Sharing Surfaces between OpenCL™ and DirectX* 11 on Intel® Processor Graphics

Adam Lake, Robert Ioffe

2/18/2015

Contents

Introduction

This tutorial demonstrates how to share surfaces between OpenCl™ and DirectX* 11 with Intel ® Processor Graphics on Microsoft Windows*, using the surface sharing extension in OpenCL. The goal is to provide access to the expressiveness enabled by the OpenCL C kernel and the rendering capabilities of the DirectX11 API. One example where this could be used would be a real-time computer vision application, which runs a feature detector over an image in OpenCL, then uses DirectX 11 to render the final output to the screen in real time with features clearly marked. Another would be using a dynamically generated procedural texture created in OpenCL when rendering a 3D object in the scene. Finally, imagine post-processing an image with OpenCL after rendering the scene using the 3D pipeline. This could be useful for color conversions, resampling, or performing compression in some scenarios.

To get you started with surface sharing, we will show how to update a texture created using DirectX 11 with OpenCL. The same processes apply to updates to a vertex buffer or an off-screen framebuffer object that might be used in a non-interactive offline image processing pipeline.

The surface sharing extension is defined in the OpenCL extension specification with the string `cl_khr_dx11_sharing`. We also leverage the context property `CL_CONTEXT_INTEROP_USER_SYNC`, which is supported on Intel Processor Graphics and set to the default value `CL_FALSE`.

We occasionally use DX11 as shorthand for DirectX 11.

Motivation

This tutorial will show you how to create shared surfaces between OpenCL and DirectX.

Key Takeaway

This tutorial shows you how to share a surface between OpenCL and DirectX11 using the `cl_khr_dx11_sharing` extension, and take advantage of `CL_CONTEXT_INTEROP_USER_SYNC` on Intel Processor Graphics. When creating the DX11 texture description, set the flag `MiscFlags` field of `D3D11_TEXTURE2D_DESC` object to `D3D11_RESOURCE_MISC_SHARED` to ensure no copy takes place between OpenCL and DirectX 11. Note that the `D3D11_RESOURCE_MISC_FLAG` cannot be used when creating resources with `D3D11_CPU_ACCESS` flags. For more detail on this last point see the MSDN references at the end of the article.

Intel® Processor Graphics with Shared Physical Memory

Intel Processor Graphics shares memory with the CPU. Figure 1 shows their relationship. While not shown in this figure, several architectural features exist that enhance the memory subsystem. For example, cache hierarchies, samplers, support for atomics, and read and write queues are all utilized to get maximum performance from the memory subsystem.

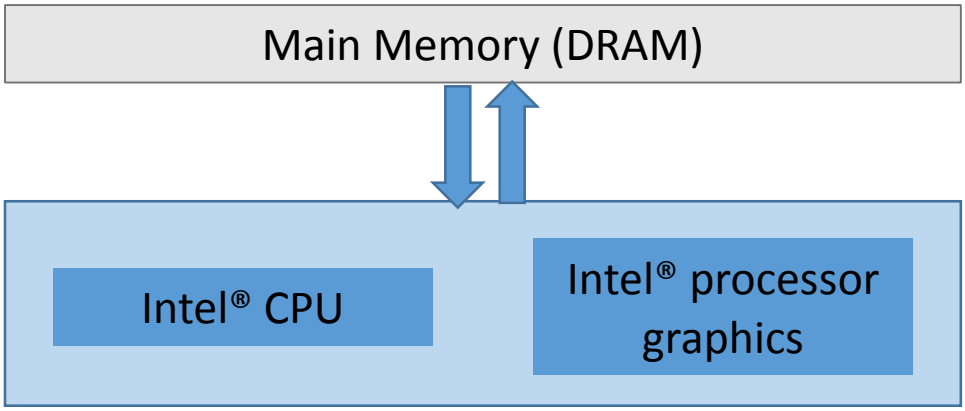

Figure 1. *Relationship of the CPU, Intel® Processor Graphics, and main memory. Notice a single pool of memory is shared by the CPU and GPU, unlike discrete GPUs that have their own dedicated memory that must be managed by the driver.*

Synchronization between OpenCL and DirectX 11

When sharing surfaces between OpenCL and DirectX 11, it is important to handle the possibility that a program may have both DirectX and OpenCL trying to update the same surface or surface location at the same time. There are two ways to handle this. The flag `CL_CONTEXT_INTEROP_USER_SYNC` is used to determine if the developer is responsible for handling the synchronization between the APIs. If this flag is set to CL_FALSE, as is the case in this sample, the driver will handle ensuring that the DirectX 11 operations on the surface issued before `clEnqueueAcquireD3D11ObjectsKHR` have posted their results to memory before OpenCL operates on the surface. Similarly, this flag ensures that the OpenCL operations have completed before any DX11 operations are allowed to operate on the surface upon `clEneuqueReleaseD3D11ObjectsKHR`. This flag and the semantics make surface sharing very easy between the two APIs.

Overview of Surface Sharing between OpenCL and DirectX 11

Below is a summary of the logic the sample code performs for surface sharing. The next section provides detailed explanation of each step.

Initialization

1. OpenCL:
    a. Query to determine if the extension `cl_khr_dx11_sharing` is supported; exit if unsupported.
    b. Create the context, passing the appropriate device options.
    c. Create a queue on the device and context that supports sharing between OpenCL and DX11.
2. DirectX: Create a DX11 texture that will be shared with OpenCL; make sure to set the MiscFlags field of the D3D texture descriptor.
3. OpenCL: Using the DX11 handle created in 2, create a shared surface via the OpenCL extension.

Steps 1 and 2 can be interchanged. Step 3 must follow steps 1 and 2.

Writing to the shared surface

1. Lock the surface for OpenCL exclusive access.
2. Write to the surface via the OpenCL C kernel. In the case of texture data, be sure to use the image read and/or write functions and pass in the image appropriately.
3. Unlock the surface so that DirectX may now read or write the surface.

The Render Loop

The render loop uses a simple pass through a programmable vertex and a pixel shader to texture map two screen-oriented triangles that form a quadrilateral for displaying the result. The quadrilateral uses only a part of the screen to show the clear color on the background of the rendering, which is useful for debugging more complex scenarios.

Shutdown

1. Cleanup the state objects.

Details of Surface Sharing between OpenCL and DirectX 11

Initialization

1. OpenCL:
    a. Query to determine if the extension `cl_khr_dx11_sharing` is supported; exit if unsupported.

Not every implementation of OpenCL supports surface sharing between OpenCL and DirectX 11, so the first step is to determine if the extension even exists on the system. We iterate through the platforms looking at the extension string for a platform that supports surface sharing. Careful reading of the specification highlights this is a *platform* extension or a *device* extension, so we actually have to check both! Later, when we create a context we will have to query to determine which of our devices in the context can share with the DX11 context.

This sample is only supported on Intel Processor Graphics, but it should be trivial to expand the scope to other GPUs. The extension string we are looking for is `cl_khr_dx11_sharing`. The relevant code snippet is:

```
char extension_string[1024];
memset(extension_string, '\0', 1024);
status = clGetPlatformInfo( platforms[i],
                            CL_PLATFORM_EXTENSIONS,
                            sizeof(extension_string),
                            extension_string,
                            NULL);
char *extStringStart = NULL;
extStringStart = strstr(extension_string, "cl_khr_dx11_sharing");
if(extStringStart != 0){
       printf("Platform does support cl_khr_dx11_sharing\n");
       …
}
```

Note a similar code sequence applies to the device query and is included in the example code in case you want to modify the code to work on other platforms.

b. Create the context, passing the appropriate device options.

```
cl_context_properties cps[] =
{
        CL_CONTEXT_PLATFORM, (cl_context_properties)g_platformToUse,
        CL_CONTEXT_D3D11_DEVICE_KHR, (intptr_t)g_pd3dDevice,
        CL_CONTEXT_INTEROP_USER_SYNC, CL_FALSE,
        0
};
```

Create a queue on the device and context that supports sharing between OpenCL and DX11. Note that in the context we specified the CL_CONTEXT_INTEROP_USER_SYNC flag and set it to CL_FALSE, which is the default value. Use this flag to specify whether you are going to manage synchronization between OpenCL and DX11 or let the runtime handle it.

We first query for the number of devices in our platform that meet our criteria. Note because this is an extension, and we have already verified the platform supports this extension, we have to create a pointer to the extension function we use for this next step.

```
        clGetDeviceIDsFromD3D11KHR_fn ptrToFunction_clGetDeviceIDsFromD3D11KHR = NULL;
        ptrToFunction_clGetDeviceIDsFromD3D11KHR = (clGetDeviceIDsFromD3D11KHR_fn)
clGetExtensionFunctionAddressForPlatform(g_platformToUse, "clGetDeviceIDsFromD3D11KHR");

        cl_uint numDevs = 0;
        //careful with the g_pd3DDevice
        status = ptrToFunction_clGetDeviceIDsFromD3D11KHR(g_platformToUse,
        CL_D3D11_DEVICE_KHR, (void *)g_pd3dDevice, CL_PREFERRED_DEVICES_FOR_D3D11_KHR, 0,
        NULL, &numDevs);
        testStatus(status, "Failed on clGetDeviceIDsFromD3D11KHR");
```

Using this information we create a device with only the DX11 context we requested to share.

```
        cl_device_id *devID = NULL;
        g_clDevices = (cl_device_id *)malloc(sizeof(cl_device_id) * numDevs);
        ptrToFunction_clGetDeviceIDsFromD3D11KHR(g_platformToUse, CL_D3D11_DEVICE_KHR,
(void *)g_pd3dDevice, CL_PREFERRED_DEVICES_FOR_D3D11_KHR, numDevs, g_clDevices, NULL);
        testStatus(status, "Failed on clGetDeviceIDsFromD3D11KHR");

        //create an OCL context from the device we are using as our DX11 rendering device
        g_clContext = clCreateContext(cps, 1, g_clDevices, NULL, NULL, &status);
        testStatus(status, "clCreateContext error");
```

Finally, create our command queue for the application on this device. Note that this is focused on getting surface sharing to run well on Intel Processor Graphics so we are going to spare some of the complexity of making platform portable code for device selection.

```
        //create an openCL commandqueue
        g_clCommandQueue = clCreateCommandQueue(g_clContext, devID, 0, &status);
        testStatus(status, "clCreateCommandQueue error");
```

2. DirectX: Create a DX11 texture that we will share with OpenCL. Here we use the flag D3D11_RESOURCE_MISC_SHARED to ensure we create an optimal sharing scenario with Intel Processor Graphics.

```
void CreateTextureDX11()
{
	unsigned char *texture = NULL;
	texture = (unsigned char *)malloc(sizeof(unsigned char) * NUM_IMAGE_CHANNELS *
SHARED_IMAGE_HEIGHT * SHARED_IMAGE_WIDTH);
	if(texture == nullptr)
	{
		printf("error creating texture\n");
	}

	for(unsigned int i=0;i<NUM_IMAGE_CHANNELS * SHARED_IMAGE_HEIGHT *
SHARED_IMAGE_WIDTH;)
	{
		texture[i++] = 255;
		texture[i++] = 0;
		texture[i++] = 0;
		texture[i++] = 255;
	}

	D3D11_TEXTURE2D_DESC desc;
	ZeroMemory(&desc, sizeof(D3D11_TEXTURE2D_DESC));
	desc.Width = SHARED_IMAGE_WIDTH;
	desc.Height = SHARED_IMAGE_HEIGHT;
	desc.MipLevels = 1;
	desc.ArraySize = 1;
	desc.Format = DXGI_FORMAT_R8G8B8A8_UNORM; desc.SampleDesc.Count = 1;
	desc.SampleDesc.Quality = 0;
	desc.Usage = D3D11_USAGE_DEFAULT;
	desc.BindFlags = D3D11_BIND_SHADER_RESOURCE;
	desc.CPUAccessFlags = 0;
	if(g_UseD3D11_RESOURCE_MISC_SHAREDflag == true)
	{
		printf("Using the D3D11_RESOURCE_MISC_SHARED flag\n");
		desc.MiscFlags = D3D11_RESOURCE_MISC_SHARED;
	}
	else
	{
		desc.MiscFlags = 0;

	}
	D3D11_SUBRESOURCE_DATA tbsd;
	ZeroMemory(&tbsd, sizeof(D3D11_SUBRESOURCE_DATA));
	tbsd.pSysMem = (void *)texture;
	tbsd.SysMemPitch = SHARED_IMAGE_WIDTH * NUM_IMAGE_CHANNELS;
	tbsd.SysMemSlicePitch = SHARED_IMAGE_WIDTH * SHARED_IMAGE_HEIGHT *
NUM_IMAGE_CHANNELS;
	g_pd3dDevice->CreateTexture2D(&desc, &tbsd, &g_pSharedDX11Texture2D);
	//still need to bind

	free(texture);
}
```

3. OpenCL: Using the DX11 handle created in 2, create a shared surface via the OpenCL extension.

This is the heart of our surface sharing API. The sharing will work only if the following call succeeds:

```
int ShareDX11BufferWithCL()
```

```
{
	int status = 0;

	g_SharedRGBAimageCLMemObject =
ptrToFunction_clCreateFromD3D11Texture2DKHR(g_clContext, CL_MEM_WRITE_ONLY,
g_pSharedDX11Texture2D, 0, &status);
	if(status == 0)
	{
		printf("Successfully shared!\n");
		status = SUCCESS;
	}
	else
	{
		printf("Sharing failed\n");
		status = FAIL;
	}
	return status;
}
```

Steps 1 and 2 can be interchanged. Step 3 must follow steps 1 and 2.

Writing to the shared surface

We need to make sure when OpenCL is reading or writing the surface that DX11 is not using the surface and vice versa. To do this the extension supports functions to acquire and release exclusive access to the surface. Here we describe the steps to use these APIs. There are two ways to do this. First, we could use sync objects in our code. Second, and the way we do it in this sample, is to take advantage of the behavior when we use the flag `CL_CONTEXT_INTEROP_USER_SYNC` set to `CL_FALSE`. When this flag is set, the synchronization is implicit at the `clEnqueueAcquireD3D11ObjectsKHR()` and `clENqueueReleaseD3D11ObjectsKHR`() API call sides.

4. Lock the surface for OpenCL exclusive access.

```
status = ptrToFunction_clEnqueueAcquireD3D11ObjectsKHR(g_clCommandQueue, 1,
&g_SharedRGBAimageCLMemObject, 0, 0, 0);
```

5. Write to the surface via the OpenCL C kernel. In the case of texture data, be sure to use the image read and/or write functions and pass in the image appropriately. The kernel simply writes into a subset of the texels of the texture.

```
kernel void drawBox(__write_only image2d_t output, float fDimmerSwitch)
{
	int x = get_global_id(0);
	int y = get_global_id(1);

	int xMin = 0, xMax = 1, yMin = 0, yMax = 1;

	if((x >= xMin) && (x <= xMax) && (y >= yMin) && (y <= yMax))
	{
		write_imagef(output, (int2)(x, y), (float4)(0.f, 0.f, fDimmerSwitch, 1.f));
	}
}
```

6. Unlock the surface so that OpenCL can now read or write the surface.

```
status = ptrToFunction_clEnqueueReleaseD3D11ObjectsKHR(g_clCommandQueue, 1,
&g_SharedRGBAimageCLMemObject, 0, NULL, NULL);
```

The Render Loop

The render loop uses a simple pass through a programmable vertex and a pixel shader to texture map two screen-oriented triangles that form a quadrilateral for display of the result.

Shutdown

1. Clean up the state objects.

This sample doesn't require much to be done for cleanup. The one object is the `cl_mem` object for the shared surface. On the DX11 side we must release the texture we created to share, the sampler for the texture, and the view on the texture.

Future Work

This tutorial covers the basics of surface sharing. With more time we'd like to expand the scope of the tutorial to cover additional use cases touched on here.

Sharing explicit synchronization events between OpenCL and DirectX 11

In addition to the implicit synchronization used in this article, OpenCL and DirectX 11 can also use the `CL_CONTEXT_INTEROP_USER_SYNC flag` set to `CL_TRUE` and handle synchronization explicitly with Windows synchronization objects. It would be interesting to compare the performance of the two. Driver architects we've talked to have found the difference to be minimal to none at all so this is the current recommendation, but it would be instructive to verify.

Sharing Framebuffers, Depth, Stencil, and MSAA surfaces

OpenCL and DirectX 11 lack surface sharing capabilities for many useful surfaces. It would be useful to have these supported similar to the case with surface sharing between OpenGL and DirectX 11. Also, while the extension and documentation are not explicit about this, only non-mipmapped surface sharing is required to be supported with this extension.

Double-buffering

We could explore the tradeoffs in complexity and performance using a double-buffering scheme. In this tutorial we focused on functionality and the basics of surface sharing.

What do to when no surface sharing is supported?

Maxim Shevtsov has an article cited in the references that covers the case of when a copy must take place between OpenCL and OpenGL. While these recommendations are focused on OpenGL many of the recommendations apply to sharing with DirectX 11.

Surface Sharing Example

Dependencies

The surface sharing sample has the following dependencies:

- Windows 8 or greater SDK; note that the DirectX SDK is now deprecated and the Windows SDK is used for Direct3D* development.

- Microsoft Visual Studio* 2012 or 2013
    - Make use of the DirectX-specific portions of the SDK including the d3dcompiler_46.dll.
    - D3dcompiler_46.dll is copied to the respective Debug or Release directory that the executables reside at the SOLUTION (not the PROJECT) level. There are other ways to handle this dependency.
    - This sample should work with other versions of Visual Studio, but has not been tested,
- Intel® INDE (Intel® OpenCL™ Code Builder is part of it now):
    - Use the `cl.h`, `cl_d3d11.h`, and `opencl.lib` files

The following settings were used in the sample, but yours may be slightly different:

- Copy `d3dcompiler_46.dll` to the debugger release directory that the executables will be built to at the SOLUTION (not PROJECT) level.
- Add the location of `cl.h` and `cl_d3d11.h` to your include path. For example,
    - `C:\Program Files (x86)\Intel\OpenCL SDK\3.0\include\CL`
- Add the location of the OpenCL Library to your library path: For example,
    - `C:\Program Files (x86)\Intel\OpenCL SDK\3.0\lib\x86`
- Add `OpenCL.lib` to your set of statically linked libraries.

Sample File and Directory Structure

The solution is located in the directory `CL_20_DX11_surface_sharing`. A directory by the same name is where the project and source files reside. This sample code is focused on demonstrating surfaced sharing between OpenCL and DirectX 11 and is not meant to be a product-quality implementation.

The files are organized as follows:

- `Main.cpp` – Mostly windowing code, the windows message pumping loop, dispatches events to the respective OpenCL and DX11 functions, and a keyboard handler
- `Common_CL_20_DX11.h` – Flags that are used for both OpenCL and DX11
- `DX11.h, DX11.cpp` – Functions specific to DX11 including initialization, shutdown, and rendering
- `OCL.h, OCL.cpp` – Functions specific to OpenCL
- `OpenCLRGBAFile.cl` – Source code for the OpenCL kernel
- `PixelShader.hlsl, VertexShader.hlsl`– Simple vertex and pixel shaders for DX11

Building and Running the example

Build this sample code by selecting *Build->Build Solution* from the main menu. All of the executables should be generated. You can run them in Visual Studio directly or go to the Debug and/or Release directories that are located in the same location as the `CL_20_DX11_surface_sharing` solution file.

To run the sample, press F5 in the Visual Studio IDE. The three relevant kernel and shader files are `OpenCLRGBAFile.cl`, `PixelShader.hlsl`, and `VertexShader.hlsl`.

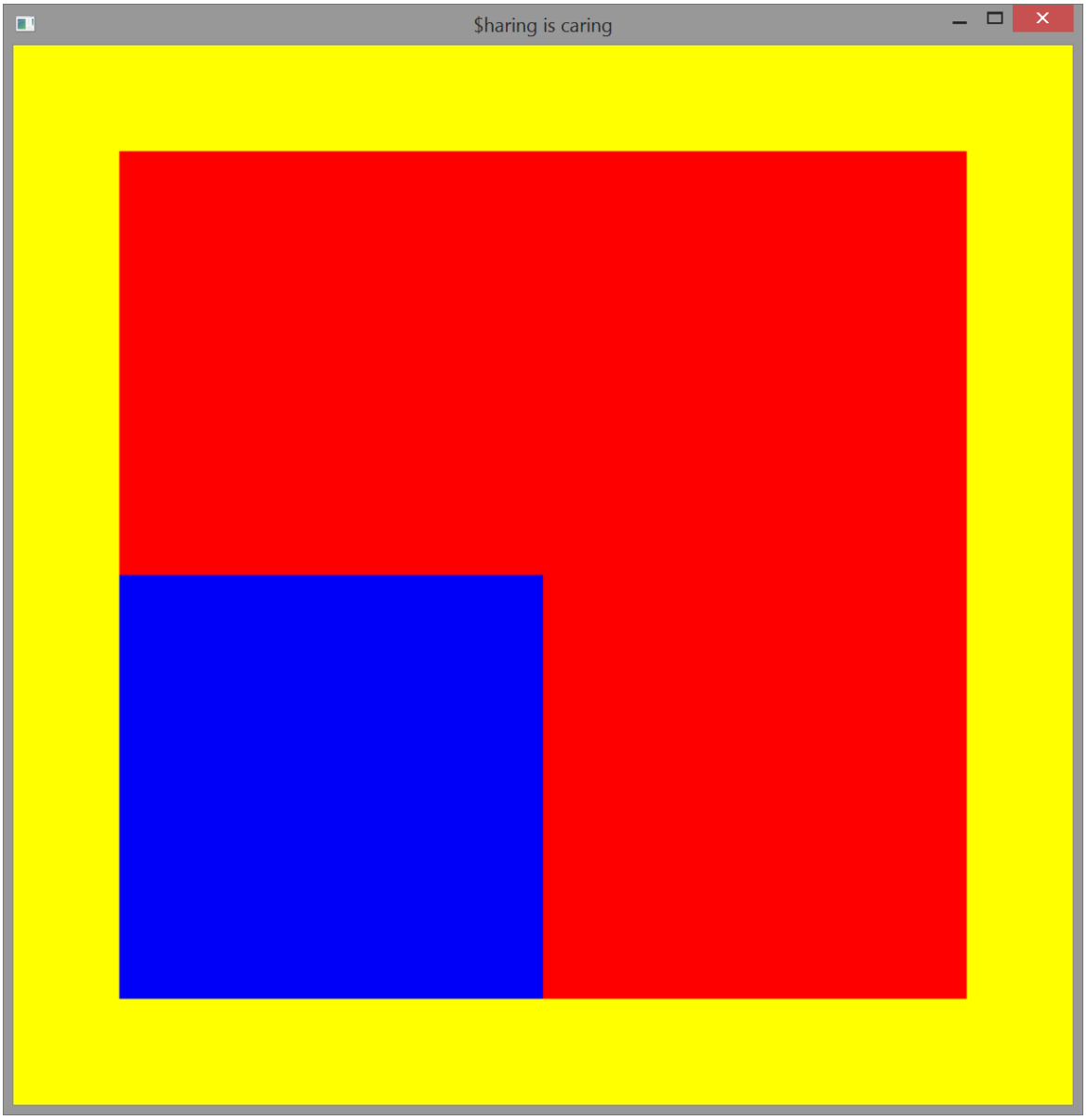

Figure 2. *Expected result of sample. The yellow is the background clear color. The image is a screen-oriented quadrilateral made of two texture-mapped triangles. The texture is a small red 4x4 texture map with the lower left portion of texels being written by OpenCL™ after being originally populated by OpenGL*. OpenCL writes a value to the blue color channel cycling from black to blue (0 to 255 in the blue channel).*

To learn more

There are additional buffer types that you can share between OpenCL and DirectX 11. Also, there are additional synchronization mechanisms you can leverage. These are detailed in the OpenCL Extension spec, and you can look there for more details.

Acknowledgements

Thanks to Murali Sundarasen, Aaron Kunze, Allen Hux, Pavan Lanka, Maxim Shevtsov, Michal Mrozek, Piotr Uminski, Stephen Junkins, Dan Petre, and Ben Ashbaugh. All were available for technical discussions, clarifications, or reviews along the way.

References

1. OpenCL 1.2 specification: https://www.khronos.org/registry/cl/
2. OpenCL 2.0 specification, composed of three volumes: the OpenCL C Language specification, the OpenCL Runtime API, and the OpenCL extensions: https://www.khronos.org/registry/cl/
3. Stephen Junkins' whitepaper: The Compute Architecture of Intel® Processor Graphics Gen 7.5: https://software.intel.com/sites/default/files/managed/f3/13/Compute_Architecture_of_Intel_Processor_Graphics_Gen7dot5_Aug2014.pdf. This is a must-read for anyone using OpenCL on Intel® Processor Graphics platforms.
4. Adam Lake's tutorial on minimizing buffer copies on Intel Processor Graphics: https://software.intel.com/en-us/articles/getting-the-most-from-opencl-12-how-to-increase-performance-by-minimizing-buffer-copies-on-intel-processor-graphics
5. Synchronization objects in Windows: http://msdn.microsoft.com/en-us/library/windows/desktop/ms686364(v=vs.85).aspx
6. Limitations of the use of the flag D3D11_RESOURCE_MISC_FLAG with D3D11_CPU_ACCESS: http://msdn.microsoft.com/en-us/library/windows/desktop/ff476203(v=vs.85).aspx
7. Limitations on the types of surfaces that can support surface sharing: http://msdn.microsoft.com/en-us/library/windows/desktop/hh404562(v=vs.85).aspx#direct3d_device_sharing

Definitions

Definitions for some terms used in this tutorial are included below. For more details consult the references.

- **Buffers**: OpenCL distinguishes between buffers and images. OpenCL buffers are laid out *linearly* in memory—think of a buffer as an array.
- **Textures**: Refer to buffers of data laid out in a *tiled* format and read via the on-die samplers in OpenGL or DX11. This memory layout enables increased performance via texture samplers that filter the input pixels read from memory via pre-specified filter kernels.
- **Surface**: Refers to buffers, textures, or images. It is a general term for data in memory that may be tiled or linear in layout. In some cases a surface has additional data such as dimension, height, width, and data layout attributes. These attributes are managed via the API (OpenCL, OpenGL, DirectX, etc.).
- **Samplers**: Are used to read from images in OpenCL and textures in OpenGL or DX11. The sampler exploits internal caches and the tiled layout of an image or texture in memory for improved performance when filtering. The samplers include caches and logic to perform sampling from several texels and (possibly) mip map levels at the same time and output a single texel value for a single request.

- **Images**: Refers to buffers of data laid out in a tiled format and read via the on-die samplers in OpenCL. They are the equivalent of OpenGL or DX11 textures. What image formats and texture formats can be shared or supported depends on the specific implementation.
- **Surface sharing:** Shorthand for *cross-API surface sharing* and is used to refer to the creation of a surface in one API and the use of the data in another. The motivation is to minimize creating multiple copies of the same surface but this is not strictly true unless we follow a set of device-dependent restrictions. This tutorial describes those restrictions for Intel Processor Graphics.
- **Texture mapping**: An association of pixels in memory to a polygon in the graphics pipeline. In this example we texture map an OpenGL or DX11 texture onto two screen-oriented polygons for display.
- **Shared Physical Memory**: The host and the device share the same physical DRAM. This is different from shared *virtual* memory, when the host and device share the same virtual addresses, and is not the subject of this paper. The key hardware feature that enables surface sharing is the fact that the CPU and GPU have shared *physical* memory. Shared physical and shared virtual memories are not mutually exclusive. Devices may not be able to see entire physical memory to support shared physical memory.
- **Intel® Processor Graphics**: The term used when referring to current Intel graphics solutions. Product names for Intel GPUs integrated in SoC include Intel® Iris™ Graphics, Intel® Iris™ Pro Graphics, or Intel® HD Graphics depending on the exact SoC. For additional hardware architecture details see The Compute Architecture of Intel® Processor Graphics Gen 7.5 document listed in the References section of this document or http://ark.intel.com/.

Legal Information