# OpenGL Interoperability with CUDA

网站：<https://www.3dgep.com/opengl-interoperability-with-cuda/>

示例程序：PosrprocessGL.zip

In this article I will discuss how you can use OpenGL textures and buffers in a CUDA kernel. I will demonstrate a simple post-process effect that can be applied to off-screen textures and then rendered to the screen using a full-screen quad. I will assume the reader has some basic knowledge of C/C++ programming, OpenGL, and CUDA. If you lack OpenGL knowledge, you can refer to my previous article titled [Introduction to OpenGL](https://www.3dgep.com/?p=636) or if you have never done anything with CUDA, you can follow my previous article titled [Introduction to CUDA](https://www.3dgep.com/?p=1821).

## Contents

* [1 Introduction](https://www.3dgep.com/opengl-interoperability-with-cuda/#Introduction)
* [2 Setting Up CUDA](https://www.3dgep.com/opengl-interoperability-with-cuda/#Setting_Up_CUDA)
* [3 Creating a Texture Object](https://www.3dgep.com/opengl-interoperability-with-cuda/#Creating_a_Texture_Object)
* [4 Creating a Pixel Buffer Object](https://www.3dgep.com/opengl-interoperability-with-cuda/#Creating_a_Pixel_Buffer_Object)
* [5 Creating a Renderbuffer](https://www.3dgep.com/opengl-interoperability-with-cuda/#Creating_a_Renderbuffer)
* [6 Creating a Framebuffer](https://www.3dgep.com/opengl-interoperability-with-cuda/#Creating_a_Framebuffer)
* [7 Register Resources with CUDA](https://www.3dgep.com/opengl-interoperability-with-cuda/#Register_Resources_with_CUDA)
  + [7.1 Register a Texture Resource with CUDA](https://www.3dgep.com/opengl-interoperability-with-cuda/#Register_a_Texture_Resource_with_CUDA)
  + [7.2 Register a Vertex Buffer or Pixel Buffer with CUDA](https://www.3dgep.com/opengl-interoperability-with-cuda/#Register_a_Vertex_Buffer_or_Pixel_Buffer_with_CUDA)
* [8 Rendering the Scene](https://www.3dgep.com/opengl-interoperability-with-cuda/#Rendering_the_Scene)
* [9 Post-Process the Scene](https://www.3dgep.com/opengl-interoperability-with-cuda/#Post-Process_the_Scene)
  + [9.1 Mapping the Resources](https://www.3dgep.com/opengl-interoperability-with-cuda/#Mapping_the_Resources)
  + [9.2 Mapping a Buffer Object to Device Memory](https://www.3dgep.com/opengl-interoperability-with-cuda/#Mapping_a_Buffer_Object_to_Device_Memory)
  + [9.3 Mapping a Texture Resource to Device Memory](https://www.3dgep.com/opengl-interoperability-with-cuda/#Mapping_a_Texture_Resource_to_Device_Memory)
  + [9.4 Binding a CUDA Array to a Texture Reference](https://www.3dgep.com/opengl-interoperability-with-cuda/#Binding_a_CUDA_Array_to_a_Texture_Reference)
    - [9.4.1 Texture Reference](https://www.3dgep.com/opengl-interoperability-with-cuda/#Texture_Reference)
    - [9.4.2 Binding the Texture](https://www.3dgep.com/opengl-interoperability-with-cuda/#Binding_the_Texture)
  + [9.5 Creating Global Memory for the Result](https://www.3dgep.com/opengl-interoperability-with-cuda/#Creating_Global_Memory_for_the_Result)
* [10 The CUDA Kernel](https://www.3dgep.com/opengl-interoperability-with-cuda/#The_CUDA_Kernel)
  + [10.1 Host Code](https://www.3dgep.com/opengl-interoperability-with-cuda/#Host_Code)
  + [10.2 The CUDA Kernel](https://www.3dgep.com/opengl-interoperability-with-cuda/#The_CUDA_Kernel-2)
* [11 Display the Final Result](https://www.3dgep.com/opengl-interoperability-with-cuda/#Display_the_Final_Result)
* [12 Exercise](https://www.3dgep.com/opengl-interoperability-with-cuda/#Exercise)
* [13 Conclusion](https://www.3dgep.com/opengl-interoperability-with-cuda/#Conclusion)
* [14 References](https://www.3dgep.com/opengl-interoperability-with-cuda/#References)
* [15 Download the Source](https://www.3dgep.com/opengl-interoperability-with-cuda/#Download_the_Source)

## Introduction

Besides the memory types discussed in previous article on the [CUDA Memory Model](https://www.3dgep.com/?p=2012), CUDA programs have access to another type of memory: **Texture** memory which is available on devices that support compute capability 1.0 and better and on devices that support compute capability 2.0 and better, you also have access to **Surface** memory. Texture memory is useful for fetching texture elements from a texture and surface memory is more like a pixel buffer object that simply represents a block of memory that can be both read from and written to.

Texture and surface memory reside in **device** memory (also called off-chip memory). Global memory also resides in device memory and we know that accessing global memory is relatively slow (about 100x slower) compared to accessing the on-chip (cache) memory. However, the high latency incurred by global memory accesses does not exactly apply to texture memory because unlike global memory, accesses to texture memory is cached on devices of compute compatibility 1.x.

On devices with compute capability 2.0, accesses to global memory is also cached.

Reading from texture or surface memory costs a single memory read from device memory only if a cache-miss occurs, otherwise it only costs a memory read from texture cache which is very low-latency memory access. Since the texture cache is optimized for 2D locality, threads of the same warp that access texture memory that are located close together in texture space will achieve best performance. Texture memory is also optimized for streaming fetches (when all the threads in a warp access a texture address with 2D locality) so even if a cache-miss does occur the latency to access texture memory will not be high.

There are several benefits to accessing device memory through texture or surface fetching rather than through global or constant memory:

* If the memory reads do not follow strict access patterns that are required to achieve high performance when accessing global or constant memory (coalesced memory access for example), we can still achieve high-bandwidth access as long as we can access the texture memory with spatial locality (texture fetches are located close to each other in the 2D texture).
* Addressing calculations are performed by dedicated units.
* Packed data may be broadcast to separate variables in a single operation.
* 8-bit and 16-bit integer input data can be converted to 32-bit floating point values during the texture fetch operation.

At the time of this writing, I don’t actually know what is meant by “Packed data may be broadcast to separate variables in a single operation”. If I find out, I will update this article with more information.

In this article I will show you how you can map an OpenGL 2D texture to a CUDA texture so that it can be accessed in an optimized way in a CUDA kernel.

## Setting Up CUDA

By default, the CUDA context is not configured to work with the OpenGL context. To tell CUDA that you will be using it with OpenGL, you must initialize the CUDA context and the OpenGL context together. To do that, you must first call **cudaGLSetGLDevice**. The only parameter to this method is the ID of the device in your system that should be setup to use the OpenGL context. If you have only 1 CUDA device, you can usually specify **0** this method to initialize the default device to share resources with OpenGL.

## Creating a Texture Object

Before we can start manipulating OpenGL textures in CUDA, we must first define a texture. You can create textures of many different pixel formats but for this article, I will use 4-component (Red, Green, Blue, and Alpha) unsigned byte textures (**GL\_RGBA**).

To create an OpenGL texture, you can use the following method:

|  |  |
| --- | --- |
| main.cpp | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | GLuint texture;  glGenTextures( 1, &texture );  glBindTexture( GL\_TEXTURE\_2D, texture );    // set basic parameters  glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_CLAMP\_TO\_EDGE);  glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_CLAMP\_TO\_EDGE);  glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_NEAREST);  glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_NEAREST);    // Create texture data (4-component unsigned byte)  glTexImage2D( GL\_TEXTURE\_2D, 0, GL\_RGBA, width, height, 0, GL\_RGBA, GL\_UNSIGNED\_BYTE, NULL );    // Unbind the texture  glBindTexture( GL\_TEXTURE\_2D, 0 ); |

On line 1, we define a handle that is used to uniquely define the OpenGL texture object. The method **glGenTextures** is used to obtain unique texture object IDs that we can use to refer to this texture throughout the application.

On line 3, the texture object is bound to the **GL\_TEXTURE\_2D** texture target. From this point on, we can use the **GL\_TEXTURE\_2D** target identifier to refer to this texture.

Each texture in OpenGL has a set of properties (or attributes) which we can manipulate using the glTexParameter[i|f] methods. The first two settings will determine what happens when we try to fetch a pixel beyond the size of the texture. In this case, we will simply clamp the out-of-bound texture coordinate to the edge of the texture map. Since texture coordinates are usually defined in the range [0..1), accessing a pixel outside of this range would usually result in an error (like trying to access an array out-of-bounds) but the **GL\_CLAMP\_TO\_EDGE** setting allows us to request a pixel of the texture outside of the normalized range without accessing out-of-bounds memory. The texture coordinates will simply be clamped into the allowed range when the texture is accessed.

The next settings on line 8, and 9 will determine how the pixels of the texture are blended if the pixel is mapped to an area larger (GL\_TEXTURE\_MIN\_FILTER) than a single texture element, or smaller (GL\_TEXTURE\_MAG\_FILTER) than a single texture element. In this case, **GL\_NEAREST** parameter specifies that no filtering should occur – just return the pixel closes to the requested texture coordinate.

We haven’t yet told OpenGL how large our texture and thus no texture memory has been allocated for it. To actually allocate memory for the texture, we use the **glTexImage2D** method. In addition to the size of the texture, we must also specify the internal format of the texture. In this case, I want to access this texture in CUDA with Red, Green, Blue, and Alpha components with each component being an unsigned byte.

On line 15, the texture object is unbound so we return OpenGL back to it’s normal state.

When no longer needed (when your application is finished running for example), the texture object can be deleted using the **glDeleteTextures** method.

## Creating a Pixel Buffer Object

If you graphics adapter has support for pixel buffer objects (if you have a graphics adapter that supports CUDA, you are pretty much guaranteed to have support for this extension), then you can use a pixel buffer object (PBO) to write the result of the CUDA kernel then copy the contents of the PBO to a texture to be rendered to the screen.

I am assuming you are using some kind of OpenGL extension library like [GLEW](http://glew.sourceforge.net/) or [GLEE](https://www.opengl.org/sdk/libs/GLee/) to check for the existence of (and to use) OpenGL extensions.

To create a pixel buffer object, you can use the following function:

|  |  |
| --- | --- |
| main.cpp | |
| 1  2  3  4  5  6 | GLuint bufferID;  glGenBuffers( 1, &bufferID );  glBindBuffer( GL\_PIXEL\_UNPACK\_BUFFER, bufferID );  glBufferData( GL\_PIXEL\_UNPACK\_BUFFER, size, NULL, GL\_STREAM\_DRAW );    glBindBuffer( GL\_PIXEL\_UNPACK\_BUFFER, 0 ); |

To create a PBO, we must perform 3 simple steps:

1. Generate a unique buffer object ID using the **glGenBuffers** method.
2. Bind the buffer using a valid target (for PBO’s this should be either **GL\_PIXEL\_PACK\_BUFFER**, or **GL\_PIXEL\_UNPACK\_BUFFER**). In this case, the target isn’t really important yet as long as it’s one of these two.
3. Define some data for the buffer. The buffer data is defined using the **glBufferData** method and it takes the target, the size of the buffer in bytes and the usage hints as parameters.

The final argument to the **glBufferData** method is the usage hints. In this case, we want a buffer that will be streamed (updated once every frame) and drawn to the screen (via a texture copy) so the **GL\_STREAM\_DRAW** usage hint is probably the best for what we want to use this buffer for. If you are curious what other usage hints are available, I encourage you to read the following topic: [http://www.songho.ca/opengl/gl\_pbo.html](http://www.songho.ca/opengl/gl_pbo.html#create).

When the buffer is no longer needed (when your application is finished running for example), you can use the **glDeleteBuffers** to release the buffer.

**Creating a Renderbuffer**

Texture objects are great for storing data that contains color information and pixel buffer objects are great for storing general (unspecified) pixel data but what about stencil or depth information? The Render buffer object is well suited for storing depth information.

To create a render buffer for storing depth values, you would use the following methods:

|  |  |
| --- | --- |
| main.cpp | |
| 1  2  3  4  5  6  7  8 | GLuint depthBuffer;  glGenRenderbuffers( 1, &depthBuffer );  glBindRenderbuffer( GL\_RENDERBUFFER, depthBuffer );    glRenderbufferStorage( GL\_RENDERBUFFER, GL\_DEPTH\_COMPONENT, width, height );    // Unbind the depth buffer  glBindRenderbuffer( GL\_RENDERBUFFER, 0 ); |

This isn’t much different than the way we define a PBO except for the way we define the storage for the render buffer. Since we want to use this render buffer for storing the depth information of our rendered scene, we will specify **GL\_DEPTH\_COMPONENT** as the internal format of the render buffer. This is perfectly suitable for the depth buffer that will be attached to the frame buffer object that I’ll define next.

Of course, if your finished with your render buffer (at the end of the program for example) then you should delete it using the **glDeleteRenderbuffers** method.

**Creating a Framebuffer**

Before we can apply the post-process effect to our scene, we must render it into an off-screen buffer called a frame-buffer. OpenGL defines several default frame-buffers but these buffers are best suited for rending our final post-processed scene onto. To create an intermediate buffer, we can just define our own frame-buffer by attaching a color texture and a depth buffer and render our scene to our custom frame-buffer. Then we can just use the color texture as an input to our CUDA kernel so we can process the scene. Then we render the post-processed image to the default OpenGL frame-buffer so that it appears on the screen.

You may want to check if your graphics card has support for frame-buffers by checking for the “**GL\_ARB\_framebuffer\_object**” extension. Again, if you have a graphics card that support CUDA, there is a pretty good chance your graphics adapter will support this extension.

To create a frame buffer we need to define at least one color texture and one depth buffer and attach these to the frame-buffer.

Using the methods described above to define a color texture and a depth buffer that match the width and height of our render window, we can then attach those buffers to our frame-buffer that will be used to render our scene.

To define a frame-buffer object you would use the following method:

|  |  |
| --- | --- |
| main.cpp | |
| 1  2  3  4  5  6 | GLuint framebuffer;  glGenFramebuffers( 1, &framebuffer );  glBindFramebuffer( GL\_FRAMEBUFFER, framebuffer );    glFramebufferTexture2D( GL\_FRAMEBUFFER, GL\_COLOR\_ATTACHMENT0, GL\_TEXTURE\_2D, colorAttachment0, 0 );  glFramebufferRenderbuffer( GL\_FRAMEBUFFER, GL\_DEPTH\_ATTACHMENT, GL\_RENDERBUFFER, depthAttachment ); |

The framebuffer is created using the **glGenFramebuffers** method shown here on line 2. Before we can populate the frame-buffer, we must bind it using the **glBindFramebuffer** method supplying **GL\_FRAMEBUFFER** as the target and the ID of the frame-buffer we just generated.

The frame-buffer can support multiple color attachment points and a single depth attachment point and a single stencil attachment point. It is not necessary to have a stencil attachment point and since we aren’t using it in this application, I will skip adding a stencil buffer to the frame-buffer in this example.

The color texture is attached to the frame buffer using the **glFramebufferTexture2D** method. The fist argument is always going to be **GL\_FRAMEBUFFER** and the second parameter is the attachment point we want to add this texture to. Theoretically, the frame buffer can support up to 32 color attachment points but the actual number of supported color attachment points should be queried using the method:

|  |  |
| --- | --- |
| main.cpp | |
| 1  2 | int maxAttachments = 0;  glGetIntegerv( GL\_MAX\_COLOR\_ATTACHMENTS, &maxAttachments ); |

The minimum supported color attachment points is 1, so if your graphics adapter has support for the **GL\_ARB\_framebuffer\_object** extension, then you are guaranteed to be able to attach at least one color attachment.

In our case, we want to attach the color texture we defined earlier to the **GL\_COLOR\_ATTACHMENT0** color attachment point.

The next parameters specify the texture target, texture object and mip-map level of the texture we generated earlier. Since we defined a 2D texture with only a single mip-map (at level 0) we specify the texture target should be **GL\_TEXTURE\_2D**, the texture object ID of the texture previously generated, and a mip-level of “0”.

The depth buffer was defined as a render buffer. The render buffers are attached to the framebuffer using the **glFramebufferRenderbuffer** method. The frame-buffer supports at most 1 depth attachment point. We use the **GL\_DEPTH\_ATTACHMENT** to specify the only depth buffer that is attached to this frame-buffer. Since it’s a render buffer, the target can only be **GL\_RENDERBUFFER** and final parameter to this method is the depth buffer ID we generated earlier.

Now that we’ve defined a color attachment and a depth attachment for our frame-buffer, it should be ready to render to; but we need to check that our frame-buffer is good enough according to our graphics driver. To do that, we use the method **glCheckFramebufferStatus** and if this method returns **GL\_FRAMEBUFFER\_COMPLETE** then we’re good to go. If it returns something else, then we need to determine what went wrong. If you are having trouble with your frame buffers, I would encourage you to read the topic on OpenGL frame buffer objects located here: <http://www.songho.ca/opengl/gl_fbo.html>.

**Register Resources with CUDA**

Before a texture or buffer can be used by a CUDA application, the buffer (or texture) must be registered. A resource that is either a texture object or a render buffer is treated differently than buffer objects (vertex buffer object or pixel buffer object). This might be confusing at first because of the naming of “render buffer” and “pixel buffer” and “vertex buffer”. A good way to remember this is that a pixel buffer object cannot be attached to a frame buffer but a render buffer can. In this way, a render buffer is more like a texture than a pixel buffer is.

**Register a Texture Resource with CUDA**

To register an OpenGL texture or render-buffer resource with CUDA, you must use the **cudaGraphicsGLRegisterImage** method. This method will accept an OpenGL texture or render-buffer resource ID as a parameter and provide a pointer to a **cudaGraphicsResource\_t** object in return. The **cudaGraphicsResource\_t** object is then used to map the memory location defined by the texture object so that it can be used as a texture reference in CUDA later.

The **cudaGraphicsGLRegisterImage** has the following signature:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | cudaError\_t cudaGraphicsGLRegisterImage(    struct cudaGraphicsResource\*\*  resource,    GLuint image,    GLenum target,    unsigned int flags  ) |

Where each property has the following definition:

* **struct cudaGraphicsResource\*\* resource**: A pointer to the registered resource object that can be used to map the OpenGL texture object to a CUDA texture reference.
* **GLuint image**: The unique identifier for the OpenGL texture or render buffer object that has been previously defined.
* **GLenum target**: Identifies the type of the object specified by **image**. If **image** is a texture resource, then **target** must be **GL\_TEXTURE\_2D**, **GL\_TEXTURE\_RECTANGLE**, **GL\_TEXTURE\_CUBE\_MAP**, **GL\_TEXTURE\_3D**, or **GL\_TEXTURE\_2D\_ARRAY**. If the **image** refers to a render-buffer object, then **target** must be **GL\_RENDERBUFFER**.
* **unsigned int flags**: The register flags specify the intended usage and can be one of the following values:
  + **cudaGraphicsRegisterFlagsNone**: This specifies no hint about the usage of this resource. In this case, CUDA assumes the resource will be used for both reading from and writing to. This is the default value.
  + **cudaGraphicsRegisterFlagsReadOnly**: This resource will be used for read-only purposes and CUDA will not be used to write to this resource.
  + **cudaGraphicsRegisterFlagsWriteDiscard**: Specifies that CUDA will not use this resource for reading from and every time it is needed, the entire buffer contents will be discarded. This is safe to do if you assume the entire buffer will be redrawn every frame.
  + **cudaGraphicsRegisterFlagsSurfaceLoadStore**: This flag specifies that this resource will be bound to a surface reference instead of a texture reference. This option is only available on devices that support compute capability 2.x.

This method will return **cudaSuccess** if nothing went wrong. If you try to use this method to bind an OpenGL resource object that is neither a texture object nor a render buffer (for example, you try to register a pixel buffer object or a vertex buffer object) then this function will probably return **cudaErrorUnknown** and some message like “Unknown device/driver error”. This vauge error probably indicates your not passing the right object type to the function or your trying to register a render buffer but specifying **GL\_TEXTURE\_2D** as the **target** value when you should be specifying **GL\_RENDERBUFFER** instead.

**Register a Vertex Buffer or Pixel Buffer with CUDA**

Vertex buffer objects and Pixel buffer objects are slightly easier to handle in CUDA because we don’t need to be concerned with such things as texture element fetching and texture filtering and texture coordinate out-of-bounds conditions. Pixel buffers and vertex buffers are more like C-arrays that reside in device memory instead of system memory. Also, it is easier to get a pointer to a pixel buffer or a vertex buffer object that can be used to directly access the memory of the buffer than it is to get a pointer to texture memory.

To register a buffer object, you need to use the **cudaGraphicsGLRegisterBuffer** method instead. This method has the following signature:

|  |  |
| --- | --- |
| 1  2  3  4  5 | cudaError\_t cudaGraphicsGLRegisterBuffer(    struct cudaGraphicsResource\*\*  resource,    GLuint  buffer,    unsigned int  flags  ) |

This method takes almost the same parameters as the previous method except we don’t need to specify the target parameter because we assume that the buffer object ID refers to a valid buffer object (pixel or vertex buffer object).

This method will also return **cudaErrorUnknown** if **buffer** is neither a pixel buffer object nor a vertex buffer object. So don’t try to register a texture object with **cudaGraphicsGLRegisterBuffer** and don’t try to register a buffer object using **cudaGraphicsGLRegisterImage**.

**Rendering the Scene**

Now that we’ve created our texture objects, render buffers, and pixel buffer objects and we’ve attached the appropriate object to the frame-buffer object, we can render our scene to the frame-buffer object.

If we assume that the **RenderScene** method will render all of the necessary geometry, then to render the scene to our custom frame-buffer, we simply do something like this:

|  |  |
| --- | --- |
| main.cpp | |
| 432  433  434  435  436 | // Bind the framebuffer that we want to use as the render target.  glBindFramebuffer( GL\_FRAMEBUFFER, g\_GLFramebuffer );  RenderScene();  // Unbind the framebuffer so we render to the back buffer again.  glBindFramebuffer( GL\_FRAMEBUFFER, 0 ); |

Using this simple technique we should now have the texture resource that we attached to the frame-buffer filled with the colors of our scene and the depth buffer is filled with the depth values of the scene.

We can now perform a post-process effect on the texture resource.

**Post-Process the Scene**

Immediately after unbinding the frame-buffer, we can perform the post-process step to apply a filter to our image.

**Mapping the Resources**

Before we can access the registered resources in CUDA, we must map the resources. This will effectively “lock” the resource to the CUDA resource object. If the texture object that was registered to the CUDA resource was accessed while the resource was mapped in CUDA, an error (or undefined behavior) would ensue. That’s why it’s very important to un-map the resource when it is no longer needed in CUDA.

To map a resource to be used in CUDA, you use the **cudaGraphicsMapResources** method. This method has the following signature:

|  |  |
| --- | --- |
| 1  2  3  4  5 | cudaError\_t cudaGraphicsMapResources (    int count,    cudaGraphicsResource\_t\* resources,    cudaStream\_t stream = 0  ) |

Where the parameters are defined:

* **int count**: The number of resources to map. It is generally a good idea to map all your resources in one call as the mapping of these resources is quite an expensive operation, this can be optimized if you do them all at once as opposed to one at a time.
* **cudaGraphicsResource\_t\* resources**: An array of pointers to the resources that are to be mapped.
* **cudaStream\_t stream**: A stream resource to help synchronize CUDA invocations. By default, this parameter is NULL (or 0) in which case, the internal stream object will be used to synchronize asynchronous CUDA invocations.

We’ve only mapped the resource so that we can guarantee that it’s safe to use in the CUDA kernel, but we still don’t have access to the contents of the resources. The next step is to get a pointer to the device memory that can be used in the CUDA kernel.

Depending on the original resource type, we will map the pointer to device memory in a different way. If it is a texture or render-buffer resource, we will use use the **cudaGraphicsSubResourceGetMappedArray** which will map the texture resource to a 2D CUDA array object. If we are using a vertex buffer or pixel buffer object, we can use the **cudaGraphicsResourceGetMappedPointer** to get a direct pointer to the device memory that refers to the graphics resource.

**Mapping a Buffer Object to Device Memory**

If you are mapping a vertex buffer object or a pixel buffer object, you must use the **cudaGraphicsResourceGetMappedPointer** method.

The signature of this method has the form:

|  |  |
| --- | --- |
| 1  2  3  4  5 | cudaError\_t cudaGraphicsResourceGetMappedPointer(    void\*\* devPtr,    size\_t\* size,    cudaGraphicsResource\_t resource  ) |

Where the parameters are defined as:

* **void\*\* devPtr**: The pointer to the device memory through which this resource will be accessed. This pointer can be used as a parameter to a CUDA kernel and accessed in the same way global memory is accessed in a kernel function.
* **size\_t\* size**: Returns the size of the buffer that is accessable from **devPtr**.
* **cudaGraphicsResource\_t resource**: The mapped resource that is to be accessed.

The resource referred to in the **resource** parameter must be mapped using the **cudaGraphicsMapResources** method described above.

If the resource registered to the **resource** parameter is not a vertex buffer or a pixel buffer object, this method will fail with **cudaErrorUnknown** error code and may give some message like “Unknown driver error” which is not very descriptive of the actual problem. If you have a resource to a texture object or a render-buffer object, you must use the **cudaGraphicsSubResourceGetMappedArray** method described next.

**Mapping a Texture Resource to Device Memory**

Mapping a texture resouce or a render-buffer resource is only possible using the **cudaGraphicsSubResourceGetMappedArray** method. In this case, the texture resource is mapped to a pointer to a **cudaArray**. However, the cudaArray cannot be used directly in a kernel and requires an additional step to access it. The additional step required depends on how the memory should be used in the kernel. If the resource will be used as a read-only texture in the kernel, then the **cudaArray** must also be bound to a texture reference that is used within the kernel to access the data. If you need to write to the resource from within the kernel, then you will need to bind the **cudaArray** to a surface reference that can be both read-from and written-to in the CUDA kernel, however surface references are only available on devices that support compute compatibility 2.0 and up. I will neglect surface references for the sake of simplicity and only talk about binding our resource to a texture reference that can be read-from in the CUDA kernel.

The first step to accessing the texture reference in the CUDA kernel is mapping the resource to a **cudaArray**. This is done using the method **cudaGraphicsSubResourceGetMappedArray**. This method has the following signature:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | cudaError\_t cudaGraphicsSubResourceGetMappedArray(    struct cudaArray\*\* array,    cudaGraphicsResource\_t resource,    unsigned int arrayIndex,    unsigned int mipLevel  ) |

And the parameters are defined as:

* **struct cudaArray\*\* array**: A pointer to a cudaArray through which the subresource of **resource** can be accessed.
* **cudaGraphicsResource\_t resource**: The mapped resource that was previously registered to an OpenGL texture or render-buffer.
* **unsigned int arrayIndex**: The array index if the resource references a texture array, or the cubemap face index if the resource references a cubemap. For a single 2D texture, this array index should be 0.
* **unsigned int mipLevel**: The texture’s mip-map level that you want to access. If the texture only has 1 mip-level, then supply 0 here again.

If you try to map a resource that isn’t a texture or render-buffer, then this function will return an error. If you are trying to access a vertex buffer object or a pixel buffer object, then you will need to use the **cudaGraphicsResourceGetMappedPointer** function.

This function returns a pointer to a CUDA array however, the CUDA array cannot be used directly in the CUDA kernel function. Before we can access the actual data, we must bind it to a texture reference.

**Binding a CUDA Array to a Texture Reference**

Before we can access the data in a CUDA array in the CUDA kernel, we must bind the array object to a texture reference or a surface reference.

**TEXTURE REFERENCE**

Before a kernel can use a CUDA array to read from a texture, the CUDA array object must be bound to a texture reference using the **cudaBindTextureToArray** method.

A texture reference is declared in global scope of your CUDA source file and has the following format:

|  |  |
| --- | --- |
| 1 | texture<DataType, Type, ReadMode> texRef; |

Where:

* **DataType**: Specifies the return type when the texture element is fetched. This parameter is restricted to the primitive integer and single-precision floating-point types and any of the 1, 2, 3, or 4-component vector types.
* **Type**: Specifies the dimensionality of the texture reference and can be **cudaTextureType1D**, **cudaTextureType2D**, or **cudaTextureType3D**. If the texture references a layered texture, this can also be one of the layered texture types **cudaTextureType1DLayered** or **cudaTextureType2DLayered**.
* **ReadMode**: This parameter determines how the value that is fetched from the texture is actually returned. It can be either **cudaReadModeNormalizedFloat** or **cudaReadModeElementType**. If **cudaReadModeNormalizedFloat** is specified and **DataType** is a 16-bit or 8-bit integer type, the value actually returned from a texture fetch is mapped to a floating point value in the range [0.0, 1.0] for an unsigned integer type and [-1.0, 1.0] for a signed integer type. If **cudaReadModeElementType** is specified, then no conversion takes place.

As an example, if we want to declare a texture reference to a 2D texture and we want the texture element to be returned as an 4-component unsigned char vector, you would declare a texture reference as such:

|  |  |
| --- | --- |
| 1 | texture<uchar4, cudaTextureType2D, cudaReadModeElementType> texRef; |

These properties explained above must be declared at compile time but a texture reference also defines a set of properties that can be manipulated at run-time by adjusting the texture reference properties in the host code. These additional properties define if the texture coordinates used to fetch a texture element are normalized or not, the addressing mode and texture filtering.

The **texture** type defined above is publicly derived from the **textureReference** type. The **textureReference** type has the following definition:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | struct textureReference {    int normalized;    enum cudaTextureFilterMode filterMode;    enum cudaTextureAddressMode addressMode[3];    struct cudaChannelFormatDesc channelDesc;  }; |

Where:

* **int normalized**: Specifies wheter texture coordinates are normalized or not. If it is non-zero, all texture elements are addressed with texture coordinates in the range **[0, 1]** rather than in the range **[0, width-1]**, **[0, height-1]** (or **[0, depth-1]** for 3D textures) where **width**, **height**, and **depth** are the dimensions of the texture. This property defaults to **0** which means that the texture coordinates are non-normalized.
* **enum cudaTextureFilterMode filterMode**: Specifies the filtering mode when fetching the texture elements. This can be either **cudaFilterModePoint** or **cudaFilterModeLinear**. For **cudaFilterModePoint**, the nearest texture element to the texture coordinate is returned without any blending. **cudaFilterModeLinear** is only valid if the **DataType** specified when the texture reference was declared is a floating point type and then the texture unit will perform a linear interpolation between neighboring texture elements.
* **enum cudaTextureAddressMode addressMode[3]**: Specifies the addressing mode for each dimension of the texture. This value can be either **cudaAddressModeClamp**, or **cuadaAddressModeWrap**. **cudaAddressModeClamp** can be used on normalized and non-normalized texture coordinates and will clamp any texture coordinates to the maximum and minimum range of texture coordinates. **cuadaAddressModeWrap** can only be specified with normalized texture coordinates and will cause out-of-range texture coordinates to wrap-around so a texture coordinate of 1.25 will be computed as 0.25. **cuadaAddressModeWrap** is useful for repeating textures.
* **struct cudaChannelFormatDesc channelDesc**: Describes the format of the value that is returned when fetching the texture. The **cudaChannelFormatDesc** structure has the following definition:

|  |  |
| --- | --- |
| 1  2  3  4 | struct cudaChannelFormatDesc {    int x, y, z, w;    enum cudaChannelFormatKind f;  }; |

* where **x**, **y**, **z**, and **w** are equal to the number of bits of each component that are returned from the texture fetch operation. The **f** member is of type **enum cudaChannelFormatKind** and can be one of:
  + **cudaChannelFormatKindSigned**: The components are signed integer type.
  + **cudaChannelFormatKindUnsigned**: The components are unsigned integer type.
  + **cudaChannelFormatKindFloat**: The components are single-precision floating point type.

Only the **normalized**, **filterMode**, and **addressMode** members can be manipulated in the host code at run-time. The **channelDesc** struct member is a read-only property and cannot be modified in the host code at run-time.

Before a kernel can use a texture reference to read from texture memory, the texture reference must be bound to a texture object using **cudaBindTexture**, or **cudaBindTexture2D** for linear memory, or **cudaBindTextureToArray** for CUDA arrays.

**BINDING THE TEXTURE**

Now that I’ve introduced texture references and we’ve declared a texture reference in the global scope of our CUDA source file we need to bind the CUDA array that we obtained with the **cudaGraphicsSubResourceGetMappedArray** method defined earlier. To do this, we’ll use the **cudaBindTextureToArray** method.

The **cudaBindTextureToArray** method has the following signature:

|  |  |
| --- | --- |
| 1  2  3  4  5 | template<class T , int dim, enum cudaTextureReadMode readMode>  cudaError\_t cudaBindTextureToArray(    const struct texture< T, dim, readMode >& tex,    const struct cudaArray\* array;  ) |

Where the parameters are defined as:

* **const struct texture< T, dim, readMode >& tex**: The texture reference that was previously defined in the global scope of your CUDA source file.
* **const struct cudaArray\* array**: A pointer to a CUDA array that was previously mapped using the **cudaGraphicsSubResourceGetMappedArray** method.

It is recommended that when you are done with the texture reference, that you unbind the texture reference using the **cudaUnbindTexture** method.

Now that we have a texture reference to the CUDA array object that is mapped to the CUDA resources that was registered to the OpenGL texture, we can use it in the kernel.

**Creating Global Memory for the Result**

Since we can only use a texture reference to read from a texture (there is no function to write to a texture reference), we need to allocate some global memory to store the result of our kernel function. To do that, we will use **cudaMalloc** to allocate some global memory that will be used to store the result of our kernel function.

First, we’ll define a parameter that will point to the block of global memory where we will write the result. Then, we’ll allocate the global memory large enough to store the result of post-process effect. We must allocate enough space in global memory to store the result of the entire source texture.

|  |  |
| --- | --- |
| Postprocess.cu | |
| 160  161  162 | uchar4\* dstBuffer = NULL;  size\_t bufferSize = width \* height \* sizeof(uchar4);  cudaMalloc( &dstBuffer, bufferSize ); |

Where **width** and **height** are the dimensions of the texture we want to process.

Since we only want to allocate this buffer once (or reallocate it only if the size of the input texture changes) and just use it every frame, we will probably allocate this buffer in some initialization function and just pass the pointer to the “scratch” buffer to the post-process method.

We now have a pointer to a block of global memory that is suitable to store the result of our post-process effect. After we run the kernel that performs the post-process effect, this block of memory will contain the result of the effect. We can copy this block of memory back to our texture that will be used to display the effect on screen. To do that, we can use the **cudaMemcpyToArray** method to copy the global memory to a CUDA array object that was previously mapped to a texture using **cudaGraphicsSubResourceGetMappedArray**.

|  |  |
| --- | --- |
| Postprocess.cu | |
| 170 | cudaMemcpyToArray( dstArray, 0, 0, dstBuffer, bufferSize, cudaMemcpyDeviceToDevice ); |

Where dstArray is a CUDA array that was previously mapped with cudaGraphicsSubResourceGetMappedArray, dstBuffer is a pointer to global device memory allocated with cudaMalloc, and bufferSize is the size of the buffer to copy in bytes and **cudaMemcpyDeviceToDevice** indicates that we are copying from device memory to device memory (from global memory to texture memory).

**The CUDA Kernel**

To perform the post-process effect in a CUDA kernel, we must execute the kernel function from the host code.

**Host Code**

On the host, we might execute the kernel function in such a way:

|  |  |
| --- | --- |
| Postprocess.cu | |
| 165  166  167  168  169  170 | size\_t blocksW = (size\_t)ceilf( width / 16.0f );  size\_t blocksH = (size\_t)ceilf( height / 16.0f );  dim3 gridDim( blocksW, blocksH, 1 );  dim3 blockDim( 16, 16, 1 );    PostprocessKernel<<< gridDim, blockDim >>>( dstBuffer, width, height ); |

If this seems foreign to you, refer to my previous article on [CUDA Execution Model](https://www.3dgep.com/?p=1913) .

You’ll notice that I am only passing the pointer to global device memory to the kernel function but I am not passing the source texture that we will be performing the post-process effect on. This is because the texture reference is declared in the global scope of my CUDA source file so it is already accessible to the CUDA kernel.

**The CUDA Kernel**

The CUDA kernel is where all the magic happens. The input texture is read from texture memory, the texture element is processed and the result is written to the destination buffer.

Let’s see how this might look in our CUDA kernel function:

|  |  |
| --- | --- |
| Postprocess.cu | |
| 36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71 | \_\_global\_\_ void PostprocessKernel( uchar4\* dst, unsigned int imgWidth, unsigned int imgHeight )  {      unsigned int tx = threadIdx.x;      unsigned int ty = threadIdx.y;      unsigned int bw = blockDim.x;      unsigned int bh = blockDim.y;        // Non-normalized U, V coordinates of input texture for current thread.      unsigned int u = ( bw \* blockIdx.x ) + tx;      unsigned int v = ( bh \* blockIdx.y ) + ty;        // Early-out if we are beyond the texture coordinates for our texture.      if ( u > imgWidth || v > imgHeight ) return;        // The 1D index in the destination buffer.      unsigned int index = ( v \* imgWidth ) + u;        float4 tempColor = make\_float4(0, 0, 0, 1);      for ( int i = 0; i < FILTER\_SIZE; ++i )      {          // Fetch a texture element from the source texture.          uchar4 color = tex2D( texRef, u + indexOffsetsU[i], v + indexOffsetsV[i] );            tempColor.x += color.x \* kernelFilter[i];          tempColor.y += color.y \* kernelFilter[i];          tempColor.z += color.z \* kernelFilter[i];      }        // Store the processed color in the destination buffer.      dst[index] = make\_uchar4(          Clamp<unsigned char>(tempColor.x \* invScale + offset, 0.0f, 255.0f),          Clamp<unsigned char>(tempColor.y \* invScale + offset, 0.0f, 255.0f),          Clamp<unsigned char>(tempColor.z \* invScale + offset, 0.0f, 255.0f),          1      );  } |

On line 57, the source texture is read using the **tex2D** function and the resulting color is returned as a 4-component unsigned char vector. The color value is processed by multiplying by the weights stored in the **kernelFilter** array (this array is declared as a static const array in the global scope of the CUDA source file).

The resulting color is scaled and offset before being stored in the resulting buffer on line 64. We also need to clamp the result to account for overflow in the color components.

The resulting buffer is then copied back to the texture in the host code after the kernel is finished processing the input texture.

The final step is to display the texture on the screen so we can see the post-processed result.

**Display the Final Result**

To display the post-processed image to the screen, we simply render the resulting texture using an orthographic projection matrix.

The following function can be used to display an OpenGL texture to the screen at the specified **x**, **y**, **width**, and **height**.

|  |  |
| --- | --- |
| main.cpp | |
| 447  448  449  450  451  452  453  454  455  456  457  458  459  460  461  462  463  464  465  466  467  468  469  470  471  472  473  474  475  476  477  478  479 | void DisplayImage( GLuint texture, unsigned int x, unsigned int y, unsigned int width, unsigned int height )  {      glBindTexture(GL\_TEXTURE\_2D, texture);      glEnable(GL\_TEXTURE\_2D);      glDisable(GL\_DEPTH\_TEST);      glDisable(GL\_LIGHTING);      glTexEnvf(GL\_TEXTURE\_ENV, GL\_TEXTURE\_ENV\_MODE, GL\_REPLACE);        glMatrixMode(GL\_PROJECTION);      glPushMatrix();      glLoadIdentity();      glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);        glMatrixMode( GL\_MODELVIEW);      glLoadIdentity();        glPushAttrib( GL\_VIEWPORT\_BIT );      glViewport(x, y, width, height );        glBegin(GL\_QUADS);      glTexCoord2f(0.0, 0.0); glVertex3f(-1.0, -1.0, 0.5);      glTexCoord2f(1.0, 0.0); glVertex3f(1.0, -1.0, 0.5);      glTexCoord2f(1.0, 1.0); glVertex3f(1.0, 1.0, 0.5);      glTexCoord2f(0.0, 1.0); glVertex3f(-1.0, 1.0, 0.5);      glEnd();        glPopAttrib();        glMatrixMode(GL\_PROJECTION);      glPopMatrix();        glDisable(GL\_TEXTURE\_2D);  } |

I will refrain from explaining this code because this is not an article on OpenGL rendering technique but on OpenGL interopability with CUDA. You can download the source code example at the end of this article to see this function in action!

The resulting effect should look like something similar to what is shown below:

This video shows the six filters (Unfiltered, Blur, Sharpen, Emboss, Invert, and Edge Detect) that are being applied to the scene. This video is best viewed at 480p resolution.

**Exercise**

1. Download the source code example at the end of this article and modify the source code so that the result of the post-process effect is stored in a pixel buffer object instead of a texture. Use the pixel buffer object to blit the result of the post-process effect to the screen.

Hint: Use **glDrawPixels** to copy pixels from a pixel buffer object to the OpenGL framebuffer.

**Q**. Is there a benefit to using pixel buffer objects to perform texture operations in CUDA over using an OpenGL texture object?

1. In the example source code provided at the end of this article, there is a macro definition called **USE\_SHARED\_MEM** which is by default disabled. Enabling it will cause the texture fetches to be stored in shared memory and the shared memory is processed instead.

**Q**. Does using shared memory improve the performance in this case? Explain your answer.

**Conclusion**

In this article I’ve demonstrated how you can bind an OpenGL texture object to a CUDA texture reference and use that texture reference in a CUDA kernel to perform a post-process effect on the OpenGL texture. I’ve also shown you can allocate a block of global memory and copy that memory to an OpenGL texture using **cudaMemcpyToArray**. And finally, we can visualize the result of the post-process effect by displaying the resulting texture using a full-screen quad.

# References

|  |
| --- |
| **NVIDIA Corporation (2011, May). NVIDIA CUDA C Programming Guide. (Version 4.0). Santa Clara, CA 95050, USA Available from:**[**http://developer.download.nvidia.com/compute/cuda/4\_0/toolkit/docs/CUDA\_C\_Programming\_Guide.pdf**](http://developer.download.nvidia.com/compute/cuda/4_0/toolkit/docs/CUDA_C_Programming_Guide.pdf)**. Accessed: 15 November 2011.** |
| **NVIDIA Corporation (2011, February). CUDA API Reference Manual. Santa Clara, CA 95050, USA Available from:**[**http://developer.download.nvidia.com/compute/cuda/4\_0/toolkit/docs/CUDA\_Toolkit\_Reference\_Manual.pdf**](http://developer.download.nvidia.com/compute/cuda/4_0/toolkit/docs/CUDA_Toolkit_Reference_Manual.pdf)**. Accessed: 5 December 2011.** |

**Download the Source**

To compile and run this demo, you must have the latest CUDA toolkit installed.  
You can download the source code example for this article from: