OpenCL™ and OpenGL* Interoperability Tutorial

By Maxim Shevtsov (Intel) (https://software.intel.com/en-us/user/456085), Updated April 28, 2014 Translate

OpenCL™ and OpenGL* are two common APIs that support efficient interoperability. OpenCL is specifically crafted to increase computing efficiency across platforms, and OpenGL is a popular graphics API. This tutorial provides an overview of basic methods for resource-sharing and synchronization between these two APIs, supported by performance numbers and recommendations. A few advanced interoperability topics are also introduced, along with references.

Introduction

A general interoperability scenario includes transferring data between the OpenCL and OpenGL on a regular basis (often per frame) and 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 (e.g. HDR tone-mapping)
- Procedural (noise) generation in OpenCL, followed by using the results as OpenGL texture in the rendering pipeline

Programmers often must choose between different APIs for programming GPUs, for example, choosing either GLSL or OpenCL kernels. For short, simple tasks that interact directly with the graphics pipeline, OpenGL Compute Shaders can be a good choice as an API. For more general (and complex) scenarios, OpenCL computing might have advantages over GLSL, since it executes the compute portion asynchronously to the rendering pipeline. Also, OpenCL allows you to utilize other devices than just GPUs. Yet, unlike conventional “native” programming in C/C++, OpenCL allows you to leverage devices other than CPUs, such as DSPs.

OpenCL interoperability also works with other APIs, too, which makes general applications flow more efficiently. A good example of this interoperability is video transcoding, which is best handled with the Intel Media SDK. Unlike OpenGL, Intel OpenCL implementation offers zero-copy with Media SDK.

General Execution Flow

It is important to understand the parameters and limitations of the interoperability, which are covered in this tutorial. Since most OpenCL and OpenGL calls are not executed immediately but are placed in command queues, host logic is required to coordinate the resource ownership between OpenCL and OpenGL.

True (zero-copy) interoperability is about passing ownership between the APIs and not the actual data of a resource. It is important to remember that it is an OpenCL memory object created from an OpenGL object and 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, etc.
- OpenGL (vertex) buffers are transformed to OpenCL buffers in a similar way (**clCreateFromGLBuffer**).

Basic Approaches to the OpenGL-OpenCL Interoperability

This tutorial discusses three different ways to utilize interoperability with OpenGL for the following general scenario:

- OpenCL memory object is created from the OpenGL texture.
- For every frame, the OpenCL memory object is acquired, then updated with an OpenCL kernel, and finally released to provide the updated texture data back to OpenGL.
- For every frame, OpenGL renders textured Screen-Quad to display the results.

There are three different interop modes that we compare in this tutorial:

1. Direct OpenGL texture sharing via **clCreateFromGLTexture**.
    - This is the most efficient mode of performance for interoperability with an Intel HD Graphics OpenCL device. It also allows the modification of textures “in-place.”
    - The number of OpenGL texture formats and targets that are possible to share via OpenCL images is limited.
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 the direct sharing method introduced above.
    - The upside is that there are fewer restrictions on the texture formats and targets beyond those imposed by the **glTexSubImage2D**.
3. Mapping the GL texture with **glMapBuffer**, wrapping the resulting host pointer as an OpenCL buffer for processing, and copying the results back on **glUnmapBuffer**.
    - Similar to the approach based on PBO referenced above, this approach allows you to perform interop for virtually any texture that can be updated with **glTexSubImage2D**.

- Unlike the original PBO-based method, it does not require any extension support.
- Performance-wise, it is even slower than the PBO-based method, particularly on small textures, because the OpenCL buffer is created/released in every frame.

From a performance perspective, the interoperability approach based on the direct OpenGL texture sharing—***Texture Sharing via clCreateFromGLTexture***—is the fastest way to share data with an Intel HD Graphics OpenCL device, while mapping the GL texture with **glMapBuffer** is the slowest. However, for a CPU OpenCL device, the performance is just the opposite.

We do not cover interoperability with OpenGL vertex buffers in this tutorial, but they are conceptually similar to Pixel-Buffer-Objects, in that they rely on the same **clCreateFromGLBuffer** for zero-copy sharing.

Also, plain data transfers from OpenGL to host memory and then to OpenCL and back (including mapping) is the most straightforward method that assumes neither extension usage nor actual sharing. As we stated, these methods allow the most general interoperability, while copying overheads (but not a power sipping) can be hidden with a multi-buffering approach. Refer to the details of the general asynchronous transfer to/from OpenGL for further reference ([3]). In order to be more performance/power efficient than a plain memory copy, an OpenCL implementation supporting **cl_khr_gl_sharing** is required, so we cover the extension in details first.

OpenCL-OpenGL Interoperability API

OpenCL-OpenGL interoperability is implemented as a Khronos extension to OpenCL [2]. The name of this extension is **cl_khr_gl_sharing**, and it should be listed among the supported extensions queried for the platform and the device.

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 provide pointers to the actual functions of the specific OpenCL platform. Following is the full list of functions for the extension and a short description for each:

Function	Description
clGetGLContextInfoKHR (http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/clGetGLContextInfoKHR.html)	Queries the devices associated with the OpenGL context
clCreateFromGLBuffer (http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/clCreateFromGLBuffer.html)	Creates an OpenCL buffer object from the OpenGL buffer object
clCreateFromGLTexture (http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/clCreateFromGLTexture.html)	Creates an OpenCL image object from the OpenGL texture object
clCreateFromGLRenderbuffer (http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/clCreateFromGLRenderbuffer.html)	Creates an OpenCL 2D image object from the OpenGL render buffer
clGetGLObjectInfo (http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/clGetGLObjectInfo.html)	Queries type and name for the OpenGL object used to create the OpenCL memory object
clGetGLTextureInfo (http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/clGetGLTextureInfo.html)	Gets additional information (target and mipmap level) about the GL texture object associated with a memory object
clEnqueueAcquireGLObjects (http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/clEnqueueAcquireGLObjects.html)	Acquires OpenCL memory objects from OpenGL
clEnqueueReleaseGLObjects (http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/clEnqueueReleaseGLObjects.html)	Releases OpenCL memory objects to OpenGL

Creating the Interoperability-Capable OpenCL Context

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

Via **clGetGLContextInfoKHR**, you can enumerate all OpenCL devices capable of sharing with the OpenGL context you are willing to interop. First, you need to set up 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, it is important to 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 potentially many more OpenCL devices that can potentially share the data with the OpenGL context (e.g. via copies). In the code example below, we use **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.

// queuring 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,…);

        …

}
```

This tutorial supports selecting the platform and device to run (refer to the section on controlling the sample), so after identifying the available devices (that support cl_khr_gl_sharing) 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);
```

Once we established a shared OpenCL-OpenGL context we can implement sharing using one of three basic ways described below:

Method 1: Texture Sharing via clCreateFromGLTexture

This method is the most efficient, allowing direct OpenGL-texture to OpenCL-image sharing with the help of **clCreateFromGLTexture**. This approach also allows modifying the content of the texture in place. Following are the required steps:

1. Creating OpenGL 2D texture the regular way:

```
//generate the texture ID

                glGenTextures(1, &texture));

                //binnding the texture

                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
                //(not GL_NEAREST_MIPMAP_* which would cause
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. Creating the OpenCL image corresponding to the texture (once):

```
cl_mem mem = clCreateFromGLTexture(context, CL_MEM_WRITE_ONLY,
GL_TEXTURE_2D, 0,texture,NULL);
```

   Note the **CL_MEM_WRITE_ONLY** flag that allows 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 access in the kernel in that case.
3. Acquiring the ownership via **clEnqueueAcquireGLObjects:**

```
glFinish();
                clEnqueueAcquireGLObjects(queue, 1,  &mem, 0, 0, NULL));
```

4. Executing 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.

Method 2: Texture Sharing via Pixel-Buffer-Object and clCreateFromGLBuffer

This method relies on the intermediate (staging) Pixel-Buffer-Object (PBO). It is less efficient than the direct sharing previously discussed, due to the final copying of the PBO bits to the texture. However, it allows for the potential of sharing textures of more formats (refer to the formats that **glTexSubImage2D** supports), unlike the limited set supported by **clCreateFromGLTexture**.

The code sequence is as follows:

1. Create OpenGL 2D texture in the standard 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);
   ```

   Note that the **CL_MEM_WRITE_ONLY** flag as buffer contains no original texture data, so there is no point to making it readable.
4. Acquire the 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 in the previous section (only kernel itself is slightly different, as it operates on the OpenCL buffer and not image).
5. Finally, streaming 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);
   ```

Method 3: Texture Sharing with glMapBuffer

This method is similar to the previous approach (PBO-based), but rather than rely on the **clCreateFromGLBuffer** to share the PBO with OpenCL, it performs straightforward mapping of the PBO to the host memory so it doesn't 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 (the buffer is created/destroyed in 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. Call the OpenCL kernel that alters the buffer content, changing values to green:

   ```
                   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**. Note 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) on **clEnqueueUnmapMemObject** only.
6. Release the OpenCL buffer:

   ```
   clReleaseMemObject(mem);
   ```

7. The rest of the procedure is the same to the last (fifth) step of the previous approach. PBO bits are copied to the texture with **glTexSubImage2D**.

Synchronization

To maintain data integrity, the application is responsible for synchronizing 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 subsequent OpenGL commands that reference the released objects (after **clEnqueueReleaseGLObjects**), the application is responsible for ensuring that any pending OpenCL operations that 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 allows creating OpenCL event objects from the OpenGL fence object. The OpenGL fence can be placed in the OpenGL command stream, allowing it to wait for completion of that fence in the OpenCL 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. Similar **clEnqueueReleaseGLObjects** guarantee the OpenCL is done with the objects, so no explicit **clFinish** is required. This is referred to as “implicit synchronization.”

This 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.

Notes on Textures, Formats, and Targets

It is important to understand that there is a limit to the number of OpenGL texture formats and targets that are possible to share via OpenCL images. You can find the full information on this topic on the Khronos web site: https://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/clCreateFromGLTexture.html (https://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/clCreateFromGLTexture.html)

There is a well-defined list of OpenGL texture usages (“targets”) that can be shared:

- An OpenCL (1D, 2D, 3D) image object can be created from the regular OpenGL (1D, 2D, 3D) texture.

- A 2D OpenCL image can be created from single face of an OpenGL cubemap texture.
- An OpenCL 1D or 2D image array can be created from an OpenGL 1D or 2D texture array.

Refer to Table 9.4 in the *The OpenCL Extension Specification*, Version 1.2 [2] that describes the list of GL texture internal formats and the corresponding image formats in OpenCL. These are the most important four-channel formats, such as GL_RGBA8 or GL_RGBA32F, for which the mapping is guaranteed.

Textures created with other OpenGL internal formats may also have a mapping to a CL image format. So, if such mappings exist (they are implementation-specific), the **clCreateFromGLTexture** succeeds; otherwise it fails with **CL_INVALID_IMAGE_FORMAT_DESCRIPTOR**.

Note that single-channel textures are generally not supported for sharing.

Also note that OpenGL depth (depth-stencil) buffer 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 **cl_khr_gl_depth_images** support from the implementation).

Conclusion

It is important to follow the right approach for OpenCL-OpenGL interoperability, taking into account limitations such as texture usages and formats and caveats like synchronization between the APIs. Also, the approach to interoperability (direct sharing, PBO, or plain mapping) might be different depending on the target OpenCL device.

Still, when utilizing the right approach from those discussed in this tutorial, you can achieve the best of both worlds: graphics and computing. For Intel HD Graphics and Iris Pro Graphics OpenCL devices, the direct sharing discussed in detail in this tutorial is ultimately the right way to go.

OpenCL can now be downloaded as part of Intel® Media Server Studio (https://software.intel.com/en-us/intel-media-server-studio)suite.

Resources

[1] 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
(http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/cl_khr_gl_sharing.html)

[2] *The OpenCL Extension Specification*, Version: 1.2, Document Revision: 19, Khronos OpenCL Working Group: http://www.khronos.org/registry/cl/specs/opencl-1.2-extensions.pdf
(http://www.khronos.org/registry/cl/specs/opencl-1.2-extensions.pdf)

[3] *OpenGL Insights* (Asynchronous Buffer Transfers Chapter):
http://www.seas.upenn.edu/~pcozzi/OpenGLInsights/OpenGLInsights-AsynchronousBufferTransfers.pdf
(http://www.seas.upenn.edu/~pcozzi/OpenGLInsights/OpenGLInsights-AsynchronousBufferTransfers.pdf)

1 comment

Top

Robby S (/en-us/user/503697) said on Aug 20,2014

[(/en-us/user/503697)](/en-us/user/503697)

There are very few articles talking about this topic. I am glad I found this detailed, up-to-date tutorial.

I have a question about your article, as posted here: https://software.intel.com/en-us/forums/topic/520556. If you could take a look and provide some advice, I'd be very grateful.

Cheers,

Add a Comment

[Sign in (/en-us/user/login?language=en-us&destination=node/509948&https=1)](/en-us/user/login?language=en-us&destination=node/509948&https=1)

Have a technical question? [Visit our forums (/en-us/forum)](/en-us/forum).
Have site or software product issues? [Contact support (/en-us/contact)](/en-us/contact).