Dr.Dobb's
THE WORLD OF SOFTWARE DEVELOPMENT

CUDA, Supercomputing for the Masses: Part 15

Using Pixel Buffer Objects with CUDA and OpenGL

January 27, 2010
URL:http://www.drdobbs.com/parallel/cuda-supercomputing-for-the-masses-part/222600097

In CUDA, Supercomputing for the Masses: Part 14 of this article series, I focused on debugging techniques and the use of CUDA-GDB to effectively diagnose and debug CUDA code -- with an emphasis on how to speed the process when looking through large amounts of data and how to use the thread syntax and semantic changes introduced in CUDA-GDB. In this article, I discuss mixing CUDA and OpenGL by utilizing a PBO (Pixel Buffer Object) to create images with CUDA on a pixel-by-pixel basis and display them using OpenGL. A subsequent article in this series will discuss the use of CUDA to generate 3D meshes and utilize OpenGL VBOs (Vertex Buffer Objects) to efficiently render meshes as a colored surface, wireframe image or set of 3D points. All demonstration code compiles and runs under both Windows and Linux.

The articles in this discussion on mixing CUDA with OpenGL cannot do more than provide a cursory introduction to OpenGL. Interested readers should look to the plethora of excellent books and tutorials that are readily available in bookstores and on the Internet. Here are a few that I have found to be useful:

- GPU Gems and NVIDIA OpenGL whitepapers.
- The OpenGL tutorials by Song Ho Ahn.
- The gamedev.net tutorials.

To focus on CUDA rather than OpenGL, I use an OpenGL framework that can mix CUDA with both pixel and vertex buffer objects. It is anticipated that this framework will be used and adapted by many others as they investigate various aspects of mixing CUDA and OpenGL not covered in my articles.

In many cases, only the CUDA kernels that generate the data will need to be modified to create and view your own content -- as will be shown in a second example at the end of this article that generates and allows interactive movement over an artificial landscape. Finally, this same framework will be used in the next article with minor modifications to discuss and demonstrate vertex buffer objects.

In a nutshell, creating a working OpenGL application requires the following steps that are instantiated through the files in the framework as illustrated in the schematic below:

1. `simpleGLmain.cpp`: Create an OpenGL window and performs basic OpenGl/GLUT setup.
2. `simplePBO.cpp`: Perform CUDA-centric setup; in this case for a Pixel Buffer Object (PBO).
3. `callbacksPBO.cpp`: Define keyboard, mouse, and other callbacks.
4. `kernelPBO.cu`: The CUDA kernel that calculates the data to be displayed.

I anticipate that many readers will just copy and paste these four files and build the example. This is fine. Similarly, many readers will also cut and paste the additional two files, perlinCallbacksPBO.cpp and perlinKernelPBO.cu used in the second example to see the artificial landscape in action.

For many, working with the source code of these two examples will be sufficient (along with the comments) to provide the basic visualization functionality needed for their work or to establish a known-working code base that can be leveraged and adapted to create other more advanced CUDA applications.

Following is the source code combined with a discussion of the essential features needed to combine CUDA and OpenGL in the same application to create images.

Interested readers might also like to watch the video of Joe Stam's presentation given at the 2009 NVIDIA GTC (GPU Technology Conference) entitled What Every CUDA Programmer Needs to Know about OpenGL. Joe's presentation discussed many of the OpenGL concepts covered in my articles and provides a live demonstration of the simplest PBO and VBO demonstrations from this and the follow-on article.

Framework and Rational for Combining CUDA and OpenGL

Just as in the NVIDIA SDK samples, GLUT (a window system independent OpenGL Toolkit) was utilized for Windows and Linux compatibility. Figure 1 illustrates the relationship between the four files used in the framework.

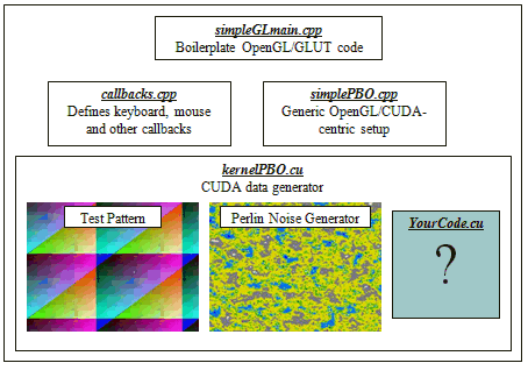

Figure 1

As we will see, CUDA and OpenGL interoperability is very fast!

The reason (aside from the speed of CUDA) is that CUDA maps OpenGL buffer(s) into the CUDA memory space with a call to `cudaGLMapBufferObject()`. On a single GPU system, no data movement is required! Once provided with a pointer, CUDA programmers are then free to exploit their knowledge of CUDA to write fast and efficient kernels that operate on the mapped OpenGL buffers. However, the separation between OpenGL and CUDA is distinct so OpenGL should not operate on any buffer while it is mapped into the CUDA memory space.

There are two very clear benefits of the separation (yet efficient interoperability) between CUDA and OpenGL:

- From a programming view: When not mapped into the CUDA memory space, OpenGL gurus are free to exploit existing legacy code bases, their expertise and the full power of all the tools available to them such as GLSL (the OpenGL Shading Language) and Cg.
- From an investment view: Efficient exploitation of existing legacy OpenGL software investments is probably the most important benefit this mapped approach provides. Essentially, CUDA code can be gradually added into existing legacy libraries and applications just by mapping the buffer into the CUDA memory space. This allows organizations to test CUDA code without significant risk and then enjoy the benefits once they are confident in the performance and productivity rewards delivered by this programming model.

From Window Creation to CUDA Buffer Registration

Before mapping the OpenGL buffer to CUDA, the following steps must be taken:

1. Create a window (OS specific)
2. Create a GL context (also OS specific)
3. Set up the GL viewport and coordinate system
4. Generate one or more GL buffers to be shared with CUDA
5. Register these buffers with CUDA

Figure 2 illustrates these steps.

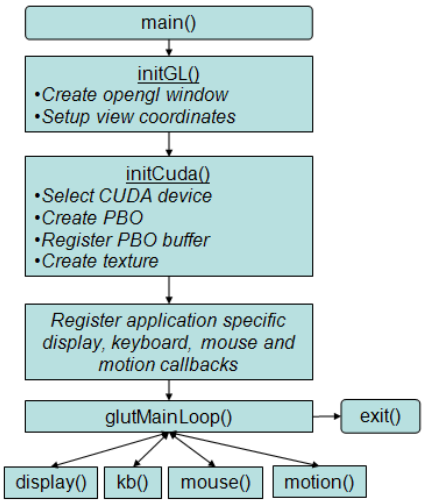

Figure 2

Let's begin walking quickly through simpleGLmain.cpp to see how this happens. Prior to the definition of :

- Various OpenGL and CUDA files are included to provide access to needed functionality and interoperability.
- References to external user-defined methods are specified.
- The size of the window is specified as 512x512 and a timer used for the frames per second calculation is created and set to zero.
- A forward declaration to `initGL()` is specified to simplify the discussion in this article.
- The routines `fpsDisplay()` and `computeFPS()` are defined to use a timer and display the calculated frames per second in the window title so we can get a sense of how fast our code is running. (Note: Windows users may need to disable vsync to see full performance because buffer swapping normally occurs at the vertical refresh interval to avoid tearing (commonly 60 hz).V-sync can be turned off in the control panel when benchmarking.)

```
// simpleGLmain.cpp (Rob Farber)

// includes
#include <GL/glew.h>
#include <cuda_runtime.h>
#include <cutil_inline.h>
#include <cutil_gl_inline.h>
#include <cutil_gl_error.h>
#include <cuda_gl_interop.h>
#include <rendercheck_gl.h>

// The user must create the following routines:
// CUDA methods
extern void initCuda(int argc, char** argv);
extern void runCuda();
extern void renderCuda(int);

// callbacks
extern void display();
extern void keyboard(unsigned char key, int x, int y);
extern void mouse(int button, int state, int x, int y);
extern void motion(int x, int y);

// GLUT specific variables
```

```
unsigned int window_width = 512;
unsigned int window_height = 512;

unsigned int timer = 0; // a timer for FPS calculations

// Forward declarations of GL functionality
CUTBoolean initGL(int argc, char** argv);

// Simple method to display the Frames Per Second in the window title
void computeFPS()
{
  static int fpsCount=0;
  static int fpsLimit=100;

  fpsCount++;

  if (fpsCount == fpsLimit) {
    char fps[256];
    float ifps = 1.f / (cutGetAverageTimerValue(timer) / 1000.f);
    sprintf(fps, "Cuda GL Interop Wrapper: %3.1f fps ", ifps);

    glutSetWindowTitle(fps);
    fpsCount = 0;

    cutilCheckError(cutResetTimer(timer));
  }
}

void fpsDisplay()
{
  cutilCheckError(cutStartTimer(timer));

  display();

  cutilCheckError(cutStopTimer(timer));
  computeFPS();
}

// Main program
int main(int argc, char** argv)
{
  // Create the CUTIL timer
  cutilCheckError( cutCreateTimer( &timer));

  if (CUTFalse == initGL(argc, argv)) {
    return CUTFalse;
  }

  initCuda(argc, argv);
  CUT_CHECK_ERROR_GL();

  // register callbacks
  glutDisplayFunc(fpsDisplay);
  glutKeyboardFunc(keyboard);
  glutMouseFunc(mouse);
  glutMotionFunc(motion);

  // start rendering mainloop
  glutMainLoop();

  // clean up
  cudaThreadExit();
  cutilExit(argc, argv);
}

CUTBoolean initGL(int argc, char **argv)
{
  //Steps 1-2: create a window and GL context (also register callbacks)
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE);
  glutInitWindowSize(window_width, window_height);
  glutCreateWindow("Cuda GL Interop Demo (adapted from NVIDIA's simpleGL");
  glutDisplayFunc(fpsDisplay);
  glutKeyboardFunc(keyboard);
  glutMotionFunc(motion);

  // check for necessary OpenGL extensions
  glewInit();
  if (! glewIsSupported( "GL_VERSION_2_0 " ) ) {
    fprintf(stderr, "ERROR: Support for necessary OpenGL extensions missing.");
    return CUTFalse;
  }

  // Step 3: Setup our viewport and viewing modes
  glViewport(0, 0, window_width, window_height);

  glClearColor(0.0, 0.0, 0.0, 1.0);
  glDisable(GL_DEPTH_TEST);


  // set view matrix
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glOrtho(0.0f, 1.0f, 0.0f, 1.0f, 0.0f, 1.0f);

  return CUTTrue;
}
```

As you can see in the `main()` routine, a timer is first initialized and the routine `initGL()` is called. As mentioned previously, GLUT is utilized to create the window in a portable

fashion for Windows and Linux users. (It is likely that these same callbacks and CUDA code can be adapted to work other windowing systems as well.)

Without going into too much detail, we see the GLUT initialization of the window and registration of the callbacks:

```
//Steps 1-2: create a window and GL context (also register callbacks)
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE);
  glutInitWindowSize(window_width, window_height);
  glutCreateWindow("Cuda GL Interop Demo (adapted from NVIDIA's simpleGL");
  glutDisplayFunc(fpsDisplay);
  glutKeyboardFunc(keyboard);
  glutMotionFunc(motion);
```

It is also important to check that the OpenGL version supports the capabilities and extensions needed to run the application. The cross-platform GLEW (OpenGL Extension Wrangler Library) library is used to load this functionality and expose it to the user. The check occurs through the call to `glewIsSupported()`.

```
  // check for necessary OpenGL extensions
  glewInit();
  if (! glewIsSupported( "GL_VERSION_2_0 " ) ) {
    fprintf(stderr, "ERROR: Support for necessary OpenGL extensions missing.");
    return CUTFalse;
  }
```

Now we setup our viewport, which defines which portion of the window is to be used; clear the color and viewport; and disable depth sorting

```
// Step 3: Setup our viewport and viewing modes
  glViewport(0, 0, window_width, window_height);

  glClearColor(0.0, 0.0, 0.0, 1.0);
  glDisable(GL_DEPTH_TEST);
```

Finally, we specify a simple orthogonal view and define the OpenGL coordinates.

```
  // set view matrix
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glOrtho(0.0f, 1.0f, 0.0f, 1.0f, 0.0f, 1.0f);
```

For more detailed information about the OpenGL coordinates, viewing, and transforms used thus far, I recommend one of the many excellent resources available on the web including the tutorial by Song Ho Ann.

Steps 4 and 5 occur via the call to `initCuda()`, which defines the CUDA and OpenGL buffers and interoperability contexts. The CUDA/OpenGL interoperability functions are defined for the runtime API with the line:

```
#include <cuda_gl_interop.h>
```

The routine `initCuda()` is defined in simplePBO.cpp, which maximized flexibility of the framework by isolating simpleGLmain.cpp from any special application requirements such as changing the names and numbers of the PBOs.

```
void initCuda(int argc, char** argv)
{
  // First initialize OpenGL context, so we can properly set the GL
  // for CUDA.  NVIDIA notes this is necessary in order to achieve
  // optimal performance with OpenGL/CUDA interop.  use command-line
  // specified CUDA device, otherwise use device with highest Gflops/s
  if( cutCheckCmdLineFlag(argc, (const char**)argv, "device") ) {
    cutilGLDeviceInit(argc, argv);
  } else {
    cudaGLSetGLDevice( cutGetMaxGflopsDeviceId() );
  }

  createPBO(&pbo);
  createTexture(&textureID,image_width,image_height);

  // Clean up on program exit
  atexit(cleanupCuda);

  runCuda();
}
```

As noted in the source code, `initCuda()` sets the CUDA and OpenGL contexts to get the best performance for the given hardware configuration.

The routine `createPBO()` is called to create the OpenGL buffer(s) plus a texture is created via `createTexture()` that can be used for rendering. We also perform some housekeeping so all our buffers and textures get cleaned up on program exit.

As can be seen in `createPBO()`, we finally complete steps 4 and 5 by generating the buffer through a call to `glGenBuffers()`, and binding it with `glBindBuffer()`, and registering it for use with CUDA with `cudaGLRegisterBufferObject()`. Please note that the opengl call to `glGenBuffers()` call performs the actual memory allocation. Since the data pointer is NULL, the data is just allocated and not initialized.

```
void createPBO(GLuint* pbo)
```

```
{

  if (pbo) {
    // set up vertex data parameter
    int num_texels = image_width * image_height;
    int num_values = num_texels * 4;
    int size_tex_data = sizeof(GLubyte) * num_values;

    // Generate a buffer ID called a PBO (Pixel Buffer Object)
    glGenBuffers(1,pbo);
    // Make this the current UNPACK buffer (OpenGL is state-based)
    glBindBuffer(GL_PIXEL_UNPACK_BUFFER, *pbo);
    // Allocate data for the buffer. 4-channel 8-bit image
    glBufferData(GL_PIXEL_UNPACK_BUFFER, size_tex_data, NULL, GL_DYNAMIC_COPY);
    cudaGLRegisterBufferObject( *pbo );
  }
}
```

Completing our discussion of `main()`, we see that the main GLUT main loop is called. Once that completes, we clean up and exit both `main()` and the application.

```
// start rendering mainloop
  glutMainLoop();

  // clean up
  cudaThreadExit();
  cutilExit(argc, argv);
}
```

This completes our discussion of simpleGLmain.cpp. For more information, please refer to the GLUT and GLEW documentation as well as one of the many excellent OpenGL references on the Internet.

Here is the complete source code for simplePBO.cpp.

```
// simplePBO.cpp (Rob Farber)

// includes
#include <GL/glew.h>
#include <cuda_runtime.h>
#include <cutil_inline.h>
#include <cutil_gl_inline.h>
#include <cuda_gl_interop.h>
#include <rendercheck_gl.h>

// external variables
extern float animTime;
extern unsigned int window_width;
extern unsigned int window_height;

// constants (the following should be a const in a header file)
unsigned int image_width = window_width;
unsigned int image_height = window_height;

extern "C" void launch_kernel(uchar4* , unsigned int, unsigned int, float);

// variables
GLuint pbo=NULL;
GLuint textureID=NULL;

void createPBO(GLuint* pbo)
{

  if (pbo) {
    // set up vertex data parameter
    int num_texels = image_width * image_height;
    int num_values = num_texels * 4;
    int size_tex_data = sizeof(GLubyte) * num_values;

    // Generate a buffer ID called a PBO (Pixel Buffer Object)
    glGenBuffers(1,pbo);
    // Make this the current UNPACK buffer (OpenGL is state-based)
    glBindBuffer(GL_PIXEL_UNPACK_BUFFER, *pbo);
    // Allocate data for the buffer. 4-channel 8-bit image
    glBufferData(GL_PIXEL_UNPACK_BUFFER, size_tex_data, NULL, GL_DYNAMIC_COPY);
    cudaGLRegisterBufferObject( *pbo );
  }
}

void deletePBO(GLuint* pbo)
{
  if (pbo) {
    // unregister this buffer object with CUDA
    cudaGLUnregisterBufferObject(*pbo);

    glBindBuffer(GL_ARRAY_BUFFER, *pbo);
    glDeleteBuffers(1, pbo);

    *pbo = NULL;
  }
}

void createTexture(GLuint* textureID, unsigned int size_x, unsigned int size_y)
{
  // Enable Texturing
  glEnable(GL_TEXTURE_2D);

  // Generate a texture identifier
  glGenTextures(1,textureID);
```

```
  // Make this the current texture (remember that GL is state-based)
  glBindTexture( GL_TEXTURE_2D, *textureID);

  // Allocate the texture memory. The last parameter is NULL since we only
  // want to allocate memory, not initialize it
  glTexImage2D( GL_TEXTURE_2D, 0, GL_RGBA8, image_width, image_height, 0,
                      GL_BGRA,GL_UNSIGNED_BYTE, NULL);

  // Must set the filter mode, GL_LINEAR enables interpolation when scaling
  glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
  glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
  // Note: GL_TEXTURE_RECTANGLE_ARB may be used instead of
  // GL_TEXTURE_2D for improved performance if linear interpolation is
  // not desired. Replace GL_LINEAR with GL_NEAREST in the
  // glTexParameteri() call

}

void deleteTexture(GLuint* tex)
{
    glDeleteTextures(1, tex);

    *tex = NULL;
}

void cleanupCuda()
{
  if(pbo) deletePBO(&pbo);
  if(textureID) deleteTexture(&textureID);
}

// Run the Cuda part of the computation
void runCuda()
{
  uchar4 *dptr=NULL;

  // map OpenGL buffer object for writing from CUDA on a single GPU
  // no data is moved (Win & Linux). When mapped to CUDA, OpenGL
  // should not use this buffer
  cudaGLMapBufferObject((void**)&dptr, pbo);

  // execute the kernel
  launch_kernel(dptr, image_width, image_height, animTime);

  // unmap buffer object
  cudaGLUnmapBufferObject(pbo);
}

void initCuda(int argc, char** argv)
{
  // First initialize OpenGL context, so we can properly set the GL
  // for CUDA.  NVIDIA notes this is necessary in order to achieve
  // optimal performance with OpenGL/CUDA interop.  use command-line
  // specified CUDA device, otherwise use device with highest Gflops/s
  if( cutCheckCmdLineFlag(argc, (const char**)argv, "device") ) {
    cutilGLDeviceInit(argc, argv);
  } else {
    cudaGLSetGLDevice( cutGetMaxGflopsDeviceId() );
  }

  createPBO(&pbo);
  createTexture(&textureID,image_width,image_height);

  // Clean up on program exit
  atexit(cleanupCuda);

  runCuda();
}
```

Aside from the previously discussed routine `initCuda()`, we have several straightforward housekeeping routines `cleanupCuda()`, `deletePBO()`, and `deleteTexture()` that will not be discussed in this article. Please refer to the CUDA documentation for more information about the calls used in these routines.

Final Steps to Render an Image from a CUDA Application

To draw an image from a CUDA application requires the following steps:

1. Allocate OpenGL buffer(s) that are the size of the image.
2. Allocate OpenGL texture(s) that are the size of the image.
3. Map OpenGL buffer(s) to CUDA memory.
4. Write the image from CUDA to the mapped OpenGL buffer(s).
5. Unmap the OpenGL buffer(s).
6. Bind the texture to the OpenGL buffer.
7. Draw a Quad that specifies the texture coordinates at each corner.
8. Swap front and back buffers to draw to the display.

As previously discussed, `createPBO()` allocates the OpenGL PBO buffer(s) with `glBufferData()`, thereby fulfilling step 1. Similarly, `createTexture()` allocates the OpenGL texture(s) specified in step 2 that can be used for rendering the image to the display.

The routine `display()` calls the CUDA kernel that creates or modifies the data in the OpenGL buffer, then renders the new image to the screen as in Figure 3:

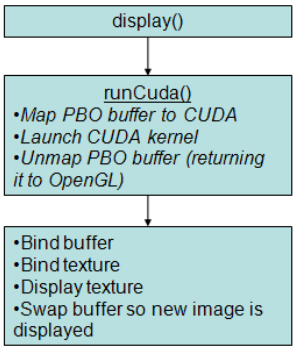

Figure 3

As you can see in Figure 3, the routine `runCuda()` performs steps 3-5 by mapping the OpenGL buffer to CUDA through the call to `cudaGLMapBufferObject()`, As can be seen below, `runCuda()` maps the OpenGL buffer, launches the CUDA kernel with `launch_kernel()` (discussed later in this article) and unmaps the buffer with `cudaGLUnmapBufferObject()`.

```
// Run the Cuda part of the computation
void runCuda()
{
  uchar4 *dptr=NULL;

  // map OpenGL buffer object for writing from CUDA on a single GPU
  // no data is moved (Win & Linux). When mapped to CUDA, OpenGL
  // should not use this buffer
  cudaGLMapBufferObject((void**)&dptr, pbo);

  // execute the kernel
  launch_kernel(dptr, image_width, image_height, animTime);

  // unmap buffer object
  cudaGLUnmapBufferObject(pbo);
}
```

It is important to clarify the distinction between registering and mapping an OpenGL buffer. Registering sets up the buffer for CUDA/OpenGL interoperability, but doesn't actually make the buffer available to CUDA. Registering a buffer is expensive, so it is only done at creation. Mapping, in contrast, actually hands access of the memory over to CUDA through a pointer. Mapping/unmapping operations are very fast these calls can be used in frequently called sections of code.

Finally, steps 6-8 occur in the callback method `display()`, contained in the file callbacksPBO.cpp listed below:

```
//callbacksPBO.cpp (Rob Farber)

#include <GL/glew.h>
#include <cuda_runtime.h>
#include <cutil_inline.h>
#include <cutil_gl_inline.h>
#include <cuda_gl_interop.h>
//#include <cutil_gl_error.h>
#include <rendercheck_gl.h>

// variables for keyboard control
int animFlag=1;
float animTime=0.0f;
float animInc=0.1f;

//external variables
extern GLuint pbo;
extern GLuint textureID;
extern unsigned int image_width;
extern unsigned int image_height;

// The user must create the following routines:
void runCuda();

void display()
{
  // run CUDA kernel
  runCuda();

  // Create a texture from the buffer
  glBindBuffer( GL_PIXEL_UNPACK_BUFFER, pbo);

  // bind texture from PBO
  glBindTexture(GL_TEXTURE_2D, textureID);


  // Note: glTexSubImage2D will perform a format conversion if the
  // buffer is a different format from the texture. We created the
  // texture with format GL_RGBA8. In glTexSubImage2D we specified
  // GL_BGRA and GL_UNSIGNED_INT. This is a fast-path combination

  // Note: NULL indicates the data resides in device memory
  glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, image_width, image_height,
                  GL_RGBA, GL_UNSIGNED_BYTE, NULL);


  // Draw a single Quad with texture coordinates for each vertex.
```

```
  glBegin(GL_QUADS);
  glTexCoord2f(0.0f,1.0f); glVertex3f(0.0f,0.0f,0.0f);
  glTexCoord2f(0.0f,0.0f); glVertex3f(0.0f,1.0f,0.0f);
  glTexCoord2f(1.0f,0.0f); glVertex3f(1.0f,1.0f,0.0f);
  glTexCoord2f(1.0f,1.0f); glVertex3f(1.0f,0.0f,0.0f);
  glEnd();

  // Don't forget to swap the buffers!
  glutSwapBuffers();

  // if animFlag is true, then indicate the display needs to be redrawn
  if(animFlag) {
    glutPostRedisplay();
    animTime += animInc;
  }
}

//! Keyboard events handler for GLUT
void keyboard(unsigned char key, int x, int y)
{
  switch(key) {
  case(27) :
    exit(0);
    break;
  case 'a': // toggle animation
  case 'A':
    animFlag = (animFlag)?0:1;
    break;
  case '-': // decrease the time increment for the CUDA kernel
    animInc -= 0.01;
    break;
  case '+': // increase the time increment for the CUDA kernel
    animInc += 0.01;
    break;
  case 'r': // reset the time increment
    animInc = 0.01;
    break;
  }

  // indicate the display must be redrawn
  glutPostRedisplay();
}

// No mouse event handlers defined
void mouse(int button, int state, int x, int y)
{
}

void motion(int x, int y)
{
}
```

The routine `display()` calls `runCuda()` (which, as we discussed, completes all of steps 1-5). The following code segment after the call to `runCuda()` binds a texture to the buffer as is required for step 6.

```
// Create a texture from the buffer
  glBindBuffer( GL_PIXEL_UNPACK_BUFFER, pbo);

  // bind texture from PBO
  glBindTexture(GL_TEXTURE_2D, textureID);


  // Note: glTexSubImage2D will perform a format conversion if the
  // buffer is a different format from the texture. We created the
  // texture with format GL_RGBA8. In glTexSubImage2D we specified
  // GL_BGRA and GL_UNSIGNED_INT. This is a fast-path combination

  // Note: NULL indicates the data resides in device memory
  glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, image_width, image_height,
                  GL_RGBA, GL_UNSIGNED_BYTE, NULL);
```

Finally, we can now draw the image!

The following code segment actually draws the image even though (at first glance) it looks like only a square is being drawn! It actually draws a single quad with the texture coordinates for each vertex. (In a sense, the 2D texture is "stretched" and "held" at each of the four corners of the 3D quad as if it were a piece of fabric suspended by four poles.) This is actually a very simple example of texture mapping.

To be more precise, the texture is not "created" by `glTexSubImage2d()`, rather the data is copied from the buffer to the texture, and then since texturing is enabled and our texture is bound the quad we draw has the texture "glued" on it.

```
// Draw a single Quad with texture coordinates for each vertex.

  glBegin(GL_QUADS);
  glTexCoord2f(0.0f,1.0f); glVertex3f(0.0f,0.0f,0.0f);
  glTexCoord2f(0.0f,0.0f); glVertex3f(0.0f,1.0f,0.0f);
  glTexCoord2f(1.0f,0.0f); glVertex3f(1.0f,1.0f,0.0f);
  glTexCoord2f(1.0f,1.0f); glVertex3f(1.0f,0.0f,0.0f);
  glEnd();
```

More complex mappings are clearly possible such as mapping a texture onto a sphere as in Figure 4.

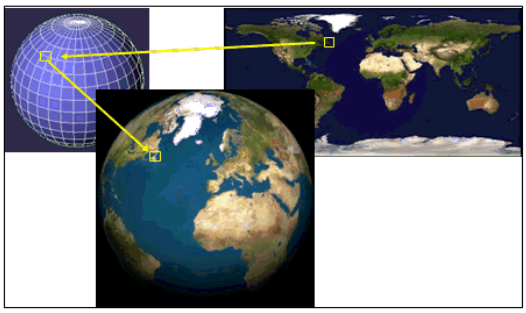

Figure 4

All drawing is performed in an off-screen framebuffer. Once all drawing is completed, the front and back buffers are swapped to make the new image visible. This use of double-buffering prevents visual artifacts from marring the image while the image is being updated.

The remaining callbacks in the file callbackPBO.cpp are straightforward and will not be discussed. Please note that this example does not perform any mouse or motion related activities so the routines `mouse()` and `motion()` do nothing. The empty routines are placeholders that will be used in the VBO examples in the next article of this series and permit linking of the executable.

A few simple keyboard commands are defined in the `keyboard()` routine:

- `<ESC>` terminates the application.
- `a` or `A` toggles the animation. Stopping the animation will freeze the picture and stop the frames per second calculation.
- `+` and `-` increases/decreases the animation time increment.
- `r` resets the time increment.

While all this might seem overly complicated for our simple example, please remember that OpenGL is general purpose and quite powerful and was designed to enable graphics applications that are far more challenging than the simple code provided here. As mentioned at the beginning of this article, we have done little more than pay cursory attention to the capabilities and features within OpenGL. Interested readers are strongly encouraged to look at the references provided at the beginning of this article -- especially the GPU Gems series of books.

The source code for the CUDA test pattern kernel is extremely simple as can be seen in the method `kernel()` below:

```
//kernelPBO.cu (Rob Farber)

#include <stdio.h>

void checkCUDAError(const char *msg) {
  cudaError_t err = cudaGetLastError();
  if( cudaSuccess != err) {
    fprintf(stderr, "Cuda error: %s: %s.\n", msg, cudaGetErrorString( err) );
    exit(EXIT_FAILURE);
  }
}

//Simple kernel writes changing colors to a uchar4 array
__global__ void kernel(uchar4* pos, unsigned int width, unsigned int height,
                       float time)
{
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  unsigned int x = index%width;
  unsigned int y = index/width;

  if(index < width*height) {
    unsigned char r = (x + (int) time)&0xff;
    unsigned char g = (y + (int) time)&0xff;
    unsigned char b = ((x+y) + (int) time)&0xff;

    // Each thread writes one pixel location in the texture (textel)
    pos[index].w = 0;
    pos[index].x = r;
    pos[index].y = g;
    pos[index].z = b;
  }
}

// Wrapper for the __global__ call that sets up the kernel call
extern "C" void launch_kernel(uchar4* pos, unsigned int image_width,
                              unsigned int image_height, float time)
{
  // execute the kernel
  int nThreads=256;
  int totalThreads = image_height * image_width;
  int nBlocks = totalThreads/nThreads;
  nBlocks += ((totalThreads%nThreads)>0)?1:0;

  kernel<<< nBlocks, nThreads>>>(pos, image_width, image_height, time);

  // make certain the kernel has completed
  cudaThreadSynchronize();

  checkCUDAError("kernel failed!");
}
```

The `launch_kernel()` routine performs any setup prior to starting the method `kernel()` on the GPU. For this example only the kernel execution configuration is specified and a few simple parameters are calculated so they can be passed to the GPU.

The actual kernel for this example, `kernel()`, is extremely simple as can be seen above. Each thread assigns an RGB value to each location within the PBO based on `animTime` and `grid` coordinates.

It is important to note that `cudaThreadSynchronize()` is called in `launch_kernel()` after the kernel execution to ensure all GPU work has completed before control is returned to the host.

Under Linux the executable testpattern can be created with the following command. (Please substitute the path to the CUDA SDK libraries for CUDA_LIBRARIES. Similarly, substitute the path to the CUDA SDK include files for CUDA_INCLUDES.)

```
nvcc –O3 -L CUDA_LIBRARIES -I CUDA_INCLUDES simpleGLmain.cpp simplePBO.cpp callbacksPBO.cpp kernelPBO.cu -lglut -lGLEW –lcutil –o testpattern
```

Running the testpattern executable displays a colorful animated test pattern that changes in color over time. Something like Figure 5 should appear on the screen.

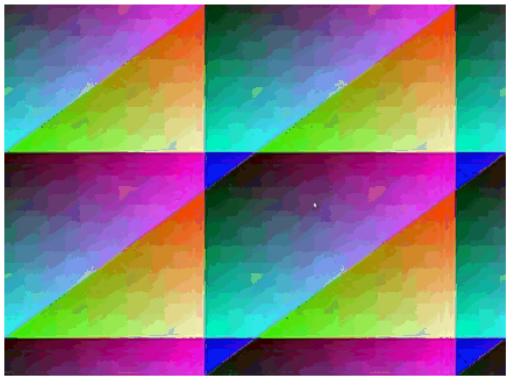

Figure 5

Creating an Entire Artificial Landscape

Now let's have some fun and exploit the flexibility of CUDA and the OpenGL framework described in this article!

In 1997, Ken Perlin received an Academy Award for developing the Perlin Noise generator. Perlin Noise has a multitude of uses ranging from the creation of natural textures (Texture demo) to artificial terrain and even worlds (Planet demo).

In this article, I implement a PBO-based CUDA version of his reference Java implementation of the Improved Perlin Noise generator to generate animated images of artificial terrain. The next article in this series will use the same code to create fully rendered 3D landscapes using this noise function.

Numerous websites on the Internet discuss Perlin Noise. It is a popular topic. Here are several of my favorites:

- Ken Perlin's homepage
- Hugo Elias has a nice web page about Perlin Noise
- Chapter 26 of GPU Gems 2

Following is the complete source of the file, perlinKernelPBO.cu. Please note that there is a bank conflict in this code that will be found in a later article on performance tools. Can you find it? Even so, the provided CUDA version of this code is extremely fast.

```
//perlinKernelPBO.cu (Rob Farber)
#include <cutil_math.h>
#include <cutil_inline.h>
#include <cutil_gl_inline.h>
#include <cuda_gl_interop.h>

float gain=0.75f;
float xStart=2.f;
float yStart=1.f;
float zOffset = 0.0f;
#define Z_PLANE 50.f

__constant__ unsigned char c_perm[256];
__shared__ unsigned char s_perm[256]; // shared memory copy of permuation array
unsigned char* d_perm=NULL; // global memory copy of permutation array
// host version of permutation array
const static unsigned char h_perm[] = {151,160,137,91,90,15,
   131,13,201,95,96,53,194,233,7,225,140,36,103,30,69,142,8,99,37,240,21,10,23,
   190, 6,148,247,120,234,75,0,26,197,62,94,252,219,203,117,35,11,32,57,177,33,
   88,237,149,56,87,174,20,125,136,171,168, 68,175,74,165,71,134,139,48,27,166,
   77,146,158,231,83,111,229,122,60,211,133,230,220,105,92,41,55,46,245,40,244,
   102,143,54, 65,25,63,161, 1,216,80,73,209,76,132,187,208, 89,18,169,200,196,
   135,130,116,188,159,86,164,100,109,198,173,186, 3,64,52,217,226,250,124,123,
   5,202,38,147,118,126,255,82,85,212,207,206,59,227,47,16,58,17,182,189,28,42,
   223,183,170,213,119,248,152,2,44,154,163, 70,221,153,101,155,167, 43,172,9,
   129,22,39,253, 19,98,108,110,79,113,224,232,178,185, 112,104,218,246,97,228,
   251,34,242,193,238,210,144,12,191,179,162,241, 81,51,145,235,249,14,239,107,
   49,192,214, 31,181,199,106,157,184,84,204,176,115,121,50,45,127, 4,150,254,
   138,236,205,93,222,114,67,29,24,72,243,141,128,195,78,66,215,61,156,180
   };

__device__ inline int perm(int i) { return(s_perm[i&0xff]); }
__device__ inline float fade(float t) { return t * t * t * (t * (t * 6.f - 15.f) + 10.f); }
__device__ inline float lerpP(float t, float a, float b) { return a + t * (b - a); }
__device__ inline float grad(int hash, float x, float y, float z) {
  int h = hash & 15;                      // CONVERT LO 4 BITS OF HASH CODE
```

```
  float u = h<8 ? x : y,                 // INTO 12 GRADIENT DIRECTIONS.
    v = h<4 ? y : h==12||h==14 ? x : z;
  return ((h&1) == 0 ? u : -u) + ((h&2) == 0 ? v : -v);
}

__device__ float inoise(float x, float y, float z) {
  int X = ((int)floorf(x)) & 255, // FIND UNIT CUBE THAT
    Y = ((int)floorf(y)) & 255,   // CONTAINS POINT.
    Z = ((int)floorf(z)) & 255;
  x -= floorf(x);                // FIND RELATIVE X,Y,Z
  y -= floorf(y);                // OF POINT IN CUBE.
  z -= floorf(z);
  float u = fade(x),             // COMPUTE FADE CURVES
    v = fade(y),                 // FOR EACH OF X,Y,Z.
    w = fade(z);
  int A = perm(X)+Y, AA = perm(A)+Z, AB = perm(A+1)+Z, // HASH COORDINATES OF
    B = perm(X+1)+Y, BA = perm(B)+Z, BB = perm(B+1)+Z; // THE 8 CUBE CORNERS,

  return lerpP(w, lerpP(v, lerpP(u, grad(perm(AA), x  , y  , z   ), // AND ADD
                                 grad(perm(BA), x-1.f, y  , z   )),   // BLENDED
                         lerpP(u, grad(perm(AB), x  , y-1.f, z   ),    // RESULTS
                                grad(perm(BB), x-1.f, y-1.f, z   ))),     // FROM  8
                 lerpP(v, lerpP(u, grad(perm(AA+1), x  , y  , z-1.f ),  // CORNERS
                                grad(perm(BA+1), x-1.f, y  , z-1.f )),    // OF CUBE
                         lerpP(u, grad(perm(AB+1), x  , y-1.f, z-1.f ),
                                grad(perm(BB+1), x-1.f, y-1.f, z-1.f ))));
#ifdef ORIG
  return(perm(X));
#endif

}

__device__ inline float height2d(float x, float y, int octaves,
                     float lacunarity = 2.0f, float gain = 0.5f)
{
  float freq = 1.0f, amp = 0.5f;
  float sum = 0.f;
  for(int i=0; i<octaves; i++) {
    sum += inoise(x*freq,y*freq, Z_PLANE)*amp;
    freq *= lacunarity;
    amp *= gain;
  }
  return sum;
}

__device__ inline uchar4 colorElevation(float texHeight)
{
  uchar4 pos;

  // color textel (r,g,b,a)
       if (texHeight < -1.000f) pos = make_uchar4(000, 000, 128, 255); //deeps
  else if (texHeight < -.2500f) pos = make_uchar4(000, 000, 255, 255); //shallow
  else if (texHeight < 0.0000f) pos = make_uchar4(000, 128, 255, 255); //shore
  else if (texHeight < 0.0625f) pos = make_uchar4(240, 240, 064, 255); //sand
  else if (texHeight < 0.1250f) pos = make_uchar4(032, 160, 000, 255); //grass
  else if (texHeight < 0.3750f) pos = make_uchar4(224, 224, 000, 255); //dirt
  else if (texHeight < 0.7500f) pos = make_uchar4(128, 128, 128, 255); //rock
  else                          pos = make_uchar4(255, 255, 255, 255); //snow

  return(pos);
}

void checkCUDAError(const char *msg) {
  cudaError_t err = cudaGetLastError();
  if( cudaSuccess != err) {
    fprintf(stderr, "Cuda error: %s: %s.\n", msg, cudaGetErrorString( err) );
    exit(EXIT_FAILURE);
  }
}

//Simple kernel fills an array with perlin noise
__global__ void k_perlin(uchar4* pos, unsigned int width, unsigned int height,
                         float2 start, float2 delta, float gain, float zOffset,
                         unsigned char* d_perm)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  float xCur = start.x + ((float) (idx%width)) * delta.x;
  float yCur = start.y + ((float) (idx/width)) * delta.y;

  if(threadIdx.x < 256)
    // Optimization: this causes bank conflicts
    s_perm[threadIdx.x] = d_perm[threadIdx.x];
  // this synchronization can be important if there are more that 256 threads
  __syncthreads();


  // Each thread creates one pixel location in the texture (textel)
  if(idx < width*height) {
    float h = height2d(xCur, yCur, 4, 2.f, 0.75f) + zOffset;

    pos[idx] = colorElevation(h);
  }
}


// Wrapper for the __global__ call that sets up the kernel call
extern "C" void launch_kernel(uchar4* pos, unsigned int image_width,
                              unsigned int image_height, float time)
{
  int nThreads=256; // must be equal or larger than 256! (see s_perm)
  int totalThreads = image_height * image_width;
  int nBlocks = totalThreads/nThreads;
  nBlocks += ((totalThreads%nThreads)>0)?1:0;
```

```
  float xExtent = 10.f;
  float yExtent = 10.f;
  float xDelta = xExtent/(float)image_width;
  float yDelta = yExtent/(float)image_height;

  if(!d_perm) { // for convenience allocate and copy d_perm here
    cudaMalloc((void**) &d_perm,sizeof(h_perm));
    cudaMemcpy(d_perm,h_perm,sizeof(h_perm),cudaMemcpyHostToDevice);
    checkCUDAError("d_perm malloc or copy failed!");
  }

  k_perlin<<< nBlocks, nThreads>>>(pos, image_width, image_height,
                                   make_float2(xStart, yStart),
                                   make_float2(xDelta, yDelta),
                                   gain, zOffset, d_perm);

  // make certain the kernel has completed
  cudaThreadSynchronize();
  checkCUDAError("kernel failed!");
}
```

We also define several additional keystrokes in callbacksPerlin.cpp that allow interactively movement around the artificial landscape using "`vi`" keystrokes (e.g., `h, j, k,` and `l`). The + and – keys are also redefined from to vary the sea level of the artificial landscape:

- `<ESC>` or `q` terminates the application.
- `A` or `a` toggles the animation. Stopping the animation will freeze the picture and stop the frames per second calculation.
- The + key raises the terrain (effectively lowering the sea level).
- The – key lowers the terrain (effectively increasing the sea level).
- `h` moves left.
- `l` moves right.
- `k` moves up.
- `j` moves down.

The complete source for callbacksPerlin.cpp follows. (Note: If no additional keyboard commands had been defined, the original callbacksPBO.cpp can be used unchanged.)

```
//perlinCallbacksPBO.cpp (Rob Farber)
#include <GL/glew.h>
#include <cuda_runtime.h>
#include <cutil_inline.h>
#include <cutil_gl_inline.h>
#include <cuda_gl_interop.h>
//#include <cutil_gl_error.h>
#include <rendercheck_gl.h>

// variables for keyboard control
int animFlag=1;
float animTime=0.0f;
float animInc=0.1f;

//external variables
extern GLuint pbo;
extern GLuint textureID;
extern unsigned int image_width;
extern unsigned int image_height;

// The user must create the following routines:
void runCuda();

void display()
{
  // run CUDA kernel
  runCuda();

  glBindBuffer( GL_PIXEL_UNPACK_BUFFER, pbo);

  // load texture from PBO
  glBindTexture(GL_TEXTURE_2D, textureID);


  // Note: glTexSubImage2D will perform a format conversion if the
  // buffer is a different format from the texture. We created the
  // texture with format GL_RGBA8. In glTexSubImage2D we specified
  // GL_BGRA and GL_UNSIGNED_INT. This is a fast-path combination

  // Note: NULL indicates the data resides in device memory
  glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, image_width, image_height,
                  GL_RGBA, GL_UNSIGNED_BYTE, NULL);


  // Draw a single Quad with texture coordinates for each vertex.

  glBegin(GL_QUADS);
  glTexCoord2f(0.0f,1.0f); glVertex3f(0.0f,0.0f,0.0f);
  glTexCoord2f(0.0f,0.0f); glVertex3f(0.0f,1.0f,0.0f);
  glTexCoord2f(1.0f,0.0f); glVertex3f(1.0f,1.0f,0.0f);
  glTexCoord2f(1.0f,1.0f); glVertex3f(1.0f,0.0f,0.0f);
  glEnd();

  // Don't forget to swap the buffers!
  glutSwapBuffers();

  // if animFlag is true, then indicate the display needs to be redrawn
  if(animFlag) {
    glutPostRedisplay();
    animTime += animInc;
  }
```

```
}

extern float xStart,yStart,zOffset;

//! Keyboard events handler for GLUT
void keyboard(unsigned char key, int x, int y)
{
  switch(key) {
  case('q') :
  case(27) :
    exit(0);
    break;
  case 'a': // toggle animation
  case 'A':
    animFlag = (animFlag)?0:1;
    break;
  case '+': // lower the ocean level
    zOffset += 0.01;
    break;
  case '-': // raise the ocean level
    zOffset -= 0.01;
    break;
  case 'k':
    yStart -= 0.1;
    break;
  case 'j':
    yStart += 0.1;
    break;
  case 'l':
    xStart += 0.1;
    break;
  case 'h':
    xStart -= 0.1;
    break;
  }

  // indicate the display must be redrawn
  glutPostRedisplay();
}

// No mouse event handlers defined
void mouse(int button, int state, int x, int y)
{
}

void motion(int x, int y)
{
}
```

The complete PBO-based Perlin Noise example utilizes the following files. (Note: that simpleGLmain.cpp and simplePBO.cpp are unchanged from the first example.)

- `simpleGLmain.cpp`: Create an OpenGL window and performs basic OpenGl/GLUT setup.
- `simplePBO.cpp`: Perform CUDA-centric setup – in this case for a Pixel Buffer Object (PBO).
- `perlinCallbacksPBO.cpp`: Define keyboard, mouse and other callbacks for the Perlin Noise demonstration using PBOs.
- `perlinKernelPBO.cu`: The CUDA kernel that calculates the Perlin Noise landscape to be displayed.

Use the following nvcc command to compile the testperlin executable. (Please substitute the path to the CUDA SDK libraries for CUDA_LIBRARIES. Similarly, substitute the path to the CUDA SDK include files for CUDA_INCLUDES.)

```
nvcc –O3 -L CUDA_LIBRARIES -I CUDA_INCLUDES simpleGLmain.cpp simplePBO.cpp callbacksPerlin.cpp kernelPerlin.cu -lglut -lGLEW –lcutil –o testperlin
```

Running the executable will display an animated artificial landscape can be interactively moved through the use of the h, j, k, and l keys. Figure 6 is one example.

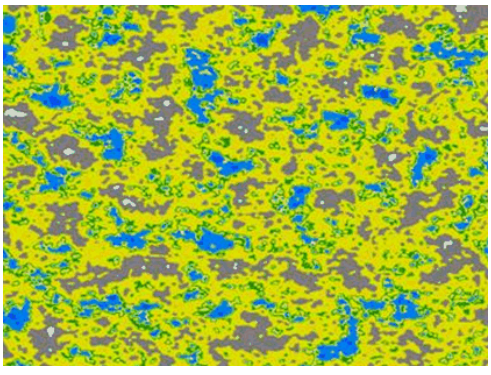

Figure 6

Note that the frame rate reported in the window title reflects the GPU performance as the application recalculates each and every pixel in the landscape for every frame. This was done on purpose to illustrate the excellent performance possible in the worst case when the CUDA kernel must modify every pixel in every frame of an application. For example, an NVIDIA GeForce GTX 285 can deliver many hundreds of frames per second for these examples. Hopefully, these demonstrations can help to illustrate the remarkable headroom available in the current generation of GPUs. Play around with the source code and see if they can deliver a high frame rate per second on your problems for smooth hesitation-free visualization even when the GPU is required to perform extensive physics and other real-time calculations.

For More Information

- CUDA, Supercomputing for the Masses: Part 15
- CUDA, Supercomputing for the Masses: Part 14
- CUDA, Supercomputing for the Masses: Part 13
- CUDA, Supercomputing for the Masses: Part 12
- CUDA, Supercomputing for the Masses: Part 11
- CUDA, Supercomputing for the Masses: Part 10
- CUDA, Supercomputing for the Masses: Part 9
- CUDA, Supercomputing for the Masses: Part 8
- CUDA, Supercomputing for the Masses: Part 7
- CUDA, Supercomputing for the Masses: Part 6
- CUDA, Supercomputing for the Masses: Part 5
- CUDA, Supercomputing for the Masses: Part 4
- CUDA, Supercomputing for the Masses: Part 3
- CUDA, Supercomputing for the Masses: Part 2
- CUDA, Supercomputing for the Masses: Part 1

Rob Farber is a senior scientist at Pacific Northwest National Laboratory. He has worked in massively parallel computing at several national laboratories and as co-founder of several startups. He can be reached at rmfarber@gmail.com.