OpenGL* Interoperability Tutorial

Sample User's Guide

Intel® SDK for OpenCL Applications - Samples*

Document Number:

Contents

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 SSSE3 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

About the OpenGL* Interop Tutorial

OpenGL* is a popular rendering API, while OpenCL™ is specifically crafted for the efficient computing. General interoperability scenario includes transferring data between the two APIs on a regular basis (often each frame) in both directions, for example:

- Physics simulation in OpenCL, producing the vertex data to be rendered with OpenGL
- Generating a frame with OpenGL, with further image post-processing applied in OpenCL (for example, HDR tone-mapping)
- Procedural (noise) generation in OpenCL, followed by using the results as OpenGL texture in the rendering pipeline

Since most OpenGL and OpenCL calls are not executed immediately, but instead placed in some command queues, a host logic to coordinate the resource ownership between OpenCL and OpenGL is required.
This tutorial gives an overview of basic methods for texture sharing and synchronization between the two APIs backed with performance numbers and recommendations. Finally, few advanced interoperability topics are also covered in this document along with some further references.

Introduction

Many programmers consider between different compute APIs for programming GPUs, for example between GLSL or OpenCL kernels. For short/simple tasks that intimately interact with graphics pipeline, the OpenGL Compute Shaders can be an API of choice. Yet for more general (and complex) scenarios the OpenCL computing might have advantages over GLSL, since it enables executing the compute portion in the asynchronous manner with respect to the rendering pipeline. Also OpenCL enables utilizing devices other than GPUs. Yet, unlike conventional “native” programming in C/C++, OpenCL enables leveraging devices other than CPUs as well, for example, DSPs.

Also notice that there are other APIs that can interoperate with OpenCLwell, which makes general application flow more efficient. For example, things like video transcoding, which are best handled with the Intel Media SDK. Unlike OpenGL, Intel OpenCL implementation offers zero-copy with Media SDK.

Still it is important to understand caveats and limitations for the interoperability, which we cover in details below.

General Execution Flow

Briefly, true (zero-copy) interoperability is about passing *ownership* and not the actual data of a resource between the APIs. It is important to keep in mind that it is OpenCL memory object created *from* OpenGL object, not vice versa:

- OpenGL texture (or render-buffer) becomes an OpenCL image (via `clCreateFromGLTexture`)

  OpenCL images are very similar to OpenGL textures, by means of supporting interpolation, border modes, normalized coordinates, and so on.

- OpenGL vertex buffers become OpenCL buffers in a similar way (`clCreateFromGLBuffer`);

This tutorial showcases three different approaches (next chapter) to utilize interoperability with OpenGL for the following general scenario: OpenCL memory object is created from the OpenGL texture.

- Every frame, the OpenCL memory object is updated with OpenCL kernel, providing the updated texture data to OpenGL after that.
- Every frame, OpenGL renders textured Screen-Quad to display the results.

Basic Approaches to the OpenGL-OpenCL™ Interoperability

This tutorial provides three different interoperability modes:

1. Direct OpenGL texture sharing via clCreateFromGLTexture
   - **This is the most performance-efficient way** when doing interoperability with Intel HD Graphics OpenCL device, which also enables modifing the texture "in-place".
   - Still, the number of OpenGL texture formats and targets that are possible to share via OpenCL *images* is limited. Refer to the dedicated chapter in the doc.
2. Creating an intermediate (staging) Pixel-Buffer-Object for the OpenGL texture via clCreateFromGLBuffer, updating the buffer with OpenCL and copying the results back to the texture
   - The downside of this approach is that even though the PBO itself does support zero-copy with OpenCL, the final PBO-to-texture transfer still relies on copying. So this method is slower than direct sharing.
   - The upside that there are **less restrictions on the texture formats and targets:** you can use formats and targets beyond that the `glTexSubImage2D` imposes.
3. Mapping of the GL texture with glMapBuffer, wrapping the resulting host pointer as OpenCL buffer for processing and copying the results back upon `glUnmapBuffer`
   - Similarly to the approach based on PBO, this approach enables interoperability for the virtually any texture that is possible to update with `glTexSubImage2D`.
   - Yet unlike the original PBO-based method it **does not require any extension support**.
   - Performance-wise, it is even slower than PBO-based method (particularly on the small textures), since OpenCL buffer is created/released each time - for every frame, example.

Notice that performance-wise the interoperability approach based on the direct OpenGL texture sharing via clCreateFromGLTexture is the fastest way to share data with Intel Graphics OpenCL device, while mapping of the GL texture with glMapBuffer is the slowest. Yet for the CPU OpenCL device, the performance picture is inversed.

The tutorial does not cover interoperability with respect to OpenGL vertex buffers in this tutorial, but these are conceptually similar to Pixel-Buffer-Objects: the approach is based on using `clCreateFromGLBuffer` for zero-copy sharing.

Also, notice that plain data transfers from OpenGL to the host memory (e.g. mapping) and then to the OpenCL memory space and back (upon unmapping) is the most straightforward method that assumes neither extension usage nor actual sharing. As we just discuss this method allows the most general interoperability, while any copying overheads (but not a power sipping) can be hidden with multi-buffering approach (which is not covered in the tutorial). Refer to the details of the general asynchronous transfer to/from OpenGL in the (last) section on the further reading. In order to be more performance/power efficient than a plain memory copy, an OpenCL implementation supporting `cl_khr_gl_sharing` is required.

cl_khr_gl_sharing extension

OpenCL-OpenGL interoperability is implemented as a Khronos extensions to OpenCL (below we refer to the extensions spec version 1.2). Therefore, the name of this extension (`cl_khr_gl_sharing`) should be listed in the list of supported extensions queried for the platform and the device, see OpenCL 1.2 specification section Table 4.3.

The interfaces (API) for this extension are provided in the `cl_gl.h` header.

Just as with other vendor extension APIs, the `clGetExtensionFunctionAddressForPlatform` function should be used to get pointers to the actual functions of the specific OpenCL platform. For example:

`clGetGLContextInfoKHR` declaration in the `cl_gl.h`:

```
/* declared in the cl_gl.h*/
extern CL_API_ENTRY cl_int CL_API_CALL
clGetGLContextInfoKHR(const cl_context_properties * /* properties */,
                      cl_gl_context_info            /* param_name */,
                      size_t                        /* param_value_size */,
                      void *                        /* param_value */,
                      size_t *                      /* param_value_size_ret */)
CL_API_SUFFIX__VERSION_1_0;

typedef CL_API_ENTRY cl_int (CL_API_CALL *clGetGLContextInfoKHR_fn)(
    const cl_context_properties * properties,
    cl_gl_context_info            param_name,
    size_t                        param_value_size,
    void *                        param_value,
    size_t *                      param_value_size_ret);
/*\declared in the cl_gl.h*/
```

In your code:

```
#include <CL\cl_gl.h>
//getting the pointer for the actual function for the specified platfrom
…
clGetGLContextInfoKHR_fn pclGetGLContextInfoKHR =
            (clGetGLContextInfoKHR_fn)
            clGetExtensionFunctionAddressForPlatform(intel_platform,
            "clGetGLContextInfoKHR");
//calling the function via its pointer
pclGetGLContextInfoKHR(…);
```

Here is the full list of functions for the extension and a short description for each:

`clGetGLContextInfoKHR`	Queries the devices associated with the OpenGL context
`clCreateFromGLBuffer`	Creates an OpenCL buffer object from the OpenGL buffer object
`clCreateFromGLTexture`	Creates an OpenCL image object from the OpenGL texture object
`clCreateFromGLRenderbuffer`	Creates an OpenCL 2D image object from the OpenGL renderbuffer
`clGetGLObjectInfo`	Queries type and name for the OpenGL object used to create the OpenCL memory object
`clGetGLTextureInfo`	Gets additional information (target and mipmap level) about the GL texture object associated with a memory object
`clEnqueueAcquireGLObjects`	Acquires OpenCL memory objects from OpenGL
`clEnqueueReleaseGLObjects`	Releases OpenCL memory objects to OpenGL

Notes on Textures Formats and Targets

It is important to understand that the number of OpenGL *texture* formats and targets that are possible to share via OpenCL *images*, is limited. You can find the full information on this topic here: https://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/clCreateFromGLTexture.html

The following OpenGL texture usages (“targets”) are available for sharing:

- OpenCL (1D, 2D, 3D) image object created from a regular OpenGL (1D, 2D,3D) texture
- 2D OpenCL image can be created from a single face of an OpenGL cubemap texture
- OpenCL 1D or 2D image array can be created from the OpenGL 1D or 2D texture array

Refer to the Table 9.4 in the OpenCL 1.2 extension spec that describes the list of GL texture internal formats and the corresponding image formats in OpenCL. These are most important *four-channel* formats like `GL_RGBA8` or `GL_RGBA32F`, for which the mapping is guaranteed.
Textures created with other OpenGL internal formats may have a mapping to a CL image format. So if such mappings exist (which is implementation-specific) the `clCreateFromGLTexture` succeeds, otherwise it fails with `CL_INVALID_IMAGE_FORMAT_DESCRIPTOR`. Notice that single-channel textures are generally not supported for sharing.

Also notice that OpenGL depth (depth-stencil) buffers sharing is subject for separate `cl_khr_depth_images` and `cl_khr_gl_depth_images` extensions, which we do not cover in this tutorial.
Finally, multi-sampled (MSAA) textures, both color and depth are subject for `cl_khr_gl_msaa_sharing` (which requires both `cl_khr_depth_images` and `cl_khr_gl_depth_images` support from the implementation).

Creating the Interoperability-Capable OpenCL Context

Since OpenCL memory objects are created *from* OpenGL objects, you need some sort of the shared OpenCL-OpenGL context. To avoid implicit copying via host, create OpenCL context for the same underlying device that drives the OpenGL context as well.

You can enumerate all OpenCL device(s) capable of sharing with the OpenGL context you are willing to interoperate, via `clGetGLContextInfoKHR`. First, you need to setup additional context parameters:

```
//Additional attributes to OpenCL context creation
//which associate an OpenGL context with the OpenCL context
cl_context_properties props[] =
 {
//OpenCL platform
      CL_CONTEXT_PLATFORM, (cl_context_properties)   platform,
//OpenGL context
      CL_GL_CONTEXT_KHR,   (cl_context_properties)   hRC,
//HDC used to create the OpenGL context
      CL_WGL_HDC_KHR,     (cl_context_properties)    hDC,
      0
    };
```

For the fastest interoperability, select a device currently *associated* with the given OpenGL context (`CL_`**`CURRENT_DEVICE`**`_FOR_GL_CONTEXT_KHR` flag for the `clGetGLContextInfoKHR`).

Notice that there are many more OpenCL devices that can potentially share the data with the OpenGL context (for example, via copies). The code example below uses `clGetGLContextInfoKHR` with `CL_`**`DEVICES`**`_FOR_GL_CONTEXT_KHR` to enumerate *all* interoperable devices:

```
size_t bytes = 0;
// Notice that extension functions are accessed via pointers
// initialized with clGetExtensionFunctionAddressForPlatform (previous section).

// queuing how much bytes we need to read
clGetGLContextInfoKHR(props, CL_DEVICES_FOR_GL_CONTEXT_KHR, 0, NULL, &bytes);
// allocating the mem
size_t devNum = bytes/sizeof(cl_device_id);
std::vector<cl_device_id> devs (devNum);
//reading the info
clGetGLContextInfoKHR(props, CL_DEVICES_FOR_GL_CONTEXT_KHR, bytes, devs, NULL));
//looping over all devices
for(size_t i=0;i<devNum; i++)
{
      //enumerating the devices for the type, names, CL_DEVICE_EXTENSIONS, etc
      clGetDeviceInfo(devs[i],CL_DEVICE_TYPE, …);
      …
      clGetDeviceInfo(devs[i],CL_DEVICE_EXTENSIONS,…);
      …
}
```

The tutorial supports selecting the platform and device to run (refer to the section on the controlling the sample), so upon enumerating the available devices for the requested platform the "OpenGL-shared" OpenCL context is created, along with a queue for the selected device:

```
context = clCreateContext(props,1,&device,0,0,NULL);
queue = clCreateCommandQueue(context,device,CL_QUEUE_PROFILING_ENABLE,NULL);
```

Texture Sharing via clCreateFromGLTexture

This is the most efficient method to enable direct OpenGL-texture to OpenCL-image sharing via `clCreateFromGLTexture`. This approach also enables modifying the content of the texture in-place. Below is the required sequence of steps:

1. Create OpenGL 2D texture in the regular way, for example:
```
//generate the texture ID
glGenTextures(1, &texture));
//bdinding the texture and setting the
glBindTexture(GL_TEXTURE_2D, texture));
//regular sampler params
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE));
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE));
//need to set GL_NEAREST
//(and not GL_NEAREST_MIPMAP_* which results in CL_INVALID_GL_OBJECT later)
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST));
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST));
//specify texture dimensions, format etc
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, g_width, g_height, 0, GL_RGBA,
GL_UNSIGNED_BYTE, 0);
```
2. Create the OpenCL image corresponding to the texture (once):
```
cl_mem mem = clCreateFromGLTexture(context, CL_MEM_WRITE_ONLY,
GL_TEXTURE_2D, 0,texture,NULL);
```
notice the `CL_MEM_WRITE_ONLY` flag that enables fast discarding of the data, use `CL_MEM_READ_WRITE` if your kernel requires reading the current texture context. Also remove the `__write_only` qualifier for the image read access in the kernel.
3. Acquire the ownership via `clEnqueueAcquireGLObjects`:
```
glFinish();
clEnqueueAcquireGLObjects(queue, 1,  &mem, 0, 0, NULL));
```
4. Execute the OpenCL kernel that alters the image:
```
clSetKernelArg(kernel_image, 0, sizeof(mem), &mem);
…
clEnqueueNDRangeKernel(queue,kernel_image, …);
```
5. Releasing the ownership via `clEnqueueReleaseGLObjects`:
```
clFinish(queue);
clEnqueueReleaseGLObjects(queue, 1,  &mem, 0, 0, NULL));
```

In this approach the relation between OpenCL image and OpenGL texture is specified just once and only acquire/release calls are used to pass ownership of the resources between APIs (thus providing zero-copy goodness for the actual texture data).
However, the number of texture formats and usages, for which this sharing is possible, is rather limited (refer to the dedicated section).

Texture Sharing via Pixel-Buffer-Object and clCreateFromGLBuffer

This method relies on the intermediate (staging) Pixel-Buffer-Object (PBO). It is less efficient than direct sharing (previous chapter) due to the final copying of the PBO bits to the texture. Yet it potentially enables sharing textures of more formats (refer to the formats that `glTexSubImage2D` supports), unlike the limited set supported by `clCreateFromGLTexture` (refer to the section on texture formats in this doc). The code sequence is like follows:

1. Create an OpenGL 2D texture in a regular way (refer to the first step in the previous section)

2. Create the OpenGL Pixel-Buffer-Object (once):
```
GLuint pbo;
glGenBuffers(1, &pbo);
glBindBuffer(GL_ARRAY_BUFFER, pbo);
//specifying the buffer size
glBufferData(GL_ARRAY_BUFFER, width * height * sizeof(cl_uchar4), …);
```
3. Create the OpenCL buffer corresponding to the Pixel-Buffer-Object (once):
```
mem = clCreateFromGLBuffer(g_context, CL_MEM_WRITE_ONLY, pbo,  NULL);
```
Notice the `CL_MEM_WRITE_ONLY` flag as buffer contains no original texture data, so no point to made it readable.
4. Acquire ownership via `clEnqueueAcquireGLObjects`, execute the kernel that updates the buffer content, and release the ownership via `clEnqueueReleaseGLObjects`: these steps are the same as steps 3-5 of the previous chapter (only kernel itself is slightly different, as it operates on OpenCL buffer, not image);
5. Finally, stream data from the PBO to the texture is required:
```
glBindBuffer(GL_PIXEL_UNPACK_BUFFER, pbo);
glBindTexture(GL_TEXTURE_2D, texture);
glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, width, height, GL_RGBA,
GL_UNSIGNED_BYTE, NULL);
```

Texture Sharing with glMapBuffer

This method is similar to the previous approach (PBO-based), but instead of relying on the `clCreateFromGLBuffer` to share the PBO with OpenCL it performs straightforward mapping of the PBO to the host memory instead (so it does not require any extension to support):

1. Create texture and PBO (refer to the first steps in the previous section)
2. Map the PBO bits to the host memory:
```
void* p = glMapBuffer(GL_PIXEL_UNPACK_BUFFER, GL_READ_WRITE));
```
3. The resulting pointer is wrapped with the OpenCL buffer using the `CL_MEM_USE_HOST_PTR` to avoid copy (yet the buffer is created/destroyed each frame)
```
cl_mem mem =
clCreateBuffer(g_context,CL_MEM_WRITE_ONLY|CL_MEM_USE_HOST_PTR,
width*height*sizeof(cl_uchar4), p, NULL);
```
4. OpenCL kernel that alters the buffer content (changing values to green) is called
```
clSetKernelArg(kernel_buffer, 0, sizeof(mem), &mem);
…
clEnqueueNDRangeKernel(queue,kernel_buffer, …);
clFinish(queue);
```
5. Upon the kernel completion, the buffer bits are copied back with `glUnmapBuffer`; also notice that calls to `clEnqueueMapBuffer`/`clEnqueueUnmapMemObject` are needed to make sure the actual buffer memory behind the mapped pointer is updated as (discrete) GPUs might mirror a buffer instead and perform an actual update (copy) upon `clEnqueueUnmapMemObject`.
6. Release the OpenCL buffer
```
clReleaseMemObject(mem);
```
7. The rest is the same to the last (fifth) step of the previous approach (PBO bits are copied to the texture with `glTexSubImage2D`).

Synchronization

In order to maintain data integrity, it is the application logic responsibility to synchronize access to shared OpenCL/OpenGL objects.

Specifically, prior to calling `clEnqueueAcquireGLObjects`, the application must ensure that any pending OpenGL operations that access the objects specified in `mem_objects` have completed (in all OpenGL contexts). Note that **no** synchronization methods other than `glFinish` are portable between OpenGL implementations at this time, so this tutorial relies on this mechanism.

Similarly, prior to executing the subsequent OpenGL commands that reference the released objects (after `clEnqueueReleaseGLObjects`), the application is responsible for ensuring that any pending OpenCL operations which access the objects have completed. The most portable way is calling `clWaitForEvents` with the event object returned by `clEnqueueReleaseGLObjects`, or by calling `clFinish` (as we do in this tutorial).

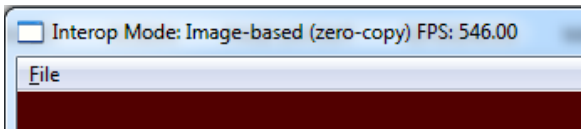
There is a more fine-grained way provided by the `cl_khr_gl_event` extension that enables creating OpenCL event objects from the OpenGL fence object. The OpenGL fence can be placed in the OpenGL command stream, while enabling to wait for completion of that fence in the CL command queue. The complimentary `GL_ARB_cl_event` extension in OpenGL provides the way of creating an OpenGL sync object from an OpenCL event.

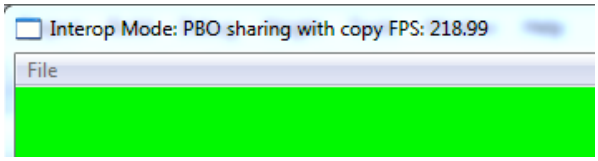
More importantly, supporting the `cl_khr_gl_event` guarantees that the OpenCL implementation will ensure that any pending OpenGL operations are complete for the OpenGL context upon calling the `clEnqueueAcquireGLObjects` in the OpenCL context. Similarly, `clEnqueueReleaseGLObjects` guarantees the OpenCL is done with the objects, so no explicit `clFinish` is required. This is referred to as implicit synchronization.

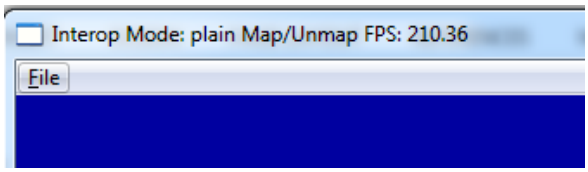
The tutorial checks for the extension support and omits calls to `clFinish()/glFinish()`, if the `cl_khr_gl_event` support is presented for the selected device.

Sample Output and Controls

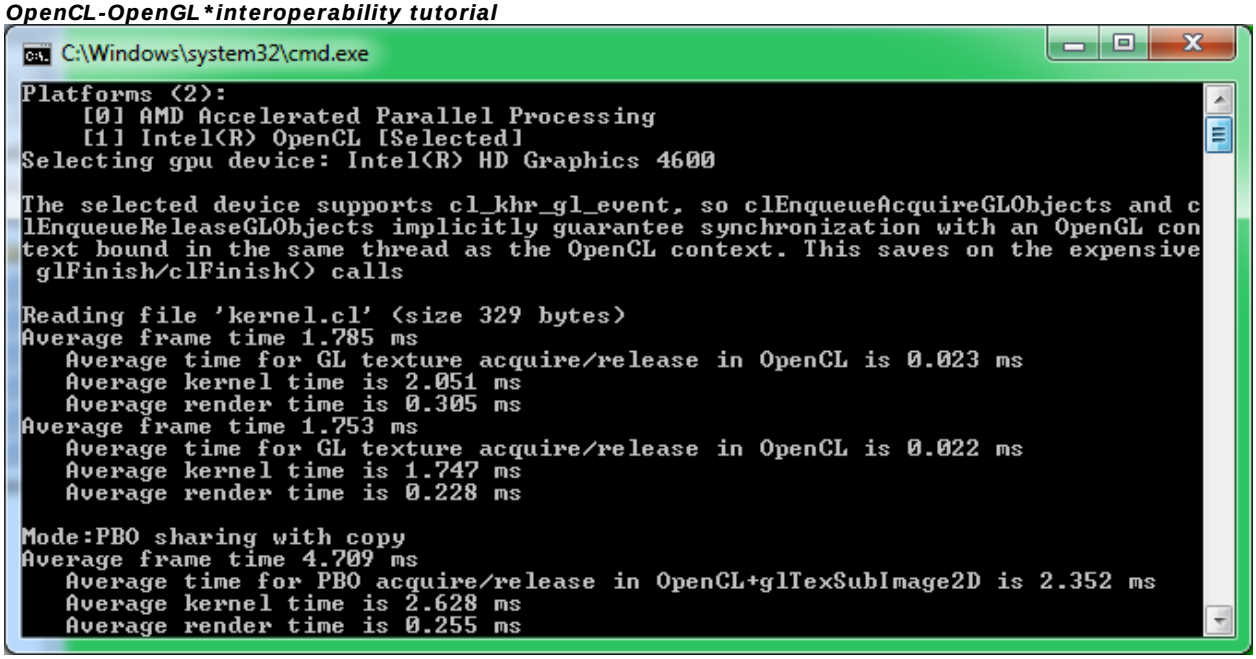
Use “Tab” key to switch between interoperability modes described in the previous sections. The texture color indicates the current interoperability mode, and title of the window is also updated accordingly. The sustained performance (in *overall* FPS) is also reported:







Notice that more detailed performance statistics on each stage of the pipeline like acquiring the data from OpenGL, updating the data, and rendering the results, is printed in the console output. For example:



You can run the sample application with the following command-line options:

Command-Line Options		
Short Form	**Full Form**	**Description**
`-h`	`--help`	Shows command-line options with descriptions.
`-p`	`--platform`	Selects OpenCL platform by ID (0 to n-1, where n is the number of available platforms).
`-t`	`--type`	Selects an OpenCL device to run by *type*. First device of the specified type will be picked. The following options are available: - cpu - gpu Combine the -t option with the -p option, which specifies OpenCL platform.
`-d`	`--device number-or-string`	Selects an OpenCL device by *number* (or name). This option combines well with the previous ones. For example, if you have multiple devices of the same *type* specified with –t, you can select the particular device to run using -d.

More Resources

Details of the cl_khr_gl_sharing extension

http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/cl_khr_gl_sharing.html

OpenCL-OpenGL*interoperability tutorial
OpenGL Insights (Asynchronous Buffer Transfers chapter)

http://www.seas.upenn.edu/~pcozzi/OpenGLInsights/OpenGLInsights-AsynchronousBufferTransfers.pdf