**CUDA Template documentation**

**Introduction**

The CUDA template has been developed to enable rapid prototyping of real-time GPU algorithms using OpenGL, GLSL and CUDA. Its structure resembles the original IGAD template, with the familiar ‘Init’, ‘Tick’ and ‘Shutdown’ functions. Graphics are typically produced by filling a texture using CUDA code, which is then displayed by OpenGL using a screen-filling quad. The quad can be rendered using a shader, which allows for convenient image post processing.

The template takes care of CUDA code compilation, and provides convenient classes that encapsulate CUDA modules, kernels, function arguments and host-side and device-side buffers. A debug panel is provided for ‘classic template style’ text rendering on top of the OpenGL output.

Starting a new experimentation project is generally a matter of extracting the template to a fresh folder, and modifying hostapp.h / hostapp.cpp.

**CUDAModule**

GPU code may be provided in the form of a .cu file. This file may contain an unlimited number of kernels. The .cu file needs to be compiled for a specific platform, which is all handled by the CUDAModule constructor. A typical instantiation looks as follows:

module = new CUDAModule( "programs/program.cu", "programs/program.h", "shared.h" );

The two header files are dependencies; changing these files (or the .cu file) will trigger recompilation of the binaries. Currently a maximum of 6 dependencies can be specified. The last dependency is typically ‘shared.h’, which is a header file that is also included by hostapp.cpp.

Apart from kernels, the module may contain global data, including surfaces. Example, in the .cu file:

surface<void,2> outputSurface;

Such a texture sampler can be linked to an OpenGL texture using the CUDAModule::LinkSurfaceToTexture method:

module->LinkSurfaceToTexture( "outputSurface", renderTarget );

This searches the module for a surface named “outputSurface” and links it to the OpenGL texture identified by ‘renderTarget’. Filling a texture directly from CUDA allows us to produce data that doesn’t have to be transferred back to the CPU; instead, OpenGL will directly use this data at a later stage to texture the screen filling quad.

**CUDAKernel**

A CUDAKernel instance represents a device function that can be executed by the host. It is created by specifying a function name:

kernel = new CUDAKernel( module, "testFunction" );

Kernels are tightly coupled to the module they reside in. Multiple kernels can share global data and surfaces.

To execute a kernel, we need to specify how work will be distributed over hardware resources. At the minimum, we need to specify the size of the total work set. This is done using method CUDAKernel::SetGridSize. Optionally, we can specify the block size using CUDAKernel::SetBlockSize. Note that this is optional; the default block is a 1D set of 64 threads (2 warps). For more information on warps, blocks and grids please see one of the many online resources on this topic.

**Launching Kernels**

A kernel is executed by calling its Launch method. Kernel arguments can be supplied via this method as well. The framework takes care of the translation of argument data and the transfer of this data to the GPU, so that a launch is just a single line of code.

**CUDABuffer**

Data buffers are commonly used when working with GPU algorithms. The use of a host / device setup complicates this somewhat; the CPU can only access host data, while the GPU can only access device data. The CUDABuffer class eases this significantly. A CUDABuffer instance encapsulates a host-side buffer, a device-side buffer, or both. These two buffers can be synchronized in both directions (when desired) using the CopyToHost and CopyToDevice methods. Data can be efficiently cleared on the device using the Clear method.

Creating a float buffer that only exists on the GPU:

buffer = new CUDABuffer<float>( 10000, ON\_DEVICE ); // 10k floats = 40Kb

Copying this buffer to the host automatically allocates room on the host for this data. Likewise, allocating the buffer on the host and copying it to the device allocates space on the device automatically.

A buffer can also be used as a parameter for kernels. A kernel that needs a float\* for instance simply accepts the above CUDABuffer (‘buffer’) as a parameter; the template will see to it that the proper (device) pointer is passed to the kernel.

**Shaders**

If the output of the CUDA kernels is displayed in the intended fashion (via an OpenGL texture), it can be useful to apply a shader to this rasterization step. Shaders can be provided as GLSL code. A shader program requires a vertex and a fragment shader. A basic post processing shader (with basic vignetting) is provided with the template.

**Example Code**

With the template comes some example code which you will want to remove (or adapt) for your own project. In this section we will briefly explain the implemented functionality.

In HostApp::Init, we first of all generate the OpenGL texture that will serve as render target for the CUDA code. In this case, a floating point texture is used. Its size matches the specified window resolution (see shared.h).

Next, the shader is retrieved from two source files. For most use cases, you will want to leave the vertex shader as it is. The fragment shader can either be made harmless, or you can implement your own postprocessing here.

After this, the CUDA module is loaded. This initializes the CUDA context, and loads and compiles the code that is newer than already present .cubin files for the current platform. In debug mode, the compiler will include debug symbols in the generated binary files. You may use this data in CUDA profilers to get source level data. Note that this may have a small impact on GPU code performance (~2%).

Module loading, shader instantiation and OpenGL texture creation can be done in arbitrary order; there are no data dependencies between these objects.

Once the module and the texture have been created, we can link the two using CUDAModule::LinkSurfaceToTexture. If you specified a non-existent surface, the template will give an error message here.

A pointer to a single kernel is then obtained by searching the module for it by name (“testFunction”). Again, if the kernel doesn’t exist, an error message will pop up. The particular kernel fills the texture with a simple gradient. This is a typical 2D task, so a block size of { 32, 4 } is used. Note that the block width is usually a multiple of 32, which is the size of a single warp. Here, we are assigning 4 warps to each block, which means that the registers on each SM need to be distributed over 128 threads. If the kernel requires lots of registers, { 32, 2 } may be a better choice; if the kernel requires few textures, we may opt for { 32, 8 } or { 32, 16 }. The optimal setting depends on the GPU hardware and generally needs to be found by trial and error.

The final lines of HostApp::Init create a simple timer and a debug panel. The panel is specified in ‘OpenGL coordinates’; i.e. screen coordinates that range from -1..1 over x and y. The exact number of pixels available in the DebugPanel depends on window resolution and specified coordinates. The debug panel behaves as a regular surface. It is designed to be used to display various performance statistics.

The HostApp::Tick function starts with a call to glFinish. This ensures that OpenGL is done using the resources we want to use in CUDA (in this case: the render target texture). If you are not sharing data between CUDA and OpenGL, this call is not needed.

Next, the timer is started, and then the kernel is launched. This particular kernel does not have any parameters; it simply extracts a pixel coordinate from the thread id and plots a color to the global surface we shared with the module.

Once the CUDA kernel execution has completed, we bind the shader, link the render target texture to it, set a matrix and render the quad. This is all standard OpenGL functionality.

We finish with clearing the debug panel (actually: this sets its alpha channel to 128), and printing something to it. The debug panel needs to be ‘presented’, which involves the transfer of surface pixels to the GPU and rendering a quad using these pixels.

**Problems compiling?**

The CUDA template assumes you have CUDA 8.0 installed. You may have to register as a developer with NVidia to access this version of CUDA. Alternatively, you may use CUDA 7.5; it is unlikely that the template requires any CUDA 8.0-specific features. You may need to edit the vcproj file for this: replace ‘8.0’ by ‘7.5’ and you should be good to go.

CUDA development obviously requires an NVidia GPU. If you are using a laptop with an NVidia GPU you may want to check if your system does not default to the integrated (Intel) GPU. This may cause your application to fail.

If you are using VS2015 you may find that CUDA code does not run at all. CUDA 8.0 is the first version that supports VS2015; sadly it does not support the latest VS2015 update (which you probably installed already). Getting CUDA 8.0 to work under VS2015 will be a major hassle; I recommend that you download VS2013 to (relatively) quickly resolve this. Because of this, the project files have not been tested under VS2015.

**Copyright**

The template is heavily influenced by the NVidia CUDA framework that has been distributed by several researchers.

This template may be used in any of your projects. I request that you do not distribute it by itself, but you may distribute it as part of your own open source projects. A brief e-mail about this would be appreciated.

Happy coding,

Jacco Bikker – bikker.j@gmail.com