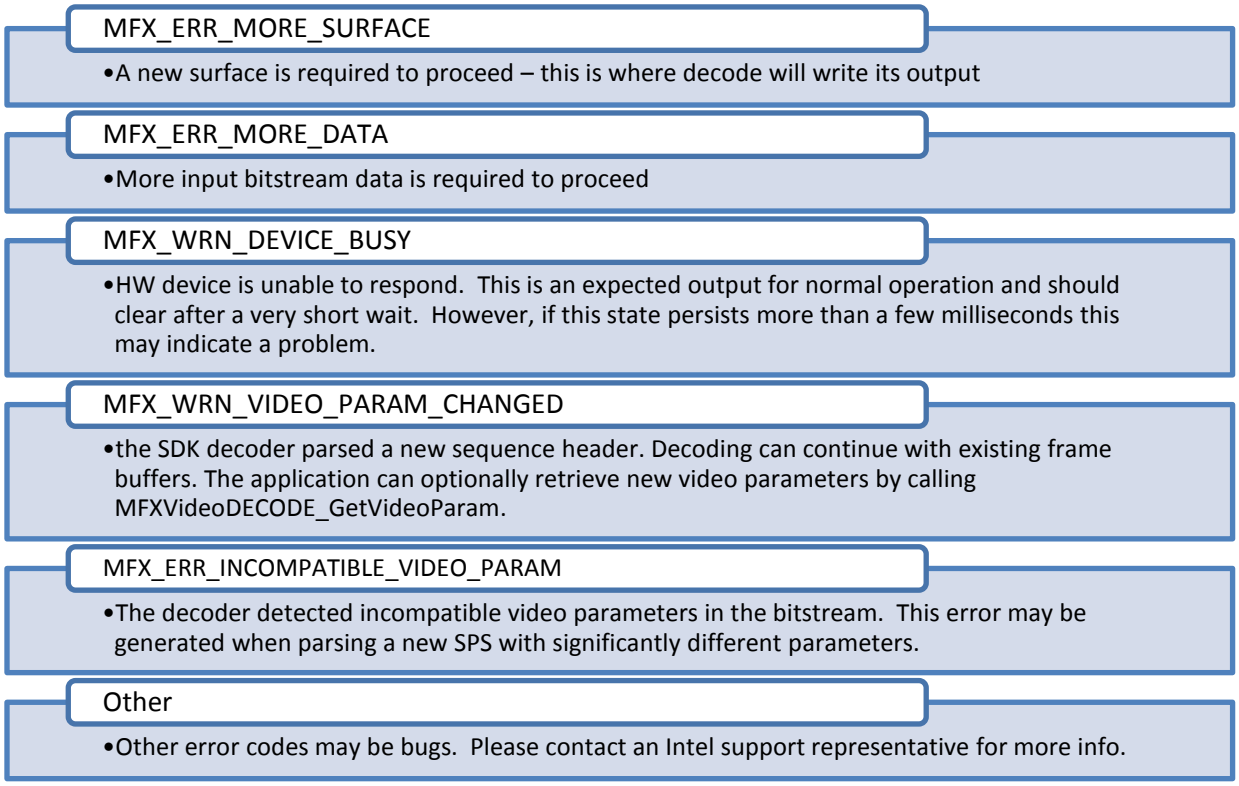
- MFX_ERR_MORE_SURFACE: implementation is asynchronous (even when async depth=1). Decode may queue up several surfaces to write results to before a decoded surface is available to use.
- MFX_ERR_MORE_DATA: decode has read to the end of the bitstream supplied and requires more input bits to continue decoding.



The diagram shows two loops. The main loop works on incoming bitstream data, so it will continue requesting more. However, when the input is finished decode can be called with a null pointer instead of a bitstream pointer. This indicates to the SDK that no more input is coming, so any buffered frames should be processed without additional bitstream input. However, there will be requests for more surfaces until all buffered bitstream data is drained.

MFX_ERR_NONE (not shown) usually indicates that no additional inputs are required so a decoded frame will be available after the next sync.

Please note that a real-world implementation will need to deal with more states. The diagram below has a more complete list of expected return codes:



Decode return codes and meanings

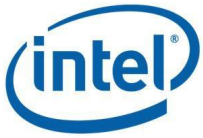
8.3 Robust decoding of network streams

Videos are increasingly streamed over a network instead of played from files. This creates many more opportunities for errors in the incoming bitstream.

A proven strategy to work with network streams is to check the Corrupted flag of each decoded frame. If there is any kind of issue (Corrupted !=0), then drain the pipeline and reset.

As an example, the following code snippets form a main loop showing all of the steps together:

```
while (1==1) {
    sts = mfxDEC.Reset(&mfxVideoParams); // Reset the Media SDK decoder
    stillgoing=1;
```



This is an “extra” reset the first time through the loop. Not optimal, but this makes it easy to come back to this point when corruption is encountered. There is an inner loop which iterates until reset is needed.

```
    while (stillgoing==1) {
      sts = mfxDEC.DecodeFrameAsync(&mfxBS, pmfxSurfaces[nIndex],
&pmfxOutSurface, &syncp);
```

Multiple calls to decode may happen per frame of output, advancing through the states below. This will loop back to DecodeFrameAsync unless sts is MFX_ERR_NONE.

```
      switch (sts){
        case MFX_WRN_DEVICE_BUSY: MSDK_SLEEP(1);break;
        case MFX_ERR_MORE_DATA:
          sts = ReadBitStreamData(&mfxBS, fSource); break;
        case MFX_ERR_MORE_SURFACE:
          nIndex = GetFreeSurfaceIndex(pmfxSurfaces, numSurfaces);break;
        case MFX_WRN_VIDEO_PARAM_CHANGED:break;
        case MFX_ERR_NONE:break;
        default: stillgoing=0;break;
      }
      if (MFX_ERR_NONE != sts) continue;
```

A SyncOperation is required to ensure the output frame is ready for use. If the Corrupted flag is not zero then exit the loop.

```
      sts = session.SyncOperation(syncp, 60000);

      if (pmfxOutSurface->Data.Corrupted!=0) {
          printf("Corruption detected: Frame number: %06d Corrupted=%03x\n",
nFrame,pmfxOutSurface->Data.Corrupted );
          stillgoing=0;
      }
       else {
         //frame output would happen here, after sync
        }
    }
```

An additional loop with a NULL bitstream is required to ensure all buffered frames are drained from the decoder.

```
    //drain remaining frames
    stillgoing=1;
    while (stillgoing==1) {
      sts = mfxDEC.DecodeFrameAsync(NULL, pmfxSurfaces[nIndex],
&pmfxOutSurface, &syncp);
      switch (sts){
        case MFX_WRN_DEVICE_BUSY:MSDK_SLEEP(1);break;
        case MFX_ERR_MORE_SURFACE:
          nIndex = GetFreeSurfaceIndex(pmfxSurfaces, numSurfaces);break;
        case MFX_ERR_NONE:
          sts = session.SyncOperation(syncp, 60000);break;
        default:
          stillgoing=0;break;
        }
      if (MFX_ERR_NONE != sts) continue;
    }
```

After draining the outside loop starts again with a reset.

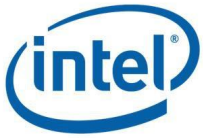
This method can avoid complications due to driver problems parsing corrupted streams, creating a way to decode that is robust enough to deal with real world input stream issues.

8.4 Decoder Bitstream Repositioning

Media SDK decoder bitstream repositioning is described in the Media SDK manual but the following information explains the concept in the context of container handling which is tightly connected to stream repositioning.

Please follow these steps to reposition a stream during a decoding session:

1) Invoke decoder MFXVideoDECODE_Reset() to reset the decoder
2) Clear current Media SDK bit stream buffer (mfxBitStream)
3) Reposition demuxer (splitter) to desired frame backward or forward in stream It is recommended that demuxer does the following before delivering frames to decoder:



- Reposition to closest I-frame to the desired position
- Insert sequence header (sequence parameter set for H.264, or sequence header for MPEG-2 and VC-1) into stream before the I-frame. Note: Some streams may already contain sequence headers before each I-frame
- If sequence header is inserted before a frame that is not an I-frame, decoder may produce artifacts

4) Read data into Media SDK bit stream buffer from new demuxer position
5) Resume decoding by calling DecodeFrameAsync as usual.

If the Media SDK decoder does not find any sequence headers while decoding from the new position DecodeFrameAsync will continuously return MFX_ERR_MORE_DATA (effectively asking for more bitstream data)

9 Encode

Intel® Media SDK encode operations provide an OS/platform neutral way to compress raw frames to bitstreams.

9.1 Encode overview

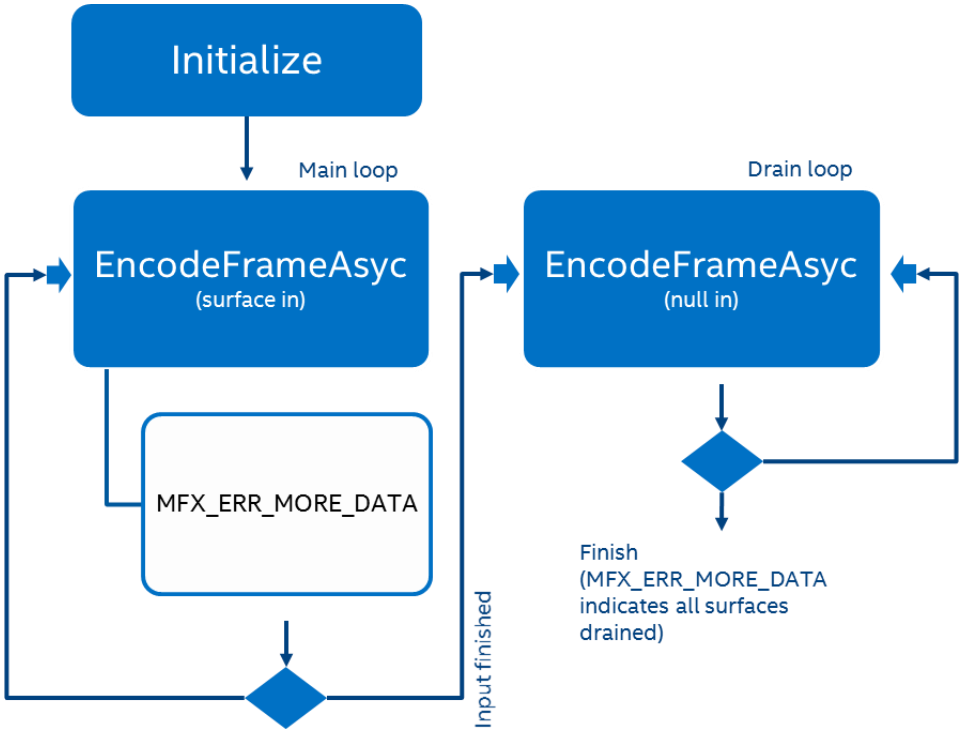
The following pseudo code illustrates the steps required to encode.

```
set_up_params(&InitParam_e);
set_up_allocator(InitParam_e);

// Initialize ENCODE
encode->Init(InitParam_e);

// Encode stream from frames
while (more frames to encode) {
    find_free_frame(&frame_e);
    load_frame(&frame_e);
    encode->EncodeFrameAsync(NULL, frame_e, bitstream, sync_e);
    SyncOperation(sync_e);
    write_bitstream(bitstream);
}
// Encoded bitstream still waiting to drain
while (MFX_ERR_MORE_SURFACE == sts) {
     encode->EncodeFrameAsync(NULL, NULL, bitstream, sync_e);     SyncOperation(sync_e);
     write_bitstream(bitstream);
}

// Close components encode->Close();
delete_surfaces_and_allocator();
```

9.2 Encode States

As with decode, SDK encode is implemented as a state machine. The encoder will progress through multiple states, requiring a loop so that encode can be called multiple times. With each iteration encode informs the application what is needed next.

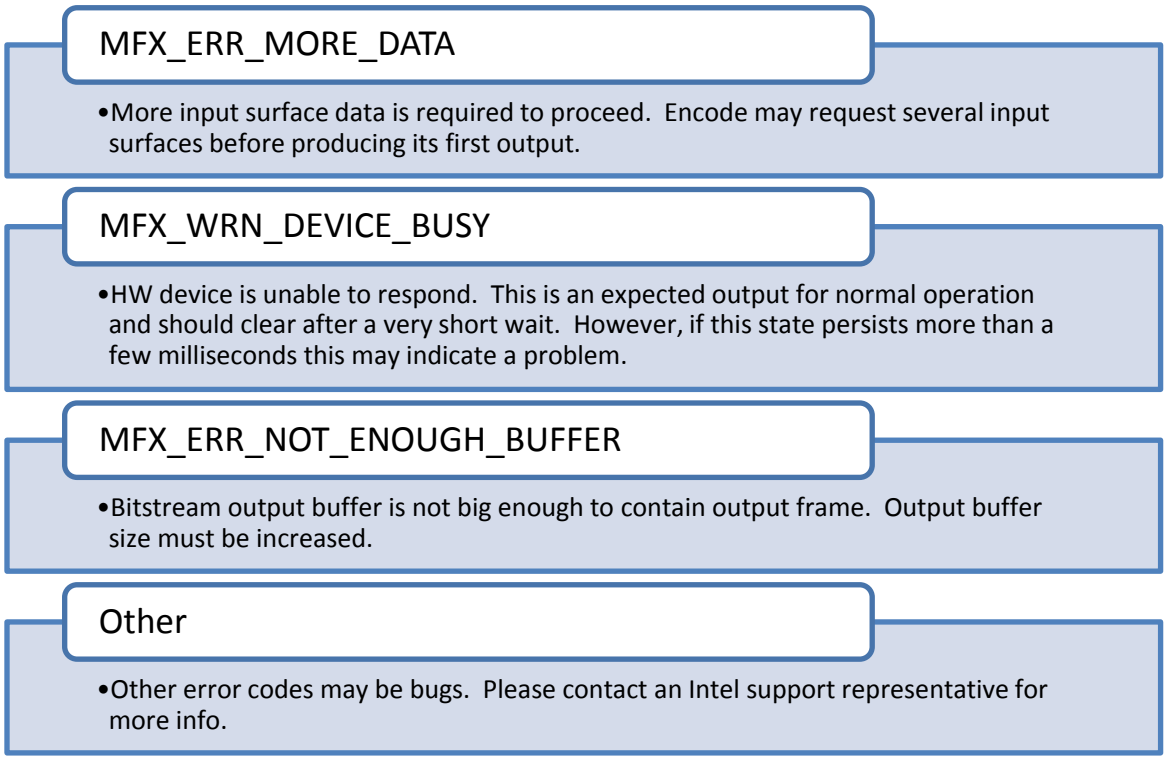
The simplified diagram below shows some of the most common states/return codes for encode.



MFX_ERR_MORE_DATA: indicates encode needs more frames. Since the implementation is asynchronous many frames may be enqueued before bitstream data is available. Final frames may be accessed by calling encode with a NULL input surface pointer.

MFX_ERR_NONE indicates that bitstream data for the encoded frame will be available after the next sync.

Please note: a real world implementation will need to handle more states. The diagram below shows a more complete list of encode errors/warnings:

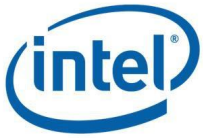


Encode states and meanings

9.3 Encode Bitstream Buffer Allocation with GetVideoParam

The MFXEncode_GetVideoParam function provides a way to determine the encoder's working parameter set. One of the values the function returns is the size of the bit stream buffer needed for the workload, BufferSizeInKB. Use this function prior to bitstream allocation. It returns the minimal buffer size required for most scenarios. It is possible to encounter an increase in the required minimum buffer allocation. When this occurs, the Encoder returns MFX_ERR_NOT_ENOUGH_BUFFER. If this situation arises, call MFXEncode_GetVideoParam again to retrieve the new allocation size, reallocate the correct amount of buffers, and then rerun **ENCODE** with the new sizes.

Media SDK encoder note: Encoder can control insertion of sequence headers via the "IdrInterval" parameter. Make sure that "GopPicSize" and "GopRefDist" values have been specified explicitly to



ensure correct behavior. Also keep in mind that "IdrInterval" has different behavior across codecs. More details can be found in Media SDK manual.

9.4 Encode Quality and Performance Settings

There are two main ways to affect encode output quality

1) Encoder algorithms and parameters (main control is covered in this section)
2) Bitrate control (next section)

The SDK predefines sets of parameters for a range of target usages (TU). These are for developer convenience. Instead of needing to understand and manage the interrelationships of multiple parameters, they are pre-grouped into 7 levels.

The default TU setting is "Balanced" (TU4), and is set via the mfxInfoMFX.TargetUsage parameter. Use TU1 for best quality and TU7 for best speed, with the other values as intermediate points on the quality/speed spectrum:

	Best Quality ←----------→ Best Performance						
Target	TU1	TU2	TU3	TU4	TU5	TU6	TU7

Different codecs have different parameter groupings. In some cases only a subset of TUs are defined, with the rest aliased to the closest defined TU. TU settings can change by hardware generation as well as driver version so it is not possible to create a single static list of settings for each TU. However, settings used for the current TU level can be obtained at runtime with **MFXVideoEncode_GetVideoParam**.

9.5 Bitrate Control Methods and Parameters

Bitrate control (BRC) algorithm choices are an important part of a video quality solution. BRC approaches can often make as much or more difference in encode quality as algorithm quality settings.

Intel encoders include many types of bitrate control, each designed for different types of use cases. There is no general purpose choice which is optimal across all scenarios. Choose the mode that best matches your requirements:

Complications of Measuring Bitrate

Unlike uncompressed video, frame sizes of encoded bitstreams are expected to vary widely. IDR and I frames may be significantly larger than accompanying P and B frames. This is one of the core ideas of video compression: P and B frames store differences and don't encode entire pictures. B frames tend to be smaller than P frames. Content differences/scene changes can cause further variations.

Because the frame sizes vary so much it can be difficult to evaluate trends. Moving average, though frequently used, turns out to not be an ideal visualization tool. The size of the moving average window, unless very large, can introduce distortions depending on how it aligns with frame size patterns. More sophisticated signal processing algorithms designed for high variation data have better results. A Savitsky-Golay (savgol) filter is used here.

HRD/VBV Compliance

The Hypothetical Reference Decoder (HRD), previously known as Video Buffering Verifier(VBV) from the mpeg2 era, is a "leaky bucket" model. Conformance increases the likelihood that any decoder will be able to receive the stream as intended without delays or dropping frames. This is especially necessary if targeting older low cost hardware decoders. However, improved HW decoder resilience and the increased buffer sizes available to software decoders on modern processors can relax the need for strictness.

HRD is a simplified model, assuming that data is arriving according to individual frame size but exiting at the given bitrate. The goal is to keep the buffer from underflow or overflow.

- Underflow (buffer empty) occurs when the incoming data frames do not supply new data as fast as the specified exit rate for the HRD.
- Overflow (buffer beyond full) happens when the exit rate for the buffer is not large enough to absorb new frames as they arrive.

Maintaining HRD conformance can be expensive. For example, in some cases frames must be re-encoded. It also increases BRC algorithm complexity and performance impact. So it is there if you need it but there are many options to run without HRD conformance if not.

Legacy:

Algorithm	HRD/VBV Compliant	Windows	Linux	Usage

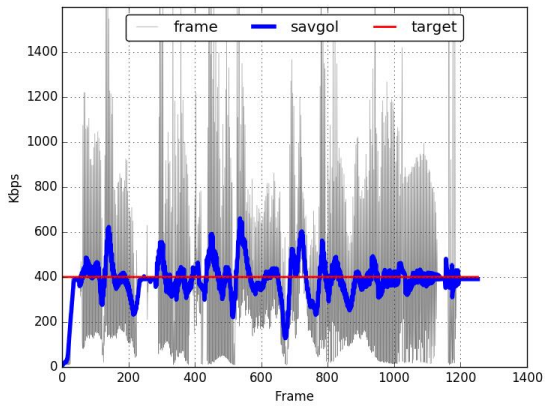
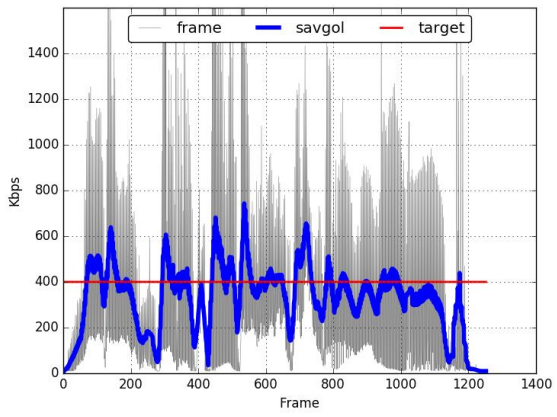
CBR	Yes (overflow, underflow)	Yes	Yes	Video conference, Video surveillance solutions
VBR	Yes(underflow)	Yes	Yes	Video survailance storage, Live streaming, Broadcast solutions

Codecs covered: H.264, H.265, MPEG2, MJPEG

Linux and Windows, hardware and software implementations

These are the standard algorithms available since the beginning of audio and video compression. In many cases there can be a better match via custom BRC or one of the “optimized” algorithms below.

- **Constant Bitrate (CBR)** attempts to minimize bitrate fluctuations between frames. In some cases this means padding (extra data) is added to the end of frames. CBR "optimizes" for constant flow of data, which means a significant portion of overall bitrate is wasted on padding instead of being used to encode frame details. Please note: CBR does not imply that all frames will be the same size, or that the output will not have spikes in bitrate. See more on **BufferSizeInKB** below.
- **Variable Bitrate (VBR)** allows more variations, and it allows immediate bitrate to fall without padding. This can often result in better quality at the same bitrate as CBR.

CBR	VBR
	

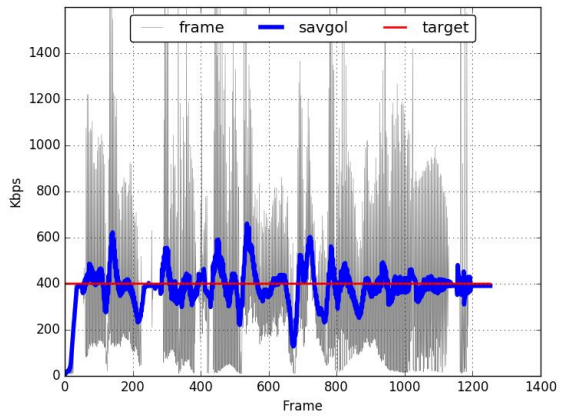
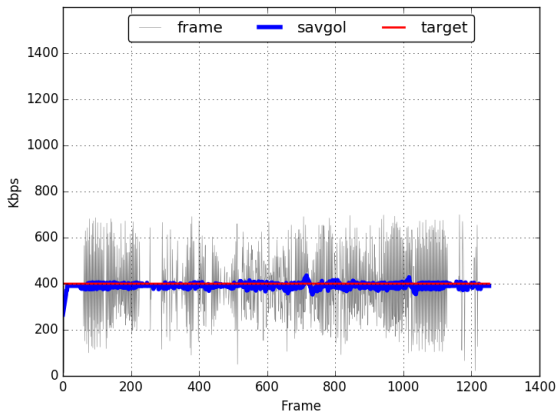
For graphs above:

- Input used: sintel-trailer-480p.mp4 from https://download.blender.org/durian/trailer/

- Transcoded with defaults using CBR and VBR BRC from simple_3_encode

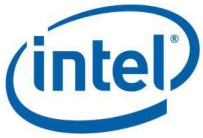
Important parameters:

- **TargetKbps**: this is the bitrate target for CBR/VBR in kilobits per second
- **BufferSizeInKB**: Smaller BufferSizeInKB means smaller frame size variations. At very small buffer sizes it becomes increasingly difficult to maintain HRD, but this is the best method to reduce CBR spikes. Larger buffers mean more variations can be allowed to preserve quality where needed. Default is 2 seconds of buffer. (BufferSizeInKB= TargetKbps*8/1024*2) Note that units are kilobytes, not bits. Buffer size should never be smaller than the largest expected group of ~5 frames.
- **NalHrdConformance**: By default, CBR and VBR algorithms focus on maintaining HRD compliance. Relaxing this requirement can improve performance and quality where these would be sacrificed for strict conformance. This should also be turned off if bitrate or related parameters are dynamically changed.

CBR (default 2 second buffer)	CBR (.5 second buffer)
	

File-to-file transcode optimized:

Algorithm	HRD/VBV Compliant	Windows	Linux	Usage
LA	No	Yes	Yes	Storage transcodes



LA_HRD	Yes	Yes	Yes	Storage transcodes; Streaming(where low latency is not a requirement)
ICQ	No	Yes	No	Storage solution
LA_ICQ	No	Yes	No	Storage solution

Codecs covered: H.264 (HW only, not available for SW)

ICQ and LA_ICQ are not available for Linux.

Lookahead algorithms

Lookahead (LA) BRC is an extension of VBR, representing a significant advance in automatic BRC for Intel codecs. It can provide large quality improvements, sometimes larger than a more expensive target usage, as CBR/VBR. It should be considered the default choice for many file-to-file transcode scenarios. An HRD-compliant flavor (LA_HRD) is available where this is needed.

Lookahead provides an answer to key VBR/CBR challenges: causes of large frame size variation like I-frames and scene changes can be anticipated. A buffer of frames (length configurable with the **LookAheadDepth** parameter) is analyzed for potential bitrate disruptions. This allows a "preview" or "lookahead" with many of the advantages of two-pass encoding. In effect, it is VBR with advance notice of frame size changes, higher variability, and longer latency.

The basic idea behind lookahead is to perform extensive analysis of the sequence including complexity, relative motion, and dependencies before encoding. It distributes available bit budget between frames to produce the best possible encoding quality. This approach generates good results on fast motion video and computer animation, improving objective metrics (SSIM, PSNR) and subjective video quality. Lookahead works with any GOP pattern but presence of B frames provides the best quality gain. One side effect is that it significantly increases encoding delay and memory consumption.

The increased latency of the lookahead approach may be a concern for streaming scenarios. Please see the next section for algorithms focusing on streaming/low latency scenarios.

Lookahead is only available for some codecs on Intel® Iris™ Pro Graphics, Intel® Iris™ Graphics and Intel® HD Graphics on Haswell architecture (4th Generation Core) and forward. It is not implemented yet for HEVC.

Important parameters:

- **TargetKbps**: this is the bitrate target for lookahead in kilobits per second. It is the only available rate control parameter for this BRC mode. Buffer size, MaxKbps, InitialDelayInKB, etc. are managed internally by lookahead and these settings are ignored.
- **LookAheadDepth** is the number of frames analyzed before encoding. Valid value range is from 10 to 100. To instruct the SDK encoder to use the default value the application should zero this field.
- **API level:** Media SDK must be initialized to use API version 1.7 or newer to use lookahead.

Intelligent Constant Quality algorithm

Intelligent Constant Quality (ICQ) and its lookahead extension (LA_ICQ) are the closest equivalent to Constant Rate Factor (CRF), which is the default BRC mode for popular codecs like x264 and x265 – the default h264 and h265 implementation in FFmpeg. Instead of enforcing a target bitrate, this algorithm allows large bitrate fluctuations to maintain a set quality level. This corresponds to the 1-51 standard quantization levels: 1=least quantization/best quality, 51=most quantization/best compression.

For file transcodes there can be many advantages to this approach similar to CBR vs. VBR. When attempting to maintain a target bitrate, bits may be used on frames/sequences where they are not visually as important – somewhat like CBR padding. This results in needing larger files for the same quality level.

Important parameters:

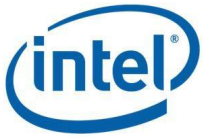
- **ICQQuality:** desired quality level (1-51). There is no targetkbps in ICQ mode.
- **LookAheadDepth:** used with LA_ICQ
- **API level:** ICQ is available in API 1.8 and higher

Streaming optimized:

Algorithm	HRD/VBV Compliant	Windows	Linux	Usage
VCM	Yes	Yes	No	Video conferencing
QVBR	Yes	Yes	No	Game/display streaming

Codecs covered: H.264 (HW only, not available for SW)

Not available for Linux



QVBR is a unique algorithm which combines a target CQ/quality level with bitrate constraints. This provides the best parts of ICQ and VBR for streaming scenarios. The large bitrate fluctuations of ICQ make it more suitable for file-to-file transcode. However, for network streaming changing bitrates can be punished by CDN policies. QVBR constrains the fluctuations around a target bitrate.

Important parameters:

- **QVBRQuality:** desired quality level (1-51).
- **TargetKbps**: bitrate target in kilobits per second

Video Conferencing Mode

The videoconferencing mode (VCM) algorithm is designed around specific assumptions which can be made for videoconferencing content:

- Low motion in most of the frame/constant background
- Low latency GOP structure (IPPP/no b frame or b-pyramid)
- No interlaced encode

It is designed to produce better quality at the same bitrate compared to other algorithms like CBR/VBR for this type of content.

Important Parameters:

- **TargetKbps**: bitrate target in kilobits per second

Other Parameters

BRCParamMultiplier can be used to achieve higher bit rate. This extends the range of the 16 bit TargetKbps to achieve higher values than can be represented in mfxU16. This also affects BufferSizeInKB, IntialDelayInKB and MaxKbps. The table below shows the scenario when this parameter is used.

Input: 720p - Park Joy YUV.

Expected Bitrate(in Mbps)	1	2	5	10	20	40	60	80	100
Obtained Bitrate without BRCParamMultiplier(in Mbps)	1.03	2.03	5.24	10.53	20.78	41.20	60.55	15.11	36.07
Obtained Bitrate with BRCParamMultiplier(in Mbps)	1.02	2.08	5.24	10.68	20.50	40.97	61.31	84.67	103.38

This table shows that without the multiplier bitrate cannot increase beyond 65535 Kbps.

9.6 Custom frame and MB QP with CQP for custom BRC

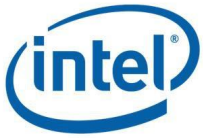
Why custom bitrate control?

Achieving good video compression means managing tradeoffs. In most cases video compression is lossy, with high correlation between compression ratio and loss of fidelity with the original. Quality can be increased with more complex and computationally intense algorithms inside the codec and via additional video analysis/quality feedback steps, but at the cost of more hardware resources. Bitrate control (sometimes called rate control or BRC), is a crucial part of optimizing how codecs manage these tradeoffs to produce the best possible result for the specific target use case. While many automatic BRC options exist, there is no one size fits all solution.

The SDK provides many pre-implemented BRC algorithm choices (see previous section) as well as the ability to build custom solutions. Custom BRC is available by setting **RateControlMethod=MFX_RATECONTROL_CQP**.

Using CQP BRC opens up many options. Used in its most basic mode QP levels can be defined once at initialization for I, P, and B frames for a true “constant” QP encode. But it can do much more. Some examples of the capabilities CQP mode unlocks:

- assign a new quantization parameter (QP) to each frame.
- manage parameters at macroblock level



This allows you to differentiate with your own “secret sauce” algorithms to build BRC algorithms custom designed for your requirements.

Without question, the best coding efficiency for Intel codec implementations can be obtained via CQP plus custom content analysis. CQP can have significant performance advantages as well since this avoids the analysis and possible re-encoding done by automatic BRC modes.

CQP can provide many advantages, but it should only be considered for cases where the benefits outweigh the costs. The main tradeoff to keep in mind for custom BRC is development and validation time. Simply using CQP mode with a static QP for an entire sequence is likely to result in significantly lower quality at comparable bitrates vs. existing BRC implementations. As there is no simple or linear mapping between QP and either perceived quality or resulting bitrate, the application developer must provide a feedback system to adapt QP to frame-by-frame video conditions. Validation should also be considered. BRC algorithm tweaks providing improvements for a small set of inputs are notoriously prone to regressions when applied to a wider variety of streams. While the automatic BRC modes covered in the previous section are extensively tested, validation for new BRC built with CQP is the responsibility of the developer.

Basic structure of a CQP solution

Encoding with an automated BRC option using the SDK is relatively straightforward. Rate control options are simply parameters to the encoder given at initialization. The main encode loop only needs to pass raw video frames to the encoder. After the encode stage completes an output bitstream is available which, if using a BRC mode with HRD/VBV support, is compliant with decoders which depend on working within a small buffer.

With custom BRC the application has much more control and is required to implement more stages of the process. In addition to feeding raw frames to the encoder, an application must analyze the input video and update QP (by frame or MB). If HRD compliance is required, the application must monitor output frame sizes and possibly re-encode if it is not possible to maintain conformance by adjusting subsequent frame sizes.

OpenCL can be an ideal tool for BRC analysis. This keeps more work on the GPU, which can avoid synchronization and copy overhead. Intel also provides OpenCL extensions to access the same fixed function hardware used by Media SDK codecs. For example, this means that the Video Motion Estimation (VME) engine is accessible separate from encode so it can be part of a custom “lookahead” approach which analyzes complexity of incoming frames well before they reach the encoder to allow for better quantization decisions.

The example code presented in this section is extremely simplistic and intended only to provide a minimal starting point. Please watch for extensions to this section in future versions of this guide.

Custom BRC controls provided by the SDK

First, CQP mode must be selected at encoder initialization:

```
mfxEncParams.mfx.RateControlMethod = MFX_RATECONTROL_CQP;
```

Changing CQP by frame also requires using the mfxEncodeCtrl parameter when calling EncodeFrameAsync. This is not required in many automatic BRC cases and is set to null in many SDK examples.

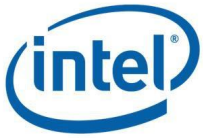
Syntax

A quick review of Media SDK encode syntax is important for this section to clarify use of the encode control structure.

```
mfxStatus MFXVideoENCODE_EncodeFrameAsync(mfxSession session, mfxEncodeCtrl
*ctrl, mfxFrameSurface1 *surface, mfxBitstream *bs, mfxSyncPoint *syncp);
```

Parameters

mfxSession session	(in) SDK session handle
mfxEncodeCtrl *ctrl	(in) Pointer to the mfxEncodeCtrl structure for per-frame encoding control; this parameter is optional—it can be NULL—if the encoder works in the display order mode
mfxFrameSurface1 *surface	(in) Pointer to the input raw frame surface structure
mfxBitstream *bs	(out) Pointer to the output bitstream
mfxSyncPoint *syncp	(out) Pointer to the returned sync point associated with this operation



The mfxEncodeCtrl structure is handled similarly to mfxFrameSurface inside the Media SDK asynchronous framework. It can simplify implementation to handle surfaces and control structures together. This means that there should be as many control structures as encode surfaces, and the same array index should be used for both. The code below shows adding control structure allocation to surface allocation.

```
// Query number of required surfaces for encoder
mfxFrameAllocRequest EncRequest;
memset(&EncRequest, 0, sizeof(EncRequest));
sts = mfxENC.QueryIOSurf(&mfxEncParams, &EncRequest);
MSDK_CHECK_RESULT(sts, MFX_ERR_NONE, sts);

const mfxU16 nEncSurfNum = EncRequest.NumFrameSuggested;

// Allocate surfaces for encoder
// - Width and height of buffer must be aligned, a multiple of 32
// - Frame surface array keeps pointers all surface planes and general frame
info
mfxU16 width = (mfxU16) MSDK_ALIGN32(EncRequest.Info.Width);
mfxU16 height = (mfxU16) MSDK_ALIGN32(EncRequest.Info.Height);
mfxU8 bitsPerPixel = 12;        // NV12 format is a 12 bits per pixel format
mfxU32 surfaceSize = width * height * bitsPerPixel / 8;
mfxU8* surfaceBuffers = (mfxU8*) new mfxU8[surfaceSize * nEncSurfNum];

mfxEncodeCtrl *pEncodeCtrl=new mfxEncodeCtrl[nEncSurfNum];
```

As the application loops through encodes, the surface locking mechanism provides a way to determine which surfaces are in use. This index can be used for the control surfaces too.

```
//
// Stage 1: Main encoding loop
//
 while (MFX_ERR_NONE <= sts || MFX_ERR_MORE_DATA == sts) {
     nEncSurfIdx = GetFreeSurfaceIndex(pEncSurfaces, nEncSurfNum);   // Find
free frame surface
     MSDK_CHECK_ERROR(MFX_ERR_NOT_FOUND, nEncSurfIdx, MFX_ERR_MEMORY_ALLOC);

     sts = LoadRawFrame(pEncSurfaces[nEncSurfIdx], fSource);
     MSDK_BREAK_ON_ERROR(sts);

     pEncodeCtrl[nEncSurfIdx].QP=currQP;


     for (;;) {
         // Encode a frame asychronously (returns immediately)
         sts = mfxENC.EncodeFrameAsync(&pEncodeCtrl[nEncSurfIdx],
pEncSurfaces[nEncSurfIdx], &mfxBS, &syncp);
```

QP Adjustment Analysis

While a "real" implementation would need something far more complex, the example simply keeps a moving average of frame size converted to kbps:

```
kbps=(mfxBS.DataLength*8.0/1024.0)*((double)options.values.FrameRateN/
(double)options.values.FrameRateD);
```

After every few frames a check if the moving average is getting too high or too low allows simplistic adjustment. If average bitrate is too high, QP is increased. Subsequent frames are encoded with less detail, which should move the bitrate back to acceptable range if the frames continue to have similar characteristics. Alternately, if average bitrate is too low then QP can be moved down, providing more detail for easier sequences.

```
if (kbps_mov_avg > options.values.Bitrate)
{
    currQP+=1;
}
else if (kbps_mov_avg < options.values.Bitrate)
{
    currQP-=1;
}
```

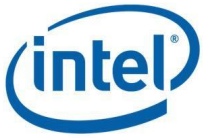
This is a simplified example, obviously not intended to be used in a real world setting. The goal here is to illustrate where more complex analysis could be added.

Per-MB QP

An EnableMBQP flag exists in mfxExtCodingOption3 for h264. If turned on in CQP mode, a mfxExtMBQP extended parameter buffer can be attached to mfxEncodeCtrl before encode. There should be the same number of parameter buffers as encode controls.

The mfxExtMBQP structure:

```
typedef struct {
    mfxExtBuffer    Header;

    mfxU32 reserved[11];
    mfxU32 NumQPAlloc;
    union {
        mfxU8  *QP;
        mfxU64 reserved2;
```



```
    };
} mfxExtMBQP;
```

Providing a buffer of QP values here (again, need the same number of buffers as encode control structures) passes an array of QP values for each macroblock in the encoded bitstream. NumQPAlloc specifies the size of the QP array, one value for each macroblock in the frame. MB QP values are in raster order. Similar controls exist to disable skip encoding to keep QP regions intact. For this see **mfxExtMBDisableSkipMap.**

9.7 IDR Insertion

Frame type can be controlled by the encode control structure in a similar way as frame and macroblock QP in the previous section. Application scene change detection with the start of a new GOP can be an effective way to improve quality. Please watch for more on this topic in the next dev guide update.

9.8 Custom control of encoded AVC SPS/PPS data

Intel® Media SDK provides direct control of a subset of SPS/PPS header parameters defined by the AVC standard. For control of specific SPS/PPS parameters that cannot be controlled via the SDK API, please use the following method:

1) Configure encoder with a set of parameters closest matching the desired SPS/PPS set
2) After initalizing the encoder call GetVideoParam() with extended buffer mfxExtCodingOptionSPSPPS to extract the SPS and PPS selected by the encoder
3) (make sure to allocate sufficient space for storage of SPS and PPS buffer)
4) Make the desired modifications to SPS and/or PPS buffer
5) Apply the changes to the encoder by calling Reset(), referencing the new mfxVideoParam structure including the mfxExtCodingOptionSPSPPS extended buffer. Note that, when using this method the SPS/PPS parameters set via mfxVideoParams will be overwritten by the custom SPS/PPS buffer
6) The simple example below shows how to manually control SPS “constraint_set” values. In this case we are setting “constraint_set1” flag to 1 (indicates constrained baseline profile). The same approach can be used to control any SPS or PPS parameters manually.

```
// ... Encoder initialized
mfxExtCodingOptionSPSPPS m_extSPSPPS; MSDK_ZERO_MEMORY(m_extSPSPPS);
m_extSPSPPS.Header.BufferId = MFX_EXTBUFF_CODING_OPTION_SPSPPS;
m_extSPSPPS.Header.BufferSz = sizeof(mfxExtCodingOptionSPSPPS);
// Allocate sufficient space for SPS/PPS buffers (100 is enough)
m_extSPSPPS.PPSBuffer = (mfxU8*)malloc(100); m_extSPSPPS.SPSBuffer =
(mfxU8*)malloc(100);  m_extSPSPPS.PPSBufSize = 100;
m_extSPSPPS.SPSBufSize = 100;

// Add SPSPPS extended buffer to encoder parameter extended buffer list
//    std::vector<mfxExtBuffer*> m_EncExtParams
//    mfxVideoParam m_mfxEncParams
m_EncExtParams.push_back((mfxExtBuffer *)&m_extSPSPPS);
m_mfxEncParams.ExtParam = &m_EncExtParams[0];
m_mfxEncParams.NumExtParam = (mfxU16)m_EncExtParams.size();
// Retrieve selected encoder parameters
mfxStatus sts = m_pmfxENC->GetVideoParam(&m_mfxEncParams);
// Very simple representation of first set of SPS buffer members typedef
struct {
    unsigned char NAL_bytes[5];      unsigned char SPS_Profile;
    unsigned char SPS_Constraint_Set;    unsigned char SPS_Level;
    // ... rest of SPS is ignored (needs parsing)  } BasicSPS;
BasicSPS* mySPS = (BasicSPS*) m_extSPSPPS.SPSBuffer;
mySPS->SPS_Constraint_Set |= 0x40; // Set constraint_set1 to 1
// Reset encoder with modified SPS buffer sts = m_pmfxENC-
>Reset(&m_mfxEncParams);
```

9.9 Dynamic encode parameter changes

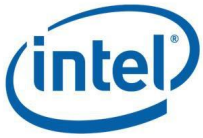
A few parameters can be changed without re-initialization under limited conditions:

PicStruct (if MFX_PICSTRUCT_UNKNOWN is specified at initialization) – this allows handling of mixed progressive/interlaced

Frame Width/Height (if <= initial resolution). Surfaces larger than needed for encode can be reused, but reset (and potentially re-allocation) is required if changing resolution parameters to larger than initial resolution.

Reset or re-initialization is the default requirement for most parameter changes, including:

Aspect ratio



IDR insertion

SPS header change (including VUI parameters)
Change in number of temporal layers

GOP settings

Number of slices

Async depth

Reference settings

Bitrate control (BRC) method and settings (note: if BRC settings are expected to change HRD conformance cannot be guaranteed. Setting NalHrdConformance to OFF can sometimes improve performance and quality by communicating relaxed requirements to Media SDK)

Resolution change (if > initial resolution)

Output by field or frame (FieldOutput)

LowPower mode

MaxSliceSize

To reset/reinitialize all buffered frames must first be drained from the pipeline.

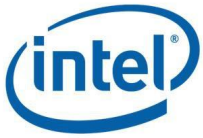
Please watch for more details on this topic in a future release of this guide.

9.10 Inserting SEI messages into AVC streams

Intel® Media SDK supports insertion of arbitrary SEI messages into encoded streams. This is achieved by adding payload (mfxPayload) to the encoded frame by specifying the encoder parameters via mfxEncodeCtrl.

The following code example showcases how to insert the SEI message type "user_data_registered_itu_t_t35", commonly used to carry closed captioning information.

```
#define SEI_USER_DATA_REGISTERED_ITU_T_T35  4

typedef struct
{
  unsigned char countryCode;   unsigned char countryCodeExtension;
  unsigned char payloadBytes[10];  // Containing arbitrary captions
} userdata_reg_t35;

// Insert SEI_USER_DATA_REGISTERED_ITU_T_T35 into payload userdata_reg_t35
m_userSEIData;
m_userSEIData.countryCode           = 0xB5; m_userSEIData.countryCodeExtension  =
0x31;
m_userSEIData.payloadBytes[0]       = 0x41; // payloadBytes[] containing captions

mfxU8 m_seiData[100]; // Arbitrary size mfxPayload m_mySEIPayload;
MSDK_ZERO_MEMORY(m_mySEIPayload);
m_mySEIPayload.Type    = SEI_USER_DATA_REGISTERED_ITU_T_T35;
m_mySEIPayload.BufSize = sizeof(userdata_reg_t35) + 2;  // 2 bytes for header
m_mySEIPayload.NumBit  = m_mySEIPayload.BufSize * 8; m_mySEIPayload.Data    =
m_seiData;

// Insert SEI header and SEI msg into data buffer
m_seiData[0] = (mfxU8)m_mySEIPayload.Type;         // SEI type
m_seiData[1] = (mfxU8)(m_mySEIPayload.BufSize – 2) // Size of following msg
memcpy(m_seiData+2, &m_userSEIData, sizeof(userdata_reg_t35));

mfxPayload* m_payloads[1];   m_payloads[0] = &m_mySEIPayload;

// Encode control structure initialization
mfxEncodeCtrl m_encodeCtrl; MSDK_ZERO_MEMORY(m_encodeCtrl);
m_encodeCtrl.Payload = (mfxPayload**)&m_payloads[0];
m_encodeCtrl.NumPayload = 1;

// Use encode control structure while calling encode m_pmfxENC-
>EncodeFrameAsync(&m_pEncodeCtrl,
                           &m_pEncSurfaces[nEncSurfIdx],
                           &pCurrentTask->mfxBS,
                           &pCurrentTask->EncSyncP);
```



10 Video Filters

Intel® Media SDK Video Processing (VPP) provides several commonly used frame processing algorithms for media pipelines. While the encode and decode SDK pipeline stages work with compressed bitstreams, VPP takes raw frames in and outputs raw frames with specified filters applied. This section provides details on how these filters are implemented and how they can be used in applications.

There are two main types of filters, each with its own activation method.

1. Mandatory filters: implicitly activated by differences in mfxInfoVPP In/Out mfxFrameInfo struct values
2. Secondary/Hint filters: activated by attaching extended parameter buffers to VPP parameters at VPP initialization

Note: VPP filters are not intended to be comprehensive. The main goal of VPP is to provide access to hardware accelerated features. Not all filters have a corresponding software implementation. Additional filters can be added by developers. These can run on the CPUs, but additional efficiency can be gained by using Intel® SDK for OpenCL™ Applications with Intel® Media SDK.

Mandatory Filters:

VPP Filter	Parameters Checked	Description
Surface format conversion	ColorFourCC, ChromaFormat	Execute conversion filter to match output color format.
Deinterlace/Inverse Telecine	PicStruct	Convert interlaced input to progressive output.
Scaling (Resize/crop)	Width, Height, Crop{X,Y,W,H}	If there is a difference in frame size or crop settings between input and output frames, run the resize stage.
Frame rate conversion	FrameRateExtN, FrameRateExtD	Support arbitrary input/output frame rate conversion based on simple frame dropping / repetition.

These filters are called “mandatory” since they are in all versions of Media SDK. However, underlying algorithms and capabilities may be different based on combination of hardware generation, OS, and driver version. Use of query mechanisms is recommended to determine specifics of what is covered at runtime.

Secondary/Hint Filters

VPP Filter	Configuration Method	Description
Composition	MFX_EXTBUFF_VPP_COMPOSITE	Combine frames and/or images with optional blending
Denoise	MFX_EXTBUFF_VPP_DENOISE	Remove image background noise. (Separate from deblocking filter inside AVC, HEVC, etc.)
Detail enhancement	MFX_EXTBUFF_VPP_DETAIL	Enhance image contrast.
ProcAmp	MFX_EXTBUFF_VPP_PROCAMP	Correct image brightness, contrast, saturation and hue settings.

These filters may not be available for all systems. Software implementations may not be available. Checks should be added at runtime to determine if these features exist.

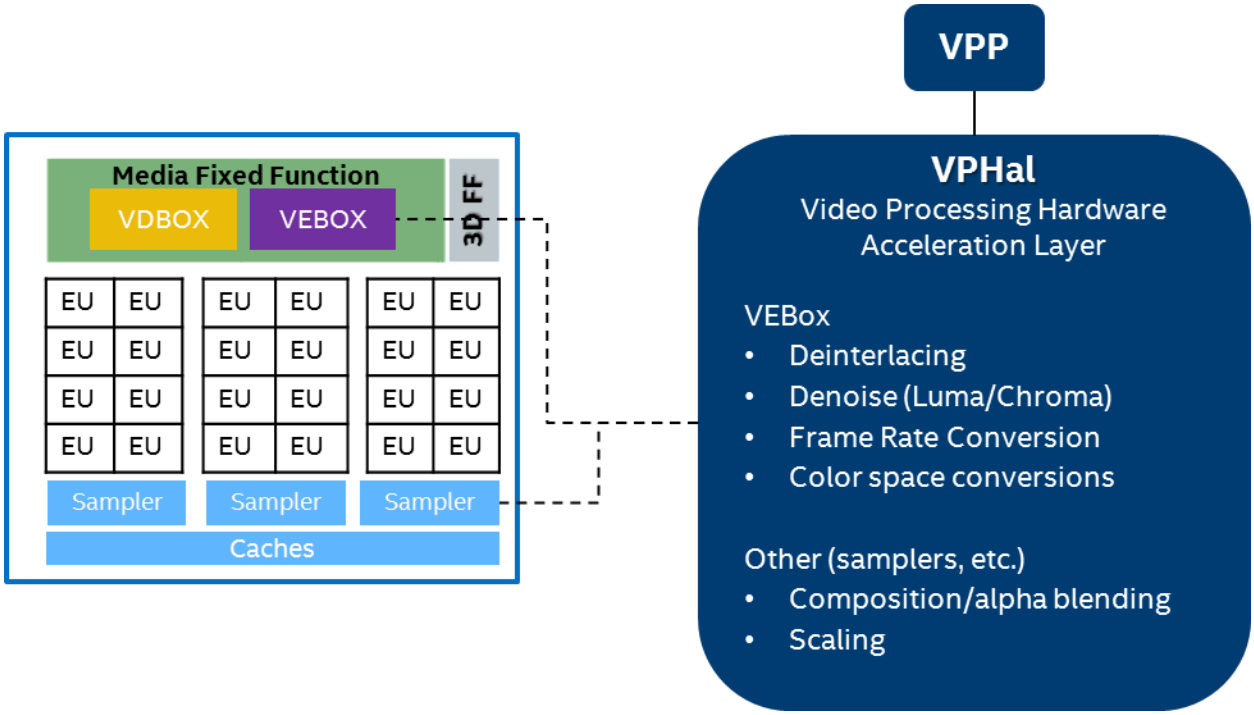
For composition: basic support requires API 1.8. API 1.9 adds blending features. This is a hardware only feature.

For denoise and detail enhancement: there are many ways that these can be disabled which are external to Media SDK. In some cases it may not be immediately apparent that, even though requested at VPP initialization with no errors/warnings, no action is taken by the filter. This is a known issue which may be worked around by application runtime checks. For example, pass a test frame through VPP and and check if any pixels changed.

Architecture:

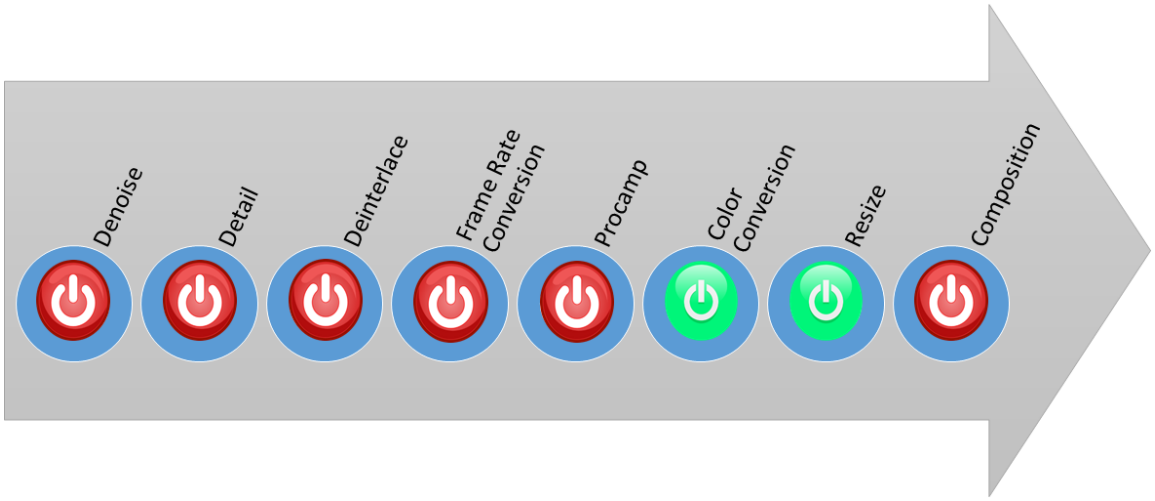
Intel® Media SDK VPP filters are designed to efficiently use the heterogeneous features of Intel GPUs. These features are accessed through the VPHal (Video Processing Hardware Acceleration Layer) underneath the GPU driver device driver interface (DDI). In VTune VPP operations are marked with a VPHal label instead of the filter used. VPHal provides a consistent interface to hardware blocks which potentially improve with each new hardware architecture.

- VEBox (Fixed function video enhancement)
- Execution units (general purpose GPU processors)
- Samplers (image access with hardware accelerated interpolation)



The filter implementation can be thought of as a pipeline with fixed order. VPP parameters turn various operations on or off. The order cannot be changed at the application level within a single VPP instance. However, the internal order of the chain can change between different hardware generations and driver versions. When many operations are used in a media pipeline it may be necessary to use multiple VPP instances to guarantee operation order. This means that the application handles passing surfaces between VPPs to manually control the chain.

Please note: the diagram below does not represent the exact order of the algorithm chain. The intent is to illustrate the fixed order with binary on/off switches for each algorithm.



An example VPP pipeline with some stages activated

Use/DoNotUse lists

In addition to supplying parameters as described above, two additional extended parameter types exist.

- VPP_DOUSE
- VPP_DONOTUSE

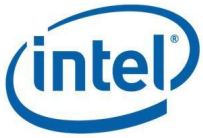
These allow passing arrays of algorithm identifiers to VPP as additional hints.

The VPP_DONOTUSE list can be important for performance optimization. In some cases the pathway for an algorithm might be executed even though it was not specifically requested. For example, the detail enhancement stage may run with default settings even if the extended parameter buffer to provide configuration values was not attached. Multiple VPP stages may be in the pipeline and this could avoid running redundant processing steps. This configuration gives developers more control over the filter chain.

Similarly, VPP_DOUSE may help to make sure a "hint" filter is fully activated so that the filter will run if the hardware and driver support it.

Using VPP filters follows the pseudocode below. Note that multiple VPPs can be set up in a pipeline.

```
mfxSession session;
MFXInit(MFX_IMPL_HARDWARE,[requested API level],&session);

//fill input parameters, attach extended parameter buffers (including
USE/DONOTUSE lists), zero unused fields
```



```
MFXVideoVPP_Query(session, in, out); /* check supported parameters */

//Check query output, adjust params if necessary

MFXVideoVPP_QueryIOSurf(session, &init_param, response);

//Allocate surfaces
MFXVideoVPP_Init(session, &init_param);

// run VPP for all frames in input while (frame_in)
{
   vpp->RunFrameVPPAsync(frame_in, frame_out, sync_v);
   SyncOperation(sync_v);
}

//drain loop
while (buffered data remains) {
vpp->RunFrameVPPAsync(NULL, frame_out, sync_v);
   SyncOperation(sync_v);
}

// Close components vpp->Close();
```

10.1 Color Format Conversion

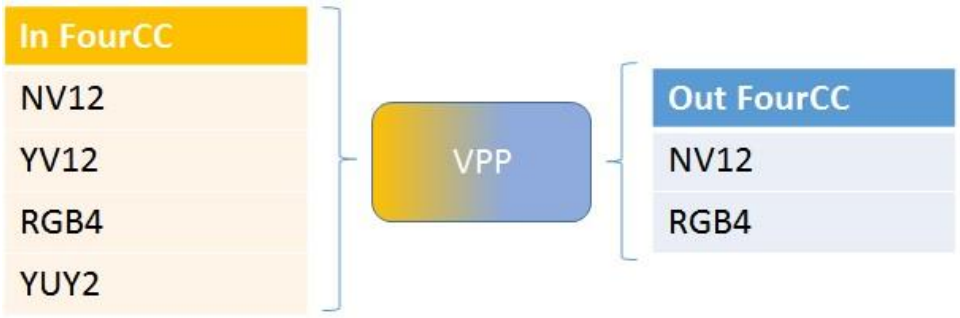
There are many formats for storing raw frame data. The SDK's default format is NV12. To help integrate with pipelines implemented in other raw frame color formats, VPP includes accelerated color format conversions. This is a deliberately small list. Additional format conversions can be implemented on the GPU using Intel® SDK for OpenCL™ Applications.

Color conversion is a mandatory filter. When VPP input and output FourCC parameters are different a color conversion will be performed.

Use of queries is recommended. If a conversion is not supported, VPP query will return -3 (MFX_ERR_UNSUPPORTED) or it will show a different input/output fourCC than selected by the application.

Inputs and Outputs

Accepted input and output formats are listed below.



Note: YV12 can be used to cover the most common YUV420 format, Y(W*H), U(.5W*.5H), V(.5W*.5H) with code set up as in the examples below.

(YUY2 is also supported as an output format in some cases. UYVY is supported as an input format in some cases.)

Evaluation in samples

You can start evaluating VPP color conversions with sample_vpp. For example:

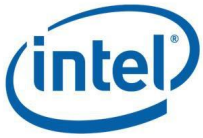
```
$ sample_vpp.exe -i crowdrun_960x544_nv12.yuv -sw 960 -sh 544 -scc nv12 -dcc
rgb4 -o out_bgra.yuv
$ ffplay -pix_fmt bgra -s 960x544 -i out_bgra.yuv
```

The -scc option for sample_vpp allows setting the source FourCC. Using -dcc allows setting a different output FourCC. In this case the conversion is from NV12 to RGB4. The RGB4 format corresponds to the bgra pixel format in FFmpeg.

The simple_4_vpp* and simple_6_* tutorials contain starting points to experiment with VPP color conversion. These examples are intended as a starting point to model your application's pipelines.

For convenience, here is a mapping between Media SDK FourCC formats and FFmpeg pixel formats.

FFmpeg pix_fmt	Media SDK FourCC
yuv420 (FFmpeg default)	MFX_FOURCC_YV12



nv12 (Media SDK default)	MFX_FOURCC_NV12
bgra	MFX_FOURCC_RGB4
yuyv422	MFX_FOURCC_YUY2

How to add to your application

The code snippets below have been tested with the tutorials. Creating a proof of concept first based on the tutorials is recommended.

I/O and surface configuration are the main things to keep in mind when using VPP color conversions. Surface sizes can be different between color formats, as well as data layout.

- Planar formats (NV12, YV12) store data in separate luma (intensity) and chroma (color) planes.
- Packed formats (RGB4, YUY2) store data for each pixel contiguously

Reading raw surfaces from files

While reading from raw files is not optimal for performance, using intermediate files is frequently used for debugging and developing interfaces. The code below should also help make the format layout clear.

Writing raw format surfaces to files can be accomplished by changing “fread” to “fwrite”.

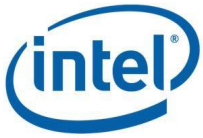
Reading NV12

```
mfxStatus LoadRawNV12Frame(mfxFrameSurface1* pSurface, FILE* fSource)
{
    //NV12 is a 12 bpp 4:2:0 planar format, Y(WxH),interleaved UV(W*.5H)
    size_t nBytesRead;
    mfxU16 w, h;
    mfxFrameInfo* pInfo = &pSurface->Info;
    mfxFrameData* pData = &pSurface->Data;

    w = pInfo->Width;
    h = pInfo->Height;


    // read luminance plane
    for (int i = 0; i < h; i++) {
        nBytesRead = (mfxU32) fread(pData->Y + i * pData->Pitch, 1, w,
fSource);
```

```
        if (w != nBytesRead)
            return MFX_ERR_MORE_DATA;
    }
    h /= 2;
    // read chroma plane
    for (int i = 0; i < h; i++) {
        nBytesRead = (mfxU32) fread(pData->UV + i * pData->Pitch, 1, w,
fSource);
        if (w != nBytesRead)
            return MFX_ERR_MORE_DATA;
    }
    return MFX_ERR_NONE;
}
```

Reading YV12 (this is the most commonly used form of YUV 4:2:0)

```
mfxStatus LoadRawI420Frame(mfxFrameSurface1* pSurface, FILE* fSource)
{
    //I420 is a 12 bpp 4:2:0 planar format, Y(WxH),U(.5W*.5H),V(.5W*.5H)
    //For VPP acceleration, treat as YV12 with U and V reversed

    size_t nBytesRead;
    mfxU16 w, h;
    mfxFrameInfo* pInfo = &pSurface->Info;
    mfxFrameData* pData = &pSurface->Data;

    w = pInfo->Width;
    h = pInfo->Height;


    // read Y (luminance) plane
    for (int i = 0; i < h; i++) {
        nBytesRead = (mfxU32) fread(pData->Y + i * pData->Pitch, 1, w,
fSource);
        if (w != nBytesRead)
            return MFX_ERR_MORE_DATA;
    }

    h /= 2;
    w /= 2;


    // read U(cb) plane
    for (int i = 0; i < h; i++) {
        nBytesRead = (mfxU32) fread(pData->U + i * pData->Pitch/2, 1, w,
fSource);
        if (w != nBytesRead)
            return MFX_ERR_MORE_DATA;
    }
```



```
    // read V(cr) plane
    for (int i = 0; i < h; i++) {
        nBytesRead = (mfxU32) fread(pData->V + i * pData->Pitch/2, 1, w,
fSource);
        if (w != nBytesRead)
            return MFX_ERR_MORE_DATA;
    }

    return MFX_ERR_NONE;
}
```

Reading RGB4

```
mfxStatus LoadRawRGB4Frame(mfxFrameSurface1* pSurface, FILE* fSource)
{
    //RGB4 is a packed 32 bpp format 4*W*H (BGRA)

    size_t nBytesRead;
    mfxU16 w, h;
    mfxFrameInfo* pInfo = &pSurface->Info;

    w = pInfo->Width;
    h = pInfo->Height;


    for (mfxU16 i = 0; i < h; i++) {
        nBytesRead = fread(pSurface->Data.B + i * pSurface->Data.Pitch,
            1, w * 4, fSource);
        if ((size_t)(w * 4) != nBytesRead)
            return MFX_ERR_MORE_DATA;
    }

    return MFX_ERR_NONE;
}
```

Reading YUY2

```
mfxStatus LoadRawYUY2Frame(mfxFrameSurface1* pSurface, FILE* fSource)
{
    //YUY2 is a packed 16 bpp format (Y0 U0 Y1 V0, Y2 U1 Y3 V1, ...)

    size_t nBytesRead;
    mfxU16 w, h;
    mfxFrameInfo* pInfo = &pSurface->Info;

    w = pInfo->Width;
    h = pInfo->Height;


    for (mfxU16 i = 0; i < h; i++) {
```

```
        nBytesRead = fread(pSurface->Data.Y + i * pSurface->Data.Pitch,
            1, w * 2, fSource);
        if ((size_t)(w * 2) != nBytesRead)
            return MFX_ERR_MORE_DATA;
    }

    return MFX_ERR_NONE;
}
```

Surface buffer layout

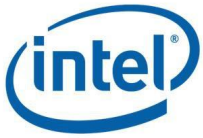
System memory allocations also need to explicitly match data layout expectations for the format used. While the examples below are for input surfaces, output surfaces can also be set up the same way.

NV12

```
bitsPerPixel = 12;        // NV12, YV12, and I420 formats are 12 bits per
pixel
surfaceSize = width * height * bitsPerPixel / 8;
surfaceBuffersIn = (mfxU8*) new mfxU8[surfaceSize * nVPPSurfNumIn];

pVPPSurfacesIn = new mfxFrameSurface1 *[nVPPSurfNumIn];
MSDK_CHECK_POINTER(pVPPSurfacesIn, MFX_ERR_MEMORY_ALLOC);

for (int i = 0; i < nVPPSurfNumIn; i++) {
    pVPPSurfacesIn[i] = new mfxFrameSurface1;
    memset(pVPPSurfacesIn[i], 0, sizeof(mfxFrameSurface1));
    memcpy(&(pVPPSurfacesIn[i]->Info), &(VPPParams.vpp.In),
sizeof(mfxFrameInfo));
    pVPPSurfacesIn[i]->Data.Y = &surfaceBuffersIn[surfaceSize * i];
    pVPPSurfacesIn[i]->Data.UV = pVPPSurfacesIn[i]->Data.Y + width * height;

    pVPPSurfacesIn[i]->Data.Pitch = width;
}
```

YV12

```
bitsPerPixel = 12;        // NV12, YV12, and I420 formats are 12 bits per
pixel
surfaceSize = width * height * bitsPerPixel / 8;
surfaceBuffersIn = (mfxU8*) new mfxU8[surfaceSize * nVPPSurfNumIn];

pVPPSurfacesIn = new mfxFrameSurface1 *[nVPPSurfNumIn];
```



```
MSDK_CHECK_POINTER(pVPPSurfacesIn, MFX_ERR_MEMORY_ALLOC);

for (int i = 0; i < nVPPSurfNumIn; i++) {
    pVPPSurfacesIn[i] = new mfxFrameSurface1;
    memset(pVPPSurfacesIn[i], 0, sizeof(mfxFrameSurface1));
    memcpy(&(pVPPSurfacesIn[i]->Info), &(VPPParams.vpp.In),
sizeof(mfxFrameInfo));
    pVPPSurfacesIn[i]->Data.Y = &surfaceBuffersIn[surfaceSize * i];
    pVPPSurfacesIn[i]->Data.V = pVPPSurfacesIn[i]->Data.Y + width * height;
    pVPPSurfacesIn[i]->Data.U =
pVPPSurfacesIn[i]->Data.V+(int)(width*height*.5);
    pVPPSurfacesIn[i]->Data.Pitch = width;
}
```

RGB4

```
bitsPerPixel = 32;        // RGB4 format is 32 bits per pixel
surfaceSize = width * height * bitsPerPixel / 8;
surfaceBuffersIn = (mfxU8*) new mfxU8[surfaceSize * nVPPSurfNumIn];

pVPPSurfacesIn = new mfxFrameSurface1 *[nVPPSurfNumIn];
MSDK_CHECK_POINTER(pVPPSurfacesIn, MFX_ERR_MEMORY_ALLOC);

for (int i = 0; i < nVPPSurfNumIn; i++) {
    pVPPSurfacesIn[i] = new mfxFrameSurface1;
    memset(pVPPSurfacesIn[i], 0, sizeof(mfxFrameSurface1));
    memcpy(&(pVPPSurfacesIn[i]->Info), &(VPPParams.vpp.In),
sizeof(mfxFrameInfo));
    pVPPSurfacesIn[i]->Data.B = &surfaceBuffersIn[surfaceSize * i];
    pVPPSurfacesIn[i]->Data.G = pVPPSurfacesIn[i]->Data.B + 1;
    pVPPSurfacesIn[i]->Data.R = pVPPSurfacesIn[i]->Data.B + 2;
    pVPPSurfacesIn[i]->Data.A = pVPPSurfacesIn[i]->Data.B + 3;
    pVPPSurfacesIn[i]->Data.Pitch = width*4;
}
```

YUY2

```
bitsPerPixel = 16;        // YUY2 format is 16 bits per pixel
surfaceSize = width * height * bitsPerPixel / 8;
surfaceBuffersIn = (mfxU8*) new mfxU8[surfaceSize * nVPPSurfNumIn];

pVPPSurfacesIn = new mfxFrameSurface1 *[nVPPSurfNumIn];
MSDK_CHECK_POINTER(pVPPSurfacesIn, MFX_ERR_MEMORY_ALLOC);

for (int i = 0; i < nVPPSurfNumIn; i++) {
```

```
    pVPPSurfacesIn[i] = new mfxFrameSurface1;
    memset(pVPPSurfacesIn[i], 0, sizeof(mfxFrameSurface1));
    memcpy(&(pVPPSurfacesIn[i]->Info), &(VPPParams.vpp.In),
sizeof(mfxFrameInfo));
    pVPPSurfacesIn[i]->Data.Y = &surfaceBuffersIn[surfaceSize * i];
    pVPPSurfacesIn[i]->Data.U = pVPPSurfacesIn[i]->Data.Y + 1;
    pVPPSurfacesIn[i]->Data.Pitch = width*2;
}
```

Additional info:

If the color format conversion needed is not on the list, it may already be implemented in Intel® Integrated Performance Primitives (IPP), OpenCV, or another package. For best performance, or for conversions not found elsewhere, you may want to consider writing format conversions in OpenCL.

Please watch for more on this topic in future versions of this guide.

For more information on raw surface formats:

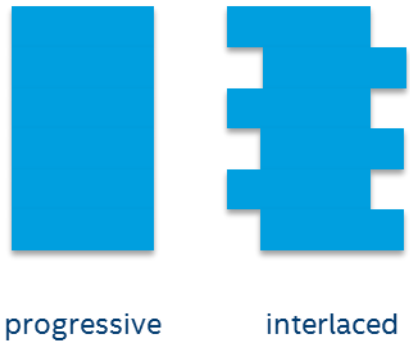
https://msdn.microsoft.com/en-us/library/windows/hardware/ff569028(v=vs.85).aspx

http://www.fourcc.org/yuv.php

10.2 Deinterlace

Interlaced video is stored as temporally offset fields. Progressive video is stored as frames, where all pixels are recorded at the same time. Deinterlacing means finding a way to merge the time disparities of the fields to convert to a progressive frame.

In the figure above, the circles on the left are an "end view" of image lines, offset to show the time differences in when they were recorded. This is a "top field first" (TFF) field layout. A "bottom field first" (BFF) layout is also possible. Basically, interlaced means that two frames of half resolution are presented together on the screen. This is fine when there is very little motion, but causes a characteristic "combing" effect due to capture time differences when motion occurs.



Two deinterlace modes are available. A simple bob/linear deinterlace is available for time constrained cases (for some, but not all, configurations – queries can be used to determine if bob is an option). An "advanced" deinterlace (ADI) is usually used for better quality, with the tradeoff that it can be a very heavy operation – in some cases taking longer than encode. The rest of this section focuses on ADI.

The ADI algorithm uses a 2-mode spatial-temporal approach:

- **Temporal**: for low motion areas. Assumes high temporal correlation between fields without need to consider motion vectors (interpolates even and odd fields)
- **Spatial**: for high motion areas. Higher interpolation weights for data within the temporal neighbor in the same field

From the VEBox Programmer's Reference Manual (https://01.org/sites/default/files/documentation/intel-gfx-prm-osrc-hsw-media-vebox_0.pdf)

"The motion adaptive deinterlacing is implemented by computing a measure of motion called the Spatial-Temporal Motion Measure (STMM). If this measure shows that there is little motion in an area around the current pixel, then the missing pixels/lines are filled in by averaging the pixel values from the previous and next fields. If the STMM shows that there is motion, then the missing pixels/lines are filled in by interpolating from spatially neighboring lines. The two results from TDI and SDI are alpha blended for intermediate values of STMM to prevent sudden transitions between TDI and SDI modes."

Evaluation in samples:

Deinterlace can be previewed using sample_multi_transcode and sample_vpp.

How to add to your application:

The most general option is to set VPP input picstruct to MFX_PICSTRUCT_UNKNOWN and output to MFX_PICSTRUCT_PROGRESSIVE. This allows interlaced/progressive to be determined from content, which also works with streams that mix interlaced and progressive. It also works with MBAFF input. However, TFF and BFF modes are available if this is known about the input.

```
VPPParams.vpp.In.PicStruct  = MFX_PICSTRUCT_FIELD_TFF;
VPPParams.vpp.Out.PicStruct = MFX_PICSTRUCT_PROGRESSIVE;
```

The mfxExtVPPDeinterlacing extended parameter buffer allows additional details about deinterlace mode and telecine.

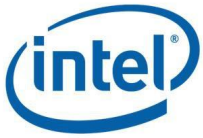
```
typedef struct {
 mfxExtBuffer Header;
 mfxU16 Mode;
 mfxU16 TelecinePattern;
 mfxU16 TelecineLocation;
 mfxU16 reserved[9];
} mfxExtVPPDeinterlacing;
```

While there are many deinterlace modes in the DeinterlacingMode enum, only three apply to standard VPP deinterlace. The rest are for PTIR.

MFX_DEINTERLACING_BOB	Use bob/linear (fast) DI, if available
MFX_DEINTERLACING_ADVANCED	Use ADI spatial-temporal high quality DI
MFX_DEINTERLACING_ADVANCED_NOREF	(API 1.17+) ADI without reference frames

10.3 Frame Rate Conversion

Frame rate is the expected playback rate in frames per second (FPS). This is separate from encode speed, which can be significantly higher for Media SDK. . There are many frame rates in common use, for example, NTSC uses 29.97 FPS and PAL is 25 FPS.



Frame rate conversion is a mandatory VPP filter. The application must specify the input and output frame rates using the FrameRateExtN and FrameRateExtD parameters in the mfxFrameInfo structure.

For example a 29.97 frame rate can be specified by

FrameRateExtN=2997

FrameRateExtD=100

If used in a pipeline with decode, these values can be read after DecodeHeader.

Please note that the Intel Media SDK implementation of frame rate conversion assumes constant frame rate. Do not use frame rate conversion for variable frame rate (VFR) input.

Algorithm

The basic idea behind frame rate conversion is to add or remove frames so that playback will still cover the same timeline at the new frame rate. When input and output are evenly divisible this is straightforward:

- 50 FPS to 25 FPS: drop every other frame, output frame count is ½ original
- 25 FPS to 50 FPS: insert a new frame between each frame, so output has 2x frames

However, for a general solution the approach must be more complex, based on frame cadence ratios. For example, when converting 60 FPS to 50 FPS every 6^{th} frame would be dropped, leaving $5/6^{th}$ frame count.

There are some special cases. For example, If VPP is initialized to perform deinterlacing with an input frame rate of 29.97 (NTSC) and an output frame rate of 23.976 (film/DVD/Blu-Ray), the Inverse Telecine algorithm is used.

Some combinations of hardware, OS, and driver version may attempt motion vector based frame interpolation for inserted frame content if the MFX_FRCALGM_FRAME_INTERPOLATION algorithm is selected. Commonly supported conversion ratios are 1:2 and 2:5 (useful for creating 60Hz content from 30Hz and 24Hz, respectfully). If the ratio of input to output frame rate is not supported, the VPP operation will report MFX_WRN_FILTER_SKIPPED at init or when queried. In practice this may not be a reliable indicator of whether the more sophisticated frame interpolation was used.

VPP frame rate conversion usually works by simply dropping or repeating frames.

More complex frame rate conversion approaches can be implemented as user plugins. For example, OpenCL could be used to create a simple weighted average based on frame cadence ratio (for framerate doubling, inserted frame is equal rate average of temporal neighbors, but weights can be adjusted for other ratios). For a more sophisticated approach, the OpenCL VME plugin could be used to create a history of motion vectors to reconstruct inserted frames without the blurring caused by simple weighted averaging.

Timestamp effects on frame insertion/frame dropping

The VPP frame rate conversion algorithm is based entirely on the in and out FrameRateExtN/ FrameRateExtD parameters.

Timestamps do not affect VPP frame insertion/deletion in any way. These are considered constant parameters. While the application could change output cadence, this would be external to VPP. For example:

1) To specify a new input or output framerate, reset/re-initialize VPP (may require reset of the entire pipeline)
2) Application handles frame dropping/multiple insertions of the same frame instead of VPP FRC

Timestamps and VPP FRC

Usually timestamps are simply forwarded from input to output through a Media SDK pipeline, with values unchanged and ignored. FRC and deinterlace are the only two cases where timestamp values might be changed.

For FRC, how timestamps are handled is specified by the FRCAlgm enum:

MFX_FRCALGM_FRAME_INTERPOLATION,

MFX_FRCALGM_PRESERVE_TIMESTAMP: frames keep their original timestamp values. Inserted frames have MFX_TIMESTAMP_UNKNOWN. The application can interpolate unknown timestamps as a later step. **Recommended.**

MFX_FRCALGM_DISTRIBUTED_TIMESTAMP: more on this mode in a future version of this guide.

Use in samples

Frame rate conversion can be found in sample_vpp and sample_multi_transcode.

How to add to your application

VPP FRC is triggered by differences in input and output frame rate. For example, the settings below double the number of frames.

```
VPPParams.vpp.In.FrameRateExtN = 30;
VPPParams.vpp.In.FrameRateExtD = 1;
…
VPPParams.vpp.Out.FrameRateExtN = 60;
VPPParams.vpp.Out.FrameRateExtD = 1;
```

In the main loop,

- VPP assumes a ready-to-process input surface
- "regular" output is ready to process after VPP returns MFX_ERR_NONE, but there are two cases to consider for VPP to handle increasing/decreasing the frame rate
    1) If the output framerate is less than the input framerate, this means frames need to be skipped. VPP will return MFX_ERR_MORE_DATA, which indicates that the application should read ahead in the input stream to the next frame. This will continue until the next "regular" frame which VPP should output.
    2) If the output framerate is greater than the input framerate, VPP will return MFX_ERR_MORE_SURFACE when an additional surface is required to hold an inserted frame. Output handling should proceed from this state as well as "regular" output when VPP returns MFX_ERR_NONE.

An example main loop:

```
boolean processing_file=true;
while (processing_file) {

    //load in a frame
    nSurfIdxIn = GetFreeSurfaceIndex(pVPPSurfacesIn, nVPPSurfNumIn);

    pVPPSurfacesIn[nSurfIdxIn]->Data.TimeStamp=
        nFrame*90000/VPPParams.vpp.Out.FrameRateExtN;
    sts = LoadRawFrame(pVPPSurfacesIn[nSurfIdxIn], fSource);
    if (sts!=MFX_ERR_NONE) {processing_file=false;}

    boolean processing_vpp=true;

    while (processing_vpp) {
        // Process a frame asychronously (returns immediately)
        sts = mfxVPP.RunFrameVPPAsync(pVPPSurfacesIn[nSurfIdxIn],
              pVPPSurfacesOut[nSurfIdxOut], NULL, &syncp);
```

```
            switch (sts)
            {
            case MFX_ERR_MORE_DATA:
                //skip a frame
                nSurfIdxIn = GetFreeSurfaceIndex(pVPPSurfacesIn,
nVPPSurfNumIn);
                MSDK_CHECK_ERROR(MFX_ERR_NOT_FOUND, nSurfIdxIn,
MFX_ERR_MEMORY_ALLOC);

                pVPPSurfacesIn[nSurfIdxIn]->Data.TimeStamp=
                  nFrame*90000/VPPParams.vpp.Out.FrameRateExtN;
                sts = LoadRawFrame(pVPPSurfacesIn[nSurfIdxIn], fSource);
                if (sts!=MFX_ERR_NONE) {processing_file=false;}
                break;
            case MFX_ERR_MORE_SURFACE:
                //add (often duplicate) a frame
                nSurfIdxOut = GetFreeSurfaceIndex(pVPPSurfacesOut,
                   nVPPSurfNumOut);     // Find free output frame surface

                sts = session.SyncOperation(syncp, 60000

                ++nFrame;

                sts = WriteRawFrame(pVPPSurfacesOut[nSurfIdxOut], fSink);
                MSDK_BREAK_ON_ERROR(sts);
                break;

            case MFX_WRN_DEVICE_BUSY:
                MSDK_SLEEP(1);
                break;
            default:
                processing_vpp=false;
                break;
            }
        }
        if (sts!=MFX_ERR_NONE) continue;

        sts = session.SyncOperation(syncp, 60000
        ++nFrame;

        sts = WriteRawFrame(pVPPSurfacesOut[nSurfIdxOut], fSink);
    }
```

(note: an additional drain loop with null inputs may be required to drain any buffered frames.)

10.4 Scaling

Scaling is one of the most frequently used VPP algorithms.

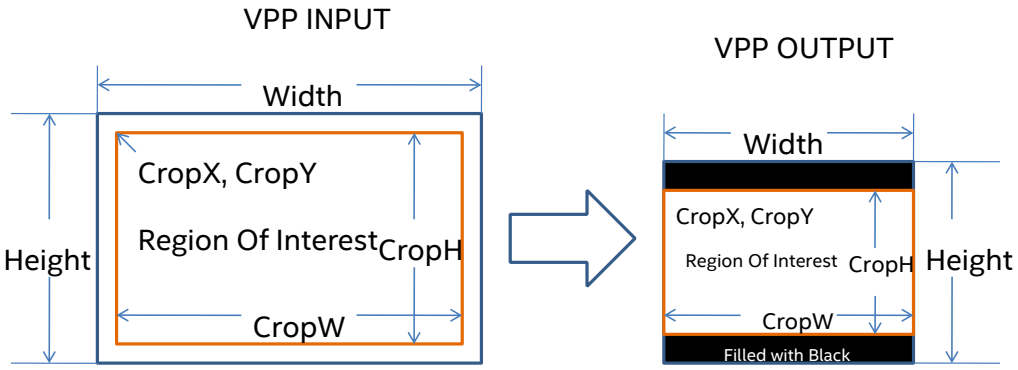
This filter can be used for several related operations.

Cropping

Resizing

Stretching

To perform scaling, define the region of interest using the CropX, CropY, CropW and CropH parameters to specify the input and output VPP parameters in the mfxVideoParam structure.



VPP resize capabilities can be combined to retain aspect ratio, for example with ROI extraction or letterbox/pillarbox conversions.

- Letterbox: add black bars above and below to pad unused lines
- Pillarbox: add black columns to the right and left to pad unused columns

Example settings:

	VPP Input		VPP Output	
Operation	**Width x Height**	**CropX, CropY, CropW, CropH**	**Width x Height**	**CropX, CropY, CropW, CropH**
Cropping	720x480	16,16,688,448	720x480	16,16,688,448

Resizing	720x480	0,0,720,480	1440x960	0,0,1440,960
Horizontal stretching	720x480	0,0,720,480	640x480	0,0,640,480
16:9 → 4:3 with letter boxing at the top and bottom	1920x1088	0,0,1920,1088	720x480	0,36,720,408
4:3 → 16:9 with pillar boxing at the left and right	720x480	0,0,720,480	1920x1088	144,0,1632,1088

Algorithm

The scaling application makes use of the hardware accelerated interpolation available on the sampler hardware. It also utilizes additional GPU capabilities to achieve quality similar to frequenly used convolution-based algorithms for downscaling.

Note: upscaling is supported, but algorithm is not designed for high quality upscaling. It may be possible to create better upscaling uptions in OpenCL.

Evaluation in samples

Sample_multi_transcode, sample_encode, and sample_vpp showcase resize.

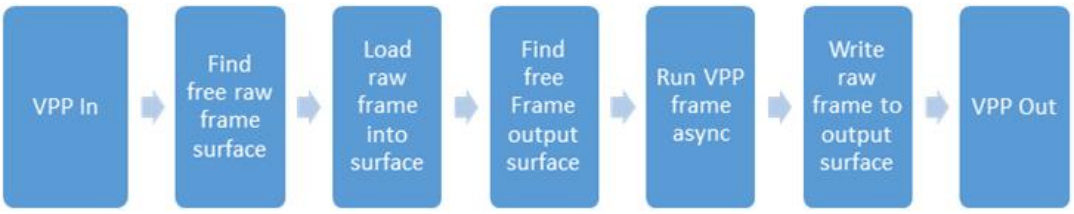
Use in applications

Resize is triggered when VPP Width, Height, or crop{X,Y,W,H} are different for in and out parameters:

```
    VPPParams.vpp.In.Width = 1920;
    VPPParams.vpp.In.Height = 1080;
…
    VPPParams.vpp.In.Width = 640;
    VPPParams.vpp.In.Height = 480;
```

Two different pools of surfaces are required for decode operations, since by definition the input and output frame sizes are different.

Below is the basic flow chart showing the pipeline of the functions being to achieve scaling-

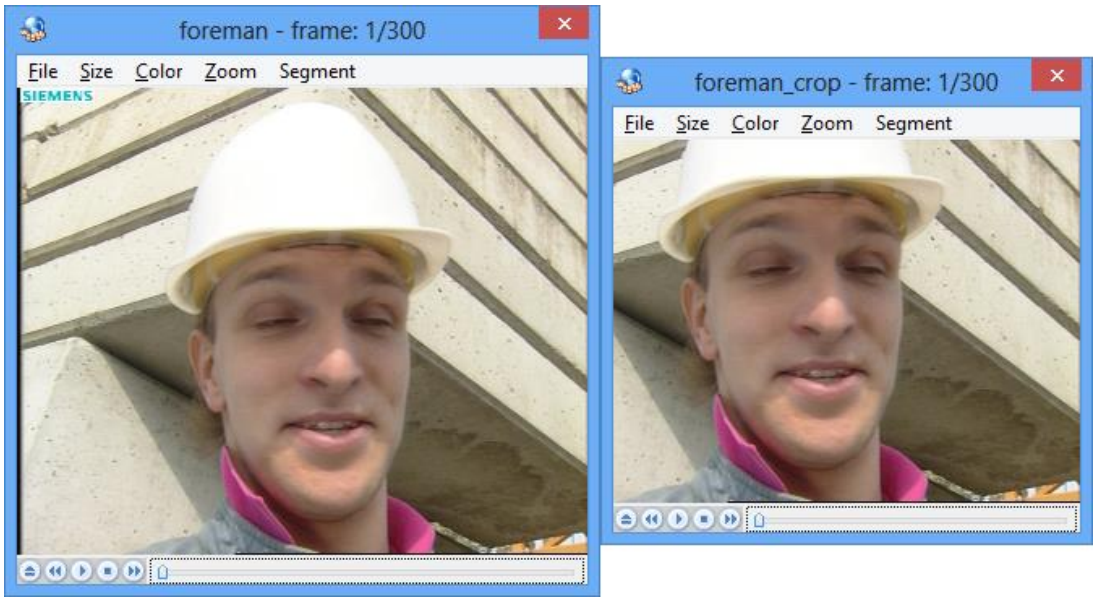


Cropping

This is one of the most commonly used video processing operation which is often used to define region of Interest (ROI). This can help you preserve or change aspect ratio as well . Most common changes are 16:9->4:3 and 4:3->16:9. Pillar and letter boxing can be used to achieve desired aspect ratio. Below is the table for the input options can be used to achieve cropping, letter boxing and pillar boxing in sample_vpp tutorial.

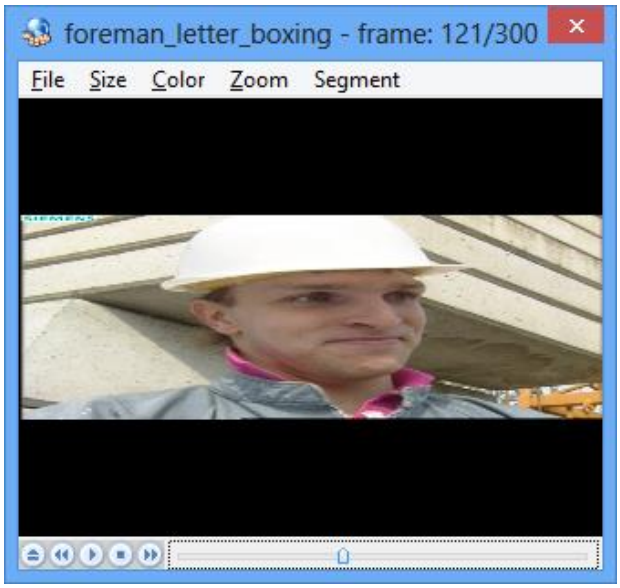
	CropX	CropY	CropW	CropH	Width	Height
Input	128	128	1024	464	1280	720
Output_Crop	0	0	1024	464	1024	464
Output_PillarBoxing	128	0	1024	720	1280	720
Output_LetterBoxing	0	128	1280	464	1280	720

Crop output:

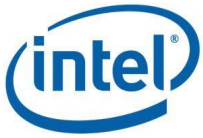


Pillar box output:

Letter box output:



10.5 Composition



Video Composition is a very useful feature for many video applications such as logo insertion, video wall, advertisements and any application that combines multiple streams.

Please note, it is available only for hardware sessions.

Media SDK can composite up-to 16 different video streams with one VPP session. Multiple VPPs can be chained together if composition of more streams is required. Inputs to composition can be dynamic content (changing each frame) or static (image/logo overlay). Blending features are also available.

Evaluate in samples

Composition can be used from sample_multi_transcode or sample_vpp.

Use in sample_vpp requires many parameters which can be provided with a par file.

Example par file

```
stream=/path/to/stream/in_352_288.yuv
width=352
height=288
cropx=0
cropy=0
cropw=352
croph=288
dstx=0
dsty=0
dstw=352
dsth=288
fourcc=nv12

stream=/path/to/stream/in_176_144.yuv
width=176
height=144
cropx=0
cropy=0
cropw=88
croph=72
dstx=0
dsty=0
dstw=88
dsth=72
fourcc=nv12
```

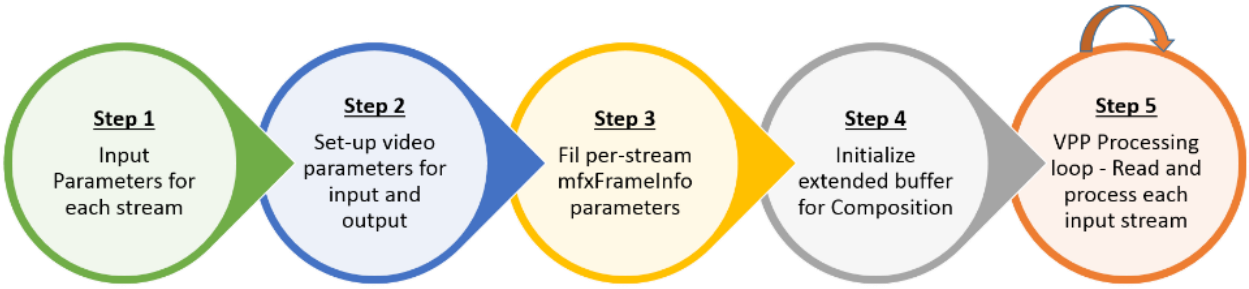
To run sample_vpp with the par file

```
$ sample_vpp -lib hw -scc nv12 -dcc nv12 -composite example.par -o out.yuv
```

Please note: both YUV420 inputs use nv12 color format.

Use in application

The SDK provides a mfxExtVPPComposite structure to provide parameters to composition, such as the resolution, size, placement coordinates on the destination surface, etc. for each individual stream.



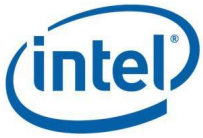
Step 1: Set up inputs

Open files, set up resolutions, etc.

Step 2: Set-up the video parameters in the application

Populate the video parameters for both input and output. For input VPP parameters, the details such as resolution and crop dimensions should correspond to the largest input stream.

```
/******
Initialize VPP parameters
******/
```



```
    mfxVideoParam VPPParams;
    memset(&VPPParams, 0, sizeof(VPPParams));
    // Input data
    VPPParams.vpp.In.FourCC = MFX_FOURCC_NV12;
    VPPParams.vpp.In.ChromaFormat = MFX_CHROMAFORMAT_YUV420;
    VPPParams.vpp.In.CropX = 0;
    VPPParams.vpp.In.CropY = 0;
    VPPParams.vpp.In.CropW = inputWidth;
    VPPParams.vpp.In.CropH = inputHeight;
    VPPParams.vpp.In.PicStruct = MFX_PICSTRUCT_PROGRESSIVE;
    VPPParams.vpp.In.FrameRateExtN = 30;
    VPPParams.vpp.In.FrameRateExtD = 1;
      // width must be a multiple of 16
      // height must be a multiple of 16 in case of frame picture and a
multiple of 32 in case of field picture
    VPPParams.vpp.In.Width = MSDK_ALIGN16(inputWidth);
    VPPParams.vpp.In.Height =
        (MFX_PICSTRUCT_PROGRESSIVE == VPPParams.vpp.In.PicStruct) ?
        MSDK_ALIGN16(inputHeight) :
        MSDK_ALIGN32(inputHeight);

    // Output data
    VPPParams.vpp.Out.FourCC = MFX_FOURCC_NV12;
    VPPParams.vpp.Out.ChromaFormat = MFX_CHROMAFORMAT_YUV420;
    VPPParams.vpp.Out.CropX = 0;
    VPPParams.vpp.Out.CropY = 0;
    VPPParams.vpp.Out.CropW = inputWidth;
    VPPParams.vpp.Out.CropH = inputHeight;
    VPPParams.vpp.Out.PicStruct = MFX_PICSTRUCT_PROGRESSIVE;
    VPPParams.vpp.Out.FrameRateExtN = 30;
    VPPParams.vpp.Out.FrameRateExtD = 1;
    // width must be a multiple of 16
    // height must be a multiple of 16 in case of frame picture and a
multiple of 32 in case of field picture
    VPPParams.vpp.Out.Width = MSDK_ALIGN16(VPPParams.vpp.Out.CropW);
    VPPParams.vpp.Out.Height =
        (MFX_PICSTRUCT_PROGRESSIVE == VPPParams.vpp.Out.PicStruct) ?
        MSDK_ALIGN16(VPPParams.vpp.Out.CropH) :
        MSDK_ALIGN32(VPPParams.vpp.Out.CropH);

    // Video memory surfaces are used to storing the raw frames. Use with HW
acceleration for better performance
    VPPParams.IOPattern = MFX_IOPATTERN_IN_VIDEO_MEMORY |
MFX_IOPATTERN_OUT_VIDEO_MEMORY;
```

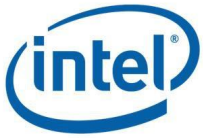
Step 3: Populate per-stream mfxFrameInfo with details from parameter file

In this step, we will specify the parameters specific to each input file such as resolution and crop dimensions. During the VPP processing loop when each stream is loaded, the surface info parameters are set to these parameters.

```
/*****************************************************************************
*********************
COMPOSITION-SPECIFIC BEGINS: Setting Phase

Crop the second stream to W/2,H/2 size starting at (0,0) co-ordinate. This
cropped stream will be composited onto the first stream at 0,0.
*****************************************************************************
********************/

    mfxU16 W1 = 352, H1 = 288;
    mfxU16 Cx1 = 0, Cy1 = 0, Cw1 = W1, Ch1 = H1;

    mfxU16 W2 = 176, H2 = 144;
    mfxU16 Cx2 = 0, Cy2 = 0, Cw2 = W2 >> 1, Ch2 = H2 >> 1;

    /** Fill frame params in mFrameInfo structures with the above parameters
**/
    for (mfxU16 i = 0; i < NUM_STREAMS; i++){
        memcpy(&inputStreams[i],&(VPPParams.vpp.In), sizeof(mfxFrameInfo));
        inputStreams[i].Width = i == 0 ? W1 : W2;
        inputStreams[i].Height = i == 0 ? H1 : H2;
        inputStreams[i].CropX = i == 0 ? Cx1 : Cx2;
        inputStreams[i].CropY = i == 0 ? Cy1 : Cy2;
        inputStreams[i].CropW = i == 0 ? Cw1 : Cw2;
        inputStreams[i].CropH = i == 0 ? Ch1 : Ch2;
    }
```

Step 4: Initialize extended buffer for Composition

```
// Initialize extended buffer for Composition
    mfxExtVPPComposite composite;
    memset(&composite, 0, sizeof(composite));
    composite.Header.BufferId = MFX_EXTBUFF_VPP_COMPOSITE;
    composite.Header.BufferSz = sizeof(mfxExtVPPComposite);
    composite.NumInputStream = 2;
    composite.Y = 10;
    composite.U = 80;
    composite.V = 80;
    composite.InputStream = (mfxVPPCompInputStream*) new
mfxVPPCompInputStream * [2];
```



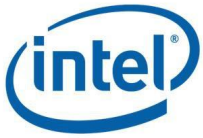
```
    composite.InputStream[0].DstX = (mfxU32)0;
    composite.InputStream[0].DstY = (mfxU32)0;
    composite.InputStream[0].DstW = (mfxU32)W1;
    composite.InputStream[0].DstH = (mfxU32)H1;

    composite.InputStream[1].DstX = (mfxU32)0;                //Co-ordinates
for where the second stream should go on the output surface
    composite.InputStream[1].DstY = (mfxU32)0;
    composite.InputStream[1].DstW = (mfxU32)Cw2;
    composite.InputStream[1].DstH = (mfxU32)Ch2;

    mfxExtBuffer* ExtBuffer[1];
    ExtBuffer[0] = (mfxExtBuffer*) &composite;
    VPPParams.NumExtParam = 1;
    VPPParams.ExtParam = (mfxExtBuffer**) &ExtBuffer[0];
```

Step 5: VPP Processing loop - Read & process each input stream

Note: VPP will return MFX_ERR_MORE_DATA for every stream to be composited.

```
// Stage 1: Main processing loop
//
while (MFX_ERR_NONE <= sts || MFX_ERR_MORE_DATA == sts) {
    nSurfIdxIn = GetFreeSurfaceIndex(pVPPSurfacesIn, nVPPSurfNumIn);
// Find free input frame surface
    MSDK_CHECK_ERROR(MFX_ERR_NOT_FOUND, nSurfIdxIn, MFX_ERR_MEMORY_ALLOC);

    // Surface locking required when read/write video surfaces
    sts = mfxAllocator.Lock(mfxAllocator.pthis, pVPPSurfacesIn[nSurfIdxIn]-
>Data.MemId, &(pVPPSurfacesIn[nSurfIdxIn]->Data));
    MSDK_BREAK_ON_ERROR(sts);


/*******************************************************************************
***********************************
    COMPOSITION-SPECIFIC CODE BEGINS:
    Loading data from each of the input streams, and
    Setting the surface parameters to the Crop, Width, Height, values of the
input stream being loaded

*******************************************************************************
***********************************/
    streamNum %= NUM_STREAMS;
    memcpy(&(pVPPSurfacesIn[nSurfIdxIn]->Info), &(inputStreams[streamNum]),
sizeof(mfxFrameInfo));
    sts = LoadRawFrame_YV12toNV12(pVPPSurfacesIn[nSurfIdxIn],
fSource[streamNum], &inputStreams[streamNum]);        // Load frame from file
into surface
    streamNum++;
    MSDK_BREAK_ON_ERROR(sts);
```

```
/*********************************************************************************
***********************************
    COMPOSITION-SPECIFIC CODE ENDS:

*********************************************************************************
***********************************/

    sts = mfxAllocator.Unlock(mfxAllocator.pthis, pVPPSurfacesIn[nSurfIdxIn]-
>Data.MemId, &(pVPPSurfacesIn[nSurfIdxIn]->Data));
    MSDK_BREAK_ON_ERROR(sts);

    nSurfIdxOut = GetFreeSurfaceIndex(pVPPSurfacesOut, nVPPSurfNumOut);
// Find free output frame surface
    MSDK_CHECK_ERROR(MFX_ERR_NOT_FOUND, nSurfIdxOut, MFX_ERR_MEMORY_ALLOC);

    for (;;) {
        // Process a frame asychronously (returns immediately)
        sts = mfxVPP.RunFrameVPPAsync(pVPPSurfacesIn[nSurfIdxIn],
pVPPSurfacesOut[nSurfIdxOut], NULL, &syncp);
        if (MFX_WRN_DEVICE_BUSY == sts) {
            MSDK_SLEEP(1);  // Wait if device is busy, then repeat the same
call
        } else
            break;
    }

    if (MFX_ERR_MORE_DATA == sts)   // Fetch more input surfaces for VPP
        continue;


    MSDK_BREAK_ON_ERROR(sts);

    sts = session.SyncOperation(syncp, 60000);      // Synchronize. Wait
until frame processing is ready
    MSDK_CHECK_RESULT(sts, MFX_ERR_NONE, sts);

    ++nFrame;
    // Surface locking required when read/write video surfaces
    sts = mfxAllocator.Lock(mfxAllocator.pthis, pVPPSurfacesOut[nSurfIdxOut]-
>Data.MemId, &(pVPPSurfacesOut[nSurfIdxOut]->Data));
    MSDK_BREAK_ON_ERROR(sts);

    sts = WriteRawFrame(pVPPSurfacesOut[nSurfIdxOut], fSink);
    MSDK_BREAK_ON_ERROR(sts);

    sts = mfxAllocator.Unlock(mfxAllocator.pthis,
pVPPSurfacesOut[nSurfIdxOut]->Data.MemId, &(pVPPSurfacesOut[nSurfIdxOut]-
>Data));
```



```
        MSDK_BREAK_ON_ERROR(sts);

        printf("Frame number: %d\n", nFrame);
        fflush(stdout);
    }
```

11 Concurrency

11.1 Handling concurrent Media SDK pipelines

Intel® Media SDK does not limit the number of concurrent sessions (regardless of the components used in each session). This means that an application can run several independent Media SDK sessions simultaneously in separate threads, processes, or applications. Individual session throughput will naturally be lower, compared to single session throughput since the sessions will share the resources. Please note that multisession performance gains are not linear. Higher end GPUs (GT3/GT4) contain multiple fixed function units, and concurrent encodes can make use of this hardware more efficiently than single encodes.

When executing several Media SDK sessions concurrently:

- If processing large number of streams with high resolution on a 32bit OS you may run into memory limitations. The error message provided by Media SDK when reaching memory limit is not very descriptive (due to underlying vague return codes from OS)
- If you encounter this limitation, please consider using 64 bit OS (greater pool of available graphics memory)

11.2 Thread Safety

Intel® Media SDK operations like decode, encode, and VPP are thread safe and are designed to be executed concurrently.

However, session initialization and video memory allocation are not thread safe and should only be executed serially -- one after the other in a single thread.

12 Tools

12.1 Debugging toolchest

Intel® Tools/SDKs

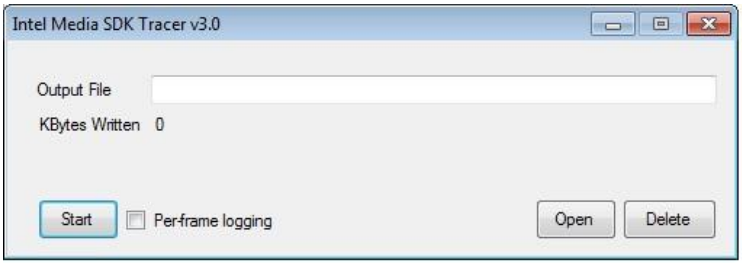
- Intel® Media SDK Tracer
- Intel® Media SDK System Analyzer
- Intel® VTune™ Amplifier
- Intel® Video Quality Caliper
- Intel® Video Pro Analyzer
- Intel® Media Server Studio – Metrics Monitor (Linux* only)

Useful tools

- TightVNC* (remote access + hardware acceleration)
- MediaInfo (media meta data details)
- ffmpeg (versatile media processing tool)
- ffprobe (stream/packet details, shows parsing/nonconformant stream errors)
- Process explorer (memory leaks, thread inspection, etc.)

12.2 MediaSDK_tracer tool

The MediaSDK_tracer tool is available in the tools/ folder of the SDK install path. It can be used to log Intel Media SDK activity to a file. Since this includes all parameters for Intel Media SDK sessions and pipeline stages, potentially for each frame, the output log for even a short run can be several megabytes. However, much of the power of this utility comes from its exhaustiveness – it captures a text representation of Intel Media SDK state which can be analyzed and compared to other runs. For more information, see the MediaSDK_tracer readme.

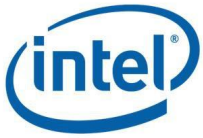


To use:

- Specify an output file. By default, logging for encode, decode, VPP, and sync is disabled. To enable, check per-frame logging. Note: the tracer tool appends to the log file.
- Press Start.
- The tool can capture from a running process, but it is often helpful to start the application after starting the trace tool.

12.3 MediaSDK_system_analyzer tool

The Media SDK "system_analyzer" tool is available on Windows only and located in the %INTELMEDIASDKROOT%\tools\mediasdk_sys_analyzer folder. The utility is used to analyze the developer platform, reporting back Media SDK related capabilities. The tool may help find environment, system configuration or driver installation issues that may prevent taking advantage of the full capabilities of the Intel Media SDK. For more information please refer to the documentation associated with the tool.



13 Appendix

13.1 Encoder Configuration for Blu-ray* and AVCHD*

This section describes how to configure the Intel® Media SDK library to encode H.264 and MPEG-2 video data in compliance with Blu-ray* or AVCHD* formats. For details on those formats, consult the following documents:

- AVCHD Format, Version 2.0, Book 1: Playback System Basic Specifications
- System Description Blu-ray Disc Read-Only Format, Part 3, Audio Visual Basic Specifications, Version 2.41, August 2010

The user may also need to consult other documents about Blu-ray and AVCHD.

Additionally, be aware that you may be required to have licenses from other companies (not Intel Corporation) to access Blu-ray or AVCHD specifications. This section provides information about the Intel Media SDK library and its data structures to help you comply with Blu-ray or AVCHD specifications.

13.1.1 Encoder Configuration

Configure SDK library encoders in the customary manner:

- Specify the Profile, Level, Width, Height, AspectRatio, ChromaFormat and other parameters in the mfxVideoParam structure;
- Set the parameter MaxDecFrameBuffering or others in the mfxExtCodingOption structure to improve encoder results.
- Explicitly specify the NumSlice value.

13.1.2 GOP Sequence

SDK encoders use a MPEG-2 style of GOP sequence control for both MPEG-2 and H.264 formats. Generate the desired GOP sequence by configuring:

- GopPicSize, GopRefDist, GopOptFlag and IdrInterval in the mfxVideoParam structure. The MFX_GOP_STRICT flag must be specified.
- ResetRefList in the mfxExtCodingOption structure.
- Applications can collect information about a desired GOP sequence and manually insert a GOP structure map into the bitstream with the following procedure:

Create a “fake” or dummy GOP structure map in accordance with the Blu-ray* specification;

Before a GOP sequence begins, insert the “fake” GOP structure map via payload insertion through the mfxEncodeCtrl structure;

Encode the GOP sequence. After encoding each frame in the sequence, the PicStruct member of the mfxBitstream structure reports the frame's picture type;

Replace the “fake” payload with the real, application-constructed GOP structure map.

13.1.3 SPS and PPS

SDK encoders write a single set of SPS and PPS NAL units to the output bitstream. Use EndofSequence, EndofStream, PicTimingSEI and AUDelimiter in the mfxExtCodingOption structure to configure how encoders write NAL units or SEI messages to the bitstream.

13.1.4 HRD Parameters

The Intel® Media SDK encoder supports HRD Type II compliance, but only with a single set of HRD parameters:

- Configurable parameters in the SDK related to HRD include RateControlMethod, InitialDelayInKB, BufferSizeInKB, TargetKbps, and MaxKbps in the mfxVideoParam structure.
- Additionally, the user can specify VuiNalHrdParameters in the mfxExtCodingOption structure to configure how the application writes HRD parameters to the bitstream.

13.1.5 Preprocessing Information

If the input pictures are preprocessed using 3:2 pull down, frame doubling or tripling, the application must convey this information through the PicStruct member of the mfxFrameSurface1 structure during encoding.

13.1.6 Closed-Captioned SEI Messages

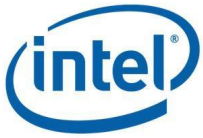
Applications can construct their own closed-captioned SEI messages (for H.264), or picture user data (for MPEG-2 video). Implement payload insertion through the mfxEncodeCtrl structure to put them into the output bitstream.

Please refer to “Inserting SEI message into encoded AVC stream” chapter for details on how to insert custom SEI message into H.264 streams. Can be used for closed captioning.

13.1.7 Unsupported Features

The Intel® Media SDK does not support:

- Decoder reference picture marking syntax; in this case, Blu-ray* requirements do not apply.
- Color matrix parameters, such as colour_primaries, transfer_characteristics or matrix_coefficients from the H.264 specification; the SDK does not write these into the bitstream.



- All NAL unit types or SEI messages; however, the SDK supports those needed by Bluray or AVCHD* and that are also defined in the H.264 specification.

13.2 Additional configuration for H.264 Stereo High Profile

The 3D video (MVC) encoding mode is enabled by the ViewOutput flag in mfxExtCodingOption structure; it instructs H.264 Encoder to output views in separate bitstream buffers and meet a set of bitstream packing conditions. Most significant of those conditions are listed below:

- Slice NAL units for each view are preceded by corresponding headers for each view (SPS, PPS, SEI).
- Prefix NAL unit with ID 0x14 is not inserted before the base view slice NAL units.
- View dependency representation delimiter (VDRD) NAL unit with ID 0x18 is inserted before header NAL units for the dependent view.
- If Buffering period and/or Picture timing SEI message(s) are present in base view component then corresponding dependent view component contains MVC scalable nested SEI with such SEI message(s). Dependent view conforms to HRD conformance with hrd_parameters() encoded in SPS/VUI inside Subset SPS and with mentioned BP, PT SEI messages encapsulated into MVC nested SEI

Applications should turn this mode on if they intend to produce H.264 Stereo High Profile data in compliance with Blu-ray* or AVCHD* formats.

13.2.1 Encoder Configuration for DVD-Video*

The Intel® Media SDK MPEG-2 encoder allows for producing DVD-Video*-compatible video streams. This section describes how to configure the Intel Media SDK library to encode MPEG-2 video data in compliance with DVD-Video format. For details on the format itself, consult the following documents:

- DVD Read Only Specifications Book, Part 3, DVD-Video Book (Book B)

The user may also need to consult other documents about DVD-Video. Additionally, be aware that you may be required to have licenses from other companies (not Intel Corporation) to access DVD-Video specifications. This section provides information about the Intel Media SDK library and its data structures to help you comply with DVD-Video specifications.

13.2.2 Encoder Configuration

Configure MPEG-2 video encoder in the customary manner:

Specify the Profile, Level, Width, Height, AspectRatio, ChromaFormat and other parameters in the mfxVideoParam structure. Note that the Intel® Media SDK accepts Pixel Aspect Ratio while applications likely operate with Display Aspect Ratio. An accurate conversion is needed.

For example for DAR = 4:3 PAR will be calculated as follows:

FrameInfo.AspectRatioW = 2304(4 * 576)

FrameInfo.AspectRatioH = 2160(3 * 720)

13.2.3 GOP Sequence

Specify the desired GOP sequence by configuring:

GopPicSize, GopRefDist, GopOptFlag and IdrInterval in the mfxVideoParam structure.

13.2.4 Setting NTSC/PAL video_format in the sequence_display_extension header

NTSC and PAL are analog encoding systems for broadcast television. Sometimes, applications require streams produced to specify the encoding system used. The Intel® Media SDK does not contain any NTSC or PAL encoding functionality. However, the encoder can write flags into the bitstream specifying one format or the other. The application is responsible for setting the encoding parameters consistent to the respective specification.

The data that identifies PAL or NTSC is stored in the video_format value of the sequence_display_extension header. Since this data is purely decorative to the Intel Media SDK, it is not directly accessible through the mfxVideoParam structure. Instead, the application can make the Encoder write this data into the stream by specifying a mfxExtCodingOptionSPSPPS structure with the required headers. This method allows the application to define custom SPS and PPS information in the stream.

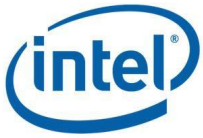
Please refer to “Custom control of encoded AVC SPS/PPS data” chapter for details on manual customizations to SPS/PPS data.

13.2.5 Preprocessing Information

If the input pictures are preprocessed—3:2 pull down, frame doubling or tripling—the application should convey such information through the PicStruct member of the mfxFrameSurface1 structure during encoding.

13.2.6 Closed Captioning

Applications can construct their own closed-captioned picture user data for MPEG-2 video. Implement payload insertion through the mfxEncodeCtrl structure to put them into the output bitstream.



13.3 Using Media SDK to build Video conferencing or streaming applications

Since API 1.3, we have extended support for video conferencing and streaming. The features listed below addresses common video conferencing and streaming requirements for improved adaptation to transmission conditions, robustness and realtime responsiveness:

- Low Latency Encode and Decode
- Dynamic Bit Rate Control
- Dynamic Resolution Control
- Forced Key Frame Generation
- Reference List Selection
- Reference Picture Marking Repetition SEI message
- Rolling I-Frame support
- Long Term Reference Frame (LTR)
- Temporal Scalability
- Motion JPEG (MJPEG) Decode

Please refer to the paper, http://software.intel.com/en-us/articles/video-conferencing-featureshttp://software.intel.com/en-us/articles/video-conferencing-features-of-intel-media-software-development-kit/of-intel-media-software-development-kit/, for details about using Media SDK in the video conferencing and streaming context.

Note: The white paper describes how to configure encoder for temporal scalability but not how the decoder application must interpret the encoded stream.

For temporal streams “nal_unit_header_svc_extension” is attached to each slice header, as described in standard (G.7.3.1.1). The headers contains “temporal_id”, which can be used to extract a certain temporal layer. Logic is simple: skip all slices with temporal_id > target_temporal_id and keep all slices with temporal_id <= target_temporal_id.

13.4 Handling Multiple Graphics Adapters

Some platforms that support Intel hardware acceleration may also include discrete graphics adapters. Desktop platforms may have 3rd party graphics adapters added by OEMs or end-users. Many OEM laptop system's also include a discrete card , and allow the end user (or even applications) to

dynamically choose which adapter is active. This is often referred to as “switchable graphics”. The increasing usage models can quickly lead to confusion when planning your application.

Until 2013, supporting hardware acceleration in a multi-gpu environment was bound by a fundamental constraint - the Intel Graphics adapter needed to have a monitor associated with the device to be active. This constraint was due to the capabilities of the Microsoft DirectX 9 infrastructure that the Intel Media SDK, and associated graphics driver were based upon. The introduction and corresponding support of the DirectX 11 in both the Intel Media SDK and graphics driver has now simplified the process of developing applications to utilize Intel’s fixed function hardware acceleration, even when the Intel graphics device is not connected to an external display device.

13.4.1 Multi-GPU and “headless” configurations

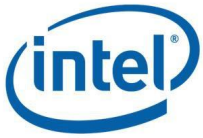
Applications wishing to leverage the Intel Media SDK’s hardware acceleration library when a discrete card is the primary device, or on devices without a monitor attached – such as “Session 0” modes, are required to initialize the Intel Media SDK to utilize the DirectX11 infrastructure, as well as provide its own memory allocation routines that manage DirectX 11 surfaces.

The following code illustrates the correct initialization parameters for initializing the library. MFX_IMPL_AUTO_ANY will scan the system for a supporting adapter, while MFX_IMPL_VIA_D3D11 indicates the need for DirectX 11 support. Similarly for VAAPI.

```
// Initialize Media SDK session
// - MFX_IMPL_AUTO_ANY selects HW accelaration if available (on any adapter)

mfxIMPL impl = MFX_IMPL_AUTO_ANY | MFX_IMPL_VIA_D3D11; mfxVersion ver = {0,
1}; MFXVideoSession mfxSession;

sts = mfxSession.Init(impl, &ver);
MSDK_CHECK_RESULT(sts, MFX_ERR_NONE, sts);
```

In addition to initializing the library, the application also needs to create the correct DirectX device context on the correct adapter. The following code is a simplistic illustration of how to enumerate the available graphics drivers on the system, and create the DirectX device on appropriate adapter. In this case the g_hAdapter handle is actually pointing to the Intel adapter which is in the secondary position.

```
mfxStatus CreateHWDevice(mfxSession session, mfxHDL* deviceHandle) {
    HRESULT hres = S_OK;

    static D3D_FEATURE_LEVEL FeatureLevels[] = {
        D3D_FEATURE_LEVEL_11_1,
        D3D_FEATURE_LEVEL_11_0,
        D3D_FEATURE_LEVEL_10_1,
```



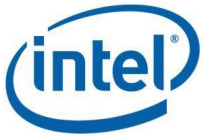
```
        D3D_FEATURE_LEVEL_10_0
    };
    D3D_FEATURE_LEVEL pFeatureLevelsOut;

    g_hAdapter = GetIntelDeviceAdapterHandle(session);
    if(NULL == g_hAdapter)
        return MFX_ERR_DEVICE_FAILED;

    hres =  D3D11CreateDevice(g_hAdapter,
                D3D_DRIVER_TYPE_UNKNOWN,
                NULL,
                0,
                   FeatureLevels,
                 (sizeof(FeatureLevels) /sizeof(FeatureLevels[0])),
                D3D11_SDK_VERSION,
                &g_pD3D11Device,
                    &pFeatureLevelsOut,
                &g_pD3D11Ctx);
    if (FAILED(hres))
        return MFX_ERR_DEVICE_FAILED;

    g_pDXGIDev     = g_pD3D11Device;    g_pDX11VideoDevice = g_pD3D11Device;
g_pVideoContext    = g_pD3D11Ctx;


 // turn on multithreading for the Context      CComQIPtr<ID3D10Multithread>
p_mt(g_pVideoContext);     if (p_mt)
        p_mt->SetMultithreadProtected(true);     else
        return MFX_ERR_DEVICE_FAILED;
    *deviceHandle = (mfxHDL)g_pD3D11Device;

    return MFX_ERR_NONE;
}
IDXGIAdapter* GetIntelDeviceAdapterHandle(mfxSession session)
{
    mfxU32 adapterNum = 0;
    mfxIMPL impl;

    MFXQueryIMPL(session, &impl);
```

```
    // Extract Media SDK base implementation type     mfxIMPL baseImpl =
MFX_IMPL_BASETYPE(impl);

   // get corresponding adapter number
    for (mfxU8 i = 0; i < sizeof(implTypes)/sizeof(implTypes[0]); i++)
    {
        if (implTypes[i].impl == baseImpl)
        {
            adapterNum = implTypes[i].adapterID;             break;
        }
    }

    HRESULT hres =
CreateDXGIFactory(__uuidof(IDXGIFactory2),(void**)(&g_pDXGIFactory) );     if
(FAILED(hres)) return NULL;

    IDXGIAdapter* adapter;
    hres = g_pDXGIFactory->EnumAdapters(adapterNum, &adapter);     if
(FAILED(hres)) return NULL;

   return adapter;
}
```

Finally, applications also need to ensure they are using the appropriate DirectX 11 routines for its memory allocator. Examples for this are available as part of the Media SDK sample code.

13.4.2 Mutliple-Monitor configurations

When a system supports display output to multiple monitors, each display is connected to the output of a graphics adapter. One graphics adapter may support multiple displays. For example, on many systems Intel's processor graphics adapter may be connected to an analog VGA* monitor and to an HDMI* monitor. The Windows* operating system will assign the primary (or "main") display as the default display device and use the associated adapter as the acceleration device if the application does not provide specific information about which device it wants to use. If the system contains one or more discrete graphics adapters in addition to the integrated graphics adapter, the user can set a display connected to the discrete adapter to be the primary display. In this case, applications that request the services of the default display adapter will not be able to take advantage of Intel's Quick Sync Video



acceleration. When an application creates an Media SDK session using the 'default' adapter, a call to MFXQueryIMPL() may not return a MFX_IMPL_HARDWARE status if the default adapter does not provide acceleration support for the MediaSDK API.

It is recommended that applications desiring hardware acceleration consider initializing their Media SDK session with MFX_IMPL_AUTO_ANY or MFX_IMPL_HARDWARE_ANY to allow all the capabilities of all the available graphics adapters to be examined, not just the 'default' adapter. When either of these options are used, calls to MFXQueryIMPL() will

return information about which device contains hardware acceleration. The application can use response of MFX_IMPL_HARDWARE2, MFX_IMPL_HARDWARE3 and

MFX_IMPL_HARDWARE4 to know which device should be used. The application should ensure that the creation of any Microsoft* Direct3D* device use the appropriate adapter, and not the default adapter, as the resources associated with the default adapter may not support acceleration.

13.4.3 Switchable Graphics configurations

Some platforms allow the user to dynamically select which graphics adapter is responsible for rendering to a single display for specific applications. For example, a user can select a game to execute on a discrete graphics adapter and a media application to execute on Intel's integrated graphics adapter. On these "switchable graphics" systems, the primary display adapter is changed to match the desired adapter for each application (process). The resources and capabilities of the inactive adapter are not available to applications, as the only adapter seen by the application is the current, active adapter.

From the application's point of view, the Media SDK operations may report

MFX_WRN_PARTIAL_ACCELERATION if a session is initialized for hardware acceleration but the currently configuration does not allow acceleration to be used.