Scientific Visualization with CUDA and OpenGL

Robert Hochberg and John Riselvato

Shodor

Module Overview

The goal of this module is to teach one pathway for offloading parts of a scientific computation onto a CUDA-enabled GPU, and for using OpenGL to visualize the results of that computation. To keep the focus on CUDA and OpenGL, we select a purely number-theoretic scientific problem with a relatively simple associated computation.

The scientific question:

We'll be looking at the function $f(x) = x^2 + a$ and investigating what happens when, for various values of *a*, we iterate this function. That is, we are interested in what happens when we start with some value *x* and look at the sequence of numbers: $x, f(x), f(f(x)), f(f(f(x))), \dots$. Although we are interested in this as a purely mathematical question, exploration of such quadratic iterations has several scientific applications, including chaos theory.

When *a* = −1, the function $f(x) = x^2 - 1$ has a *point of period 2*, that is, a value *x* such that *f*(*f*(*x*)) = *x*. Observe that our sequence: $x, f(x), f(f(x)), f(f(f(x))), \dots$ starting with *x* = −1 goes: −1, 0, −1, 0, . . ., a repeating sequence with period 2. The same function has a point of period 1, also called a *fixed point*, namely $(1+\sqrt{5})/2$, the so-called *golden ratio*, For this value, $f(x) = x$. For this module, however, we will not be interested in values of *x* or *a* that are not rational, that is, are not ratios of whole numbers. So this fixed point does not really interest us. You can quickly verify that the function $f(x) = x^2 - 1$ has no rational fixed points by solving the equation $x^2 - 1 = x$ and seeing that both its roots (one of which is the golden ratio) are irrational.

Quick Question: Find a rational value of *a* so that the function $f(x) = x^2 + a$ does have a rational point of period 1. [some answers include {a=2, x=2} and {a=−15/4, x=5/2}.]

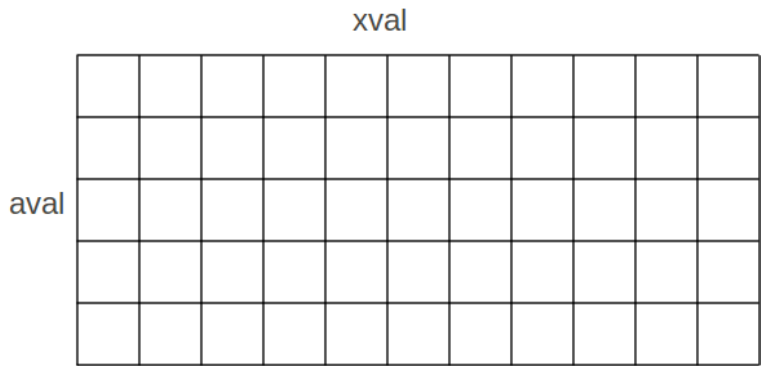
Our question is this: Does there exist a rational number *a* such that the function $f(x) = x^2 + a$ has a rational point of period 7? At the time of writing, the answer is unknown. See [http://people.maths.ox.ac.uk/flynn/arts/art11.pdf] for a very interesting mathematical treatment of this question.

Numeric Approach

Our approach in this module will be to consider the quantity $f^{(n)}(x) - x$ (where $f^{(n)}(x)$ means the *n*th iteration of the function $f(x)$), which is zero whenever *x* is a point of period *n*. Each value of *a* gives rise to a function $f(x) = x^2 + a$, and each value of *x* is a potential starting point for an iteration. So for each *n* we may define the two-variable function $d_n(x, a)$ to be the value of $f^{(n)}(x) - x$, where *x* is the starting point and $f(x) = x^2 + a$. We then search numerically for a fixed point of period n by searching for a suitable rational a and x having $d_n(x, a)$ = 0. Given some range of *a* values: $amin \le a \le amax,$ and range of *x* values: $xmin \le x \le xmax,$ we subdivide each range into values

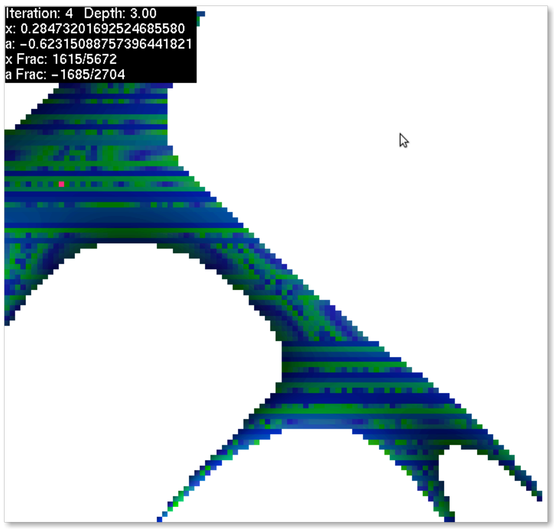
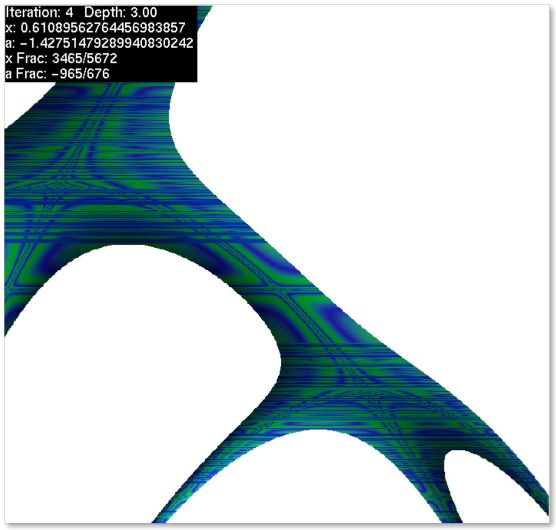
$\{amin = a_0, a_1, \ldots, a_{acount} = amax\}$ and $\{xmin = x_0, x_1, \ldots, x_{xcount} = xmax\}$ and plot the function $d_n(x, a)$ for each pair of values in this interval.

We may think of the domain of our computation as a grid, as shown in the picture below.



We plot in two ways: First, we produce a "height" which corresponds to $|d_n(x, a)|$, so that when this value is 0, we know that *x* is a point of period *n*. We also produce a *color* $c_n(x, a)$ which corresponds (in a loose way) to how close $d_n(x, a)$ is to a rational number. This is a bit silly, since every real number is arbitrarily close to a rational number. But we want to measure how close it is to a rational number with a *small* denominator. We use an algorithm based on continued fractions in the `denomVal()` function. It's very heuristic, and we don't describe it here. The reader is invited to experiment with, and suggest to the authors, better versions of the `denomVal()` function.

For each pair (x_i, a_j) we compute the value $d_n(x_i, a_j)$and plot a quadrilateral in 3-dimensional space with coordinates {$(x_i, a_j, d_n(x_i, a_j))$, $(x_{i+1}, a_j, d_n(x_{i+1}, a_j))$, $(x_{i+1}, a_{j+1}, d_n(x_{i+1}, a_{+1}))$, $(x_i, a_{j+1}, d_n(x_i, a_{j+1}))$} in the color $c_n(x_i, a_i)$. Plotted together these form a surface in 3-space. To make things a bit easier to see, we also make use of a threshold $T>0$ so that we plot a quadrilateral only if all four of its corners' associated d_n values are less than that threshold. A sample screenshot is shown below. The left one, at low resolution, shows the individual quadrilaterals. The one on the right is of higher resolution, and the quadrilaterals can no longer be distinguished very easily.


Iteration: 4   Depth: 3.00
x: 0.28473201692524685580
a: −0.62315088757396441821
x Frac: 1615/5672
a Frac: −1685/2704

Iteration: 4   Depth: 3.00
x: 0.61089562764456983857
a: −1.42751479289940830242
x Frac: 3465/5672
a Frac: −965/676

First Approach --- Single CPU

Our first implementation will use a single CPU, and does not use CUDA. We have two global arrays: `hResults` holding the values of $d_n(x_i, a_j)$, and `hDenomVals`, holding values related to the denominators of the fractions, that we use to assign the colors. (Here, "h" stands for "host," in contrast to the "device." When we use CUDA we will have "`dResults`" for the array residing on the CUDA card, the "device," as well as "`hResults`" for its copy on the host computer.

In the `recompute()` function below, the parameters `xmin` and `xmax` define the range for *x*, and `xstep` is the size of the step from one value of *x* to the next, that is, $xstep = x_{i+1} - x_i$. Same thing for `amin, amax` and `astep`. `numIt` is the number of iterations that the function should perform. This is "*n*" in the discussion above. We store hResults and hDenomVals in one-dimensional arrays, but we think of them as two-dimensional arrays. The variable `stride` gives the width of the 2-d arrays holding the values, so that the `(i, j)` entry is at location `(i*stride+j)` in the arrays. Global variables `acount` and `xcount` give the number of steps in each direction, so that we compute `acount*xcount` many entries altogether.

The `recompute()` function does all of the work of computing the values of $d_n(x_i, a_j)$. It calls the `denomVal()` function to get information about the "rationalness" of the numbers, used for coloring.

```
void recompute(double xmin, double xmax, double xstep,
               double amin, double amax, double astep,
               int numIt, size_t stride){
```

```
  // How many entries do we need? Set these global values.
  xcount = (size_t)ceil( (xmax - xmin)/xstep );
  acount = (size_t)ceil( (amax - amin)/astep );

  double newxval, xval, aval;
  int xp = 0, ap = 0;
  for(xval = gxmin; xval <= gxmax; xval += gxstep){
    ap = 0;
    for(aval = gamin; aval <= gamax; aval += gastep){
      double originalXval = xval;
      newxval = xval;

      // Perform the iteration
      for(int i = 0; i < iterations; i++) //Iterate
        newxval = newxval * newxval + aval;

      // Fill the d_n(x, a) and color arrays.
      hResults[ap*stride+xp] = newxval - originalXval;
      hDenomVals[ap*stride+xp] =
         denomVal(fabs(newxval-originalXval), fabs(aval), gdepth);
      ap++;
    }
    xp++;
  }
}
```

Once these two arrays have been filled with numbers, we render them to the screen using OpenGL, as described in the next section.

Drawing with OpenGL

Introduction to OpenGL

OpenGL (Open Graphics Library) is a hardware independent, cross-platform, cross-language API for developing graphical interfaces. It takes simple points, lines and polygons and gives the programmer the ability to create outstanding projects.

Our model uses GLUT for the simplest way of handling window display, and for Mouse and Keyboard callbacks. To use GLUT for managing tasks, you should include it in your header file:

```
#include <GL/glut.h>
```

Displaying a Window

To display a window, five necessary initial functions are required.

- `glutInit(&argc, argv)` should be called before any other GLUT function. This initializes the GLUT library.

- `glutInitDisplayMode(mode)` which is used to determine color display and buffer mode to be used. More information on modes here: http://www.opengl.org/documentation/specs/glut/spec3/node12.html
- `glutInitWindowSize (int x, int y)` specifies the size for the application.
- `glutInitWindowPosition (int x, int y)` specifies the location on screen for the upper-left corner of the window
- `glutCreateWindow(char* string)` creates the window, but requires `glutMainLoop()` for actual display. The string will create the window's label.

The above required functions should be called in an `init()` function. Our module's init function is `initGlutDisplay(argc, argv)`. In the same init function, `glutMainLoop()` should be called as the last routine; this ties everything together and runs the main loop. Example 1.0, shown below, is a complete, self-contained OpenGL program that draws a red square inside a square window 300 pixels on a side.

Example 1.0: Hello World

```
#include <GL/gl.h>
#include <GL/glut.h>

void displayForGlut(void){
  //clears the pixels
  glClear(GL_COLOR_BUFFER_BIT);
  glColor3f(1.0, 0.0, 0.0);
  glBegin(GL_QUADS);
  glVertex3f(0.10, 0.10, 0.0);
  glVertex3f(0.9, 0.10, 0.0);
  glVertex3f(0.9, 0.9, 0.0);
  glVertex3f(0.10, 0.9, 0.0);
  glEnd();
  glFlush();
}

int initGlutDisplay(int argc, char* argv[]){
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_RGB);
  glutInitWindowSize(300, 300);
  glutInitWindowPosition(100, 100);
  glutCreateWindow("Example 1.0: Hello World!");
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);
  glutDisplayFunc(displayForGlut);
  glutMainLoop();
  return 0;
}

int main(int argc, char* argv[]){
  initGlutDisplay(argc, argv);
```

```
        }
```

Callbacks

GLUT uses a great system of callbacks for both drawing its display and reacting to user input. In Example 1.0, the display is drawn in the `displayForGlut()` function. This function is called whenever OpenGL needs to redraw its window. The way we tell OpenGL to use this function whenever it wants to redraw the window is by *registering* `displayForGlut()` as a *callback* function. The line `glutDisplayFunc(displayForGlut)` within the `initGlutDisplay()` function accomplishes this. It tells OpenGL that whenever this window needs to redraw itself, execute the `displayForGlut()` function. In other parts of the code we may use the OpenGL command `glutPostRedisplay()` to ask OpenGL to redraw the window.

In general, we register our callback functions when we first initialize a window, and then these callbacks run whenever the appropriate event requires them. In the provided code file `squareExplore.c` we register all callbacks in our `initGlutDisplay() function`. In example 1.0, we used the `glutDisplayFunc()` callback to get to our display method. Keyboard, mouse, and mouse motion all have their own callback functions:

- `glutKeyboardFunc()` registers the callback for ordinary keyboard input.
- `glutSpecialFunc()` registers the callback for special keyboard input such as function keys and arrow keys.
- `glutMouseFunc()` registers the function that responds to mouse clicks.
- `glutMotionFunc()` registers the function that responds to mouse "drags."
- `glutPassiveMotionFunc()` registers the function that responds to mouse motions in the window when a button is not depressed.

We use three of these callbacks in `squareExplore.c`, but all five of them among the original code, the exercises and the solutions to the exercises. For more information about callbacks and OpenGL, see the online reference: http://www.opengl.org/documentation/specs/glut/spec3/node45.html.

Our Callbacks

Our model uses several callback functions for an enriched user experience:

- `displayForGlut()` callback function is registered to OpenGL by the `glutDisplayFunc()` registration function. We use `displayForGlut()` for most of our display, including the graphing iterations and handling text.
- `kbdForGlut()` callback function is registered to OpenGL by the `glutKeyboardFunc() registration function`. We use this callback for non-special keys such as a-z and A-Z. Certain keys in our model provide a visual change in the display. As explained before, `glutPostRedisplay()` is called at the end of this method for redisplay.
- `arrowsForGlut()` callback function is registered to OpenGL by the `glutSpecialFunc()` registration function. Glut organizes special keys such as the arrow keys and ctrl key in this callback. We use it for the arrow keys, which allow the user to pan through the display, up, down, left and right.

- `mouseMotionForGlut()` is used to capture mouse motion events for when the user is not holding down a mouse button. It is registered via the `glutPassiveMotionFunc()` registration function. This is used to update the coordinates and fractions displayed on the screen.

glutMainLoop()
As mentioned before, `glutMainLoop(`*`void`*`)` must be the very last function to be called when initializing a window. This begins event processing and the display callback is triggered. Also other registered callbacks are stored, waiting for events to trigger them. The method that contains the `glutMainLoop()` will never be called again unless the user calls `glutMainLoopEvent()` or `glutLeaveMainLoop()`, which we do not consider in this module. For more information on `glutMainLoopEvent()`, `glutLeaveMainLoop()`, or about the `glutMainLoop()`, see: http://openglut.sourceforge.net/group__mainloop.html

How Our Drawing works
Our drawing consists of quadrilaterals in 3-space, but it is easiest to think about if we start with the 2-d version. As described in the section **Numeric Approach**, we compute $d_n(x_i, y_j)$ for many points forming a square mesh, and then color each square of that mesh according to the `hResults` array entry corresponding to the bottom-left side of that square. This color corresponds roughly to how “rational” the values x_i and y_j are. Finally, we give a height to each corner of the square corresponding to its value in the `hResults` array.

In our init function `initGlutDisplay()` we set our window drawing to be double-buffered, by calling `glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB)`. This means that our drawing code will take place on a buffer off-screen, and will not replace the contents of the current screen (the current buffer) until we call `glSwapBuffers()`. This automatic double-buffering causes motion and animation to appear much smoother. More on `glutSwapBuffers()` here http://www.opengl.org/resources/libraries/glut/spec3/node21.html

Our drawing begins by clearing the buffer, giving us a blank screen on which to draw. The two `for` loops run through the 2-d array of points, and the `if` statement guarantees that a quadrilateral is drawn only if all four of its corners are sufficiently small, that is, are less than the global value `threshold`. Then, if the quad may be drawn, we tell OpenGL that we will be drawing a quadrilateral (`glBegin(GL_QUADS)`), specify the vertices by issuing `glVertex3f(x, y, z)` commands, one for each vertex, in order around the quadrilateral, and then call `glEnd()` to finish that quadrilateral. When the loops finish and all quadrilaterals have been drawn, we call `glutSwapBuffers()` to put our drawing, now complete, onto the screen.

Printing Text
OpenGL has no native way of displaying text. Fortunately, there are ways around this. In this module, we created a function called `drawString(`*`void* font, char* s, float x, float y, floatz`*`);` Glut does come with bitmap fonts; we happen to use Helvetica. For more fonts: http://pyopengl.sourceforge.net/documentation/manual/glutBitmapCharacter.3GLUT.html

The drawString function requires a font, string, x location, y location, and z location. x, y and z location are all set via the `glRasterPos3f(float x, float y, float z)` function. This is followed by a `for` loop which pushes the value of the wanted string into a `glutBitmapCharacter` function. This seems like a lot of effort to draw text, but nothing simpler presented itself. The general idea came from http://stackoverflow.com/a/10235511/525576.
View *Example 2.0 - displaying text*

Example 2.0 - displaying text
```
char gMouseXLocation[25];
void displayText(void){
      /* xMotionLoc would be the global x value retrieved from
       * glutPassiveMotionFunc callback.
       */
      sprintf(gMouseXlocation, “X Location: %2.20f”, xMotionLoc);
      drawString(GLUT_BITMAP_HELVETICA_18, gMouseXLocation, 0, 100, 0);
      glutPostRedisplay();
      glutSwapbuffers();
}
```

Concluding Message
At this point with a basic understanding of OpenGL, we hope you have an appreciation of how easy OpenGL is to work with. OpenGL has over 250 function calls with which to draw complex scenes. Three great resources to expand your newly learned skills are the OpenGL API Documentation, NEHE tutorials and The Big Red Book (OpenGL Programming Guide), all free online.

OpenGL API DOCS: http://www.opengl.org/documentation/
NeHe Tutorials: http://nehe.gamedev.net/
Big Red Book: http://www.glprogramming.com/red/

CUDA Version

The pair of nested loops in the `recompute()` function is an example of an “embarrassingly parallelizable” bit of code. The computation done within each iteration is completely independent of the computation done in other iterations. This means that we may hand the lines of code below to many different processors, each of which will perform the computation for a single pair of values (`xval, aval`) and save the result to memory.

```
double originalXval = xval;
newxval = xval;
for(int i = 0; i < iterations; i++) //Iterate
```

```
    newxval = newxval * newxval + aval;
hResults[ap*stride+xp] = newxval - originalXval;
hDenomVals[ap*stride+xp] = denomVal(fabs(newxval-originalXval),
                                    fabs(aval), gdepth);
```

The architecture of a GPGPU is uniquely well-suited for such embarrassingly-parallelizable bits of code. For example, the graphics card in my MacBook Pro (GeForce GT 330M) has six multiprocessors, each of which has eight cores, so that 48 threads may be running at any one time. Theoretically, then, we could expect a factor of 48 speedup in the running of our `recompute()` function. Of course, this is tempered by the need to transfer the computation and/or data onto the card, and then read the results off of the card. In our case, the speedup makes it worth it.

Offloading to CUDA

Each time we pan or zoom our display, or change the resolution of our drawing, we must recompute the values in the `hResults` and `hDenomVals` arrays. The bulk of that work is done by the two `for` loops, shown in the box below, within the `recompute()` function:

```
for(xval = gxmin; xval <= gxmax; xval += gxstep){
  ap = 0;
  for(aval = gamin; aval <= gamax; aval += gastep){
    double originalXval = xval;
    newxval = xval;
    for(int i = 0; i < iterations; i++) //Iterate
      newxval = newxval * newxval + aval;

    hResults[ap*stride+xp] = newxval - originalXval;
    hDenomVals[ap*stride+xp] = denomVal(fabs(newxval-originalXval),
                                        fabs(aval), gdepth);
    ap++;
  }
  xp++;
}
```

In the CUDA version, these loops will be replaced by the launch of a kernel that runs on the GPU, preceded by a bit of setup, and followed by copying the results off of the GPU and onto the host computer, as shown in the box below:

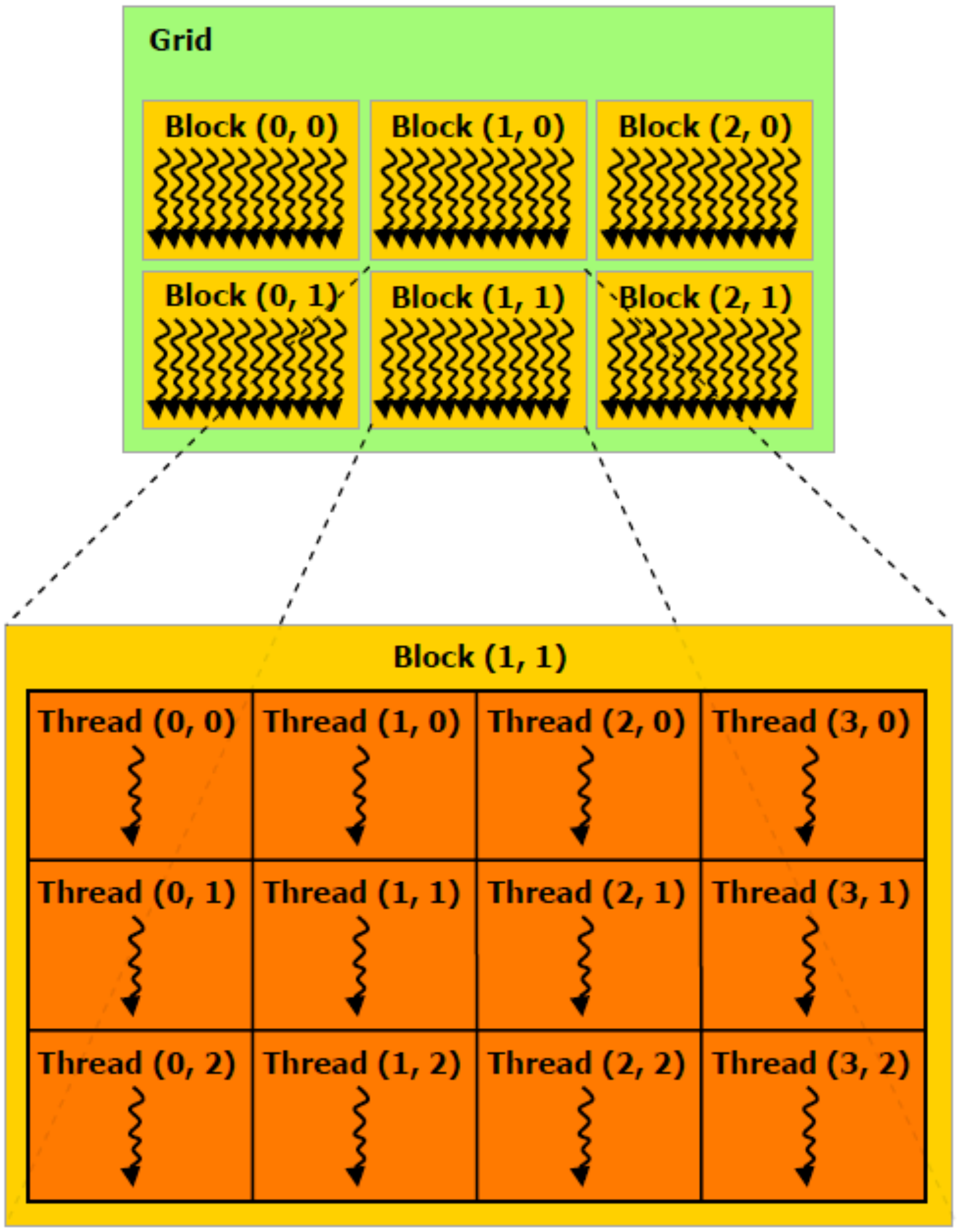
```
dim3 dimBlock(BLOCKSIZE, BLOCKSIZE);
dim3 dimGrid((int)ceil((xmax - xmin)/xstep/BLOCKSIZE),
             (int)ceil((amax - amin)/astep/BLOCKSIZE));

iterate <<< dimGrid, dimBlock >>> (xmin, xmax, xstep, amin, amax, astep,
                                   numIt, stride, (double*)dResults,
                                   gdepth, (double*)dDenomVals);
cudaThreadSynchronize();

// Copy the results off of the device onto the host
cudaMemcpy(hResults, dResults, stride * acount * sizeof(double),
           cudaMemcpyDeviceToHost);
cudaMemcpy(hDenomVals, dDenomVals, stride * acount * sizeof(double),
           cudaMemcpyDeviceToHost);
```

The first two statements define the size of the thread blocks (16x16 in this case, since `BLOCKSIZE=16` in our code) and the size of the grid, which is set to be large enough that the grid of blocks covers all of the range we wish to cover. (Blocks and Grids are discussed in the next section.) The command `iterate <<<...>>>(...)` is the kernel call. It passes eleven parameters to the kernel in the way an ordinary C function would, and it specifies the block and grid sizes within the triple-angle brackets. This tells CUDA exactly how many threads to create, and each thread will execute exactly the same code --- the kernel code given in the `iterate()` function. As discussed in the **Our Kernel** section below, threads can "locate" themselves by making use of several special variables, and determine the correct values of `xval` and `aval` to use in their computation.

Blocks and Grids

One great idea of GPU computing is that, in some sense, we may program *as if* the graphics card had an unlimited number of computing nodes. For example, suppose we wish to compute $d_n(x_i, y_j)$ for 1024 values of *i*, say $i = 0, 1, 2, \ldots, 1023$, and the same 1024 values for *j*. That is, we wish to perform 1,048,576 computations. Then we may imagine that there are 1,048,576 compute nodes laid out in a 1024x1024 array, and simply ask each node (*i*, *j*) to perform its computation and store its result $d_n(x_i, y_j)$ at the appropriate place in memory.

In reality, though, there are only six multiprocessors on my graphics card, each of which can run up to eight threads at a time. So CUDA requires that we subdivide our domain into *blocks* of threads, each block containing no more than 512 threads. In the code supplied with this module, `squareExplore.c` and `squareExplore.h`, we use a default block of dimensions 16x16, containing 256 threads.



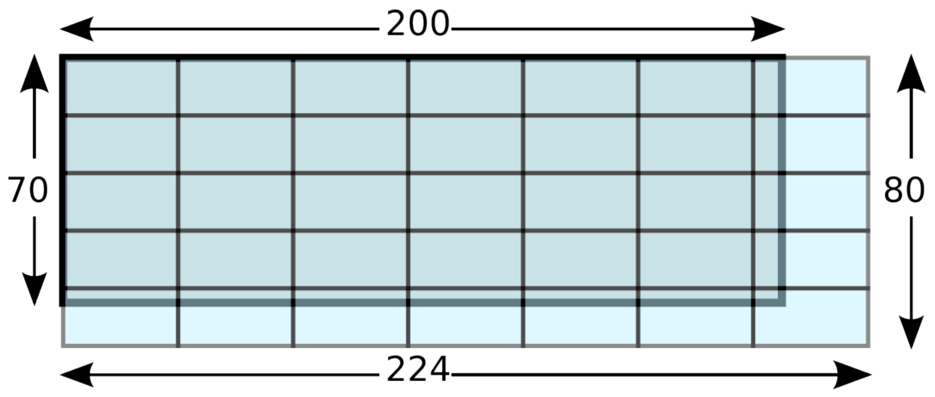
The picture above (taken from the CUDA C Programming Guide, published by NVIDIA) shows a grid (in green) of blocks. Each block contains some number of threads, and the blocks are arranged to form a *grid* covering the entire domain. In our example, we had a grid of threads of size 1024x1024. To cover this with blocks of size 16x16, we would need a 64x64 grid of blocks.

Once we've determined the size of our blocks and the size of our grid, we can launch a *kernel* on the CUDA card. (A kernel is a self-contained unit of work that we offload to the GPU. This kernel runs on the GPU, may read and write to memory on the GPU, and is often surrounded by writes from host memory to the device memory and/or reads from device memory to host memory.) To specify the sizes of the blocks and grid we use a CUDA type --- `dim3` --- which may contain one, two or three unsigned integers. More precisely, it always contains three unsigned integers, but the programmer may specify fewer, and the unspecified ones get initialized to 1. Here are the lines specifying block and grid sizes from squareExplore.c:

```
dim3 dimBlock(BLOCKSIZE, BLOCKSIZE);
dim3 dimGrid((int)ceil((gxmax - gxmin)/gxstep/BLOCKSIZE),
                         (int)ceil((gamax - gamin)/gastep/BLOCKSIZE));
```

The first line declares the variable dimBlock to be of type dim3, and sets its first two coordinates to BLOCKSIZE. So dimBlock will be the triple: (BLOCKSIZE, BLOCKSIZE, 1). Similarly, dimGrid will be a triple (gridsize, gridsize, 1) where gridsize = (int)ceil((gxmax - gxmin)/gxstep/BLOCKSIZE) is computed to be just large enough that the full ranges (gxmin, gxmax) and (gamin, gamax) get covered.

The next line of the code launches our CUDA kernel. A CUDA kernel is an ordinary function declared to be a kernel function by virtue of the "`__global__`" keyword in its declaration. The NVIDIA `nvcc` compiler recognizes this keyword and then makes several additional variables available to the function. The figure below provides a concrete example to help explain these variables. In this example, we have a 70x200 grid of threads that we wish to run, and we cover it with blocks of dimensions 16x32. Since 70 and 200 are not whole multiples of 16 and 32, the grid will need to overhang our area of interest by some amount.



gridDim	The dimensions of the grid. Use gridDim.x, gridDim.y and gridDim.z to access the dimensions. In the Figure above, gridDim.x = 7, gridDim.y = 5 and gridDim.z = 1.
blockIdx	The location of a block within the grid. All 512 threads in the middle block of the grid would find blockIdx.x = 3, blockIdx.y = 2, blockIdx.z = 0.
blockDim	The dimensions of the blocks. All threads in the example would find blockDim.x = 32, blockDim.y = 16 and blockDim.z = 1.
threadIdx	This structure gives the location of a thread within its own block. Thus the top-left thread of each block would find threadIdx.x = threadIdx.y = threadIdx.z = 0, while the top-right threads would find threadIdx.x = 31, threadIdx.y = 0 and threadIdx.z = 0.

Our kernel
Now let's look at our actual kernel code, shown in the box below:

```
__global__ void iterate(double xmin, double xmax, double xstep,
                  double amin, double amax, double astep,
                  int numIt, size_t stride, double* dResults,
                  int depth, double* dDenomVals){

  int xidx = blockIdx.x * blockDim.x + threadIdx.x;
  int aidx = blockIdx.y * blockDim.y + threadIdx.y;

  double xval = xmin + xidx * xstep;
  double aval = amin + aidx * astep;

  // Make sure we're within range
  if(xval > xmax || aval > amax)
    return;

  // Iterate
  double originalXval = xval;            // save for later
  for(int i = 0; i < numIt; i++)
    xval = xval * xval + aval;

  // compute our location in the array
  int loc = aidx * stride + xidx;

  dResults[loc] = xval - originalXval;
  dDenomVals[loc] = denomVal(fabs(xval - originalXval), fabs(aval), depth);
}
```

The `__global__` keyword is used to tell the NVIDIA compiler nvcc that this function is a kernel function, intended to be an entry point for computation on the GPU. (In contrast, the `__device__` keyword tells the compiler that a function will be run on the GPU, to be called by other GPU kernels or functions, but is not an entry point function.)

As mentioned earlier, every thread will run exactly the same code, so the first thing a thread typically does is to locate itself within the space of threads that will be launched by the kernel. The variables `xidx` and `aidx` accomplish that. Recall that our computation may be viewed as computing at points comprising a grid. In this sense, `xidx` tells how far "across" we are, and `aidx` tells how far "up" we are. These values take the place of `xp` and `ap` in our non-CUDA code for `recompute()`.

Next our code uses the values of `xidx` and `aidx` to determine what values of `xval` and `aval` it should use for its computation, and performs the iteration. The `loc` variable tells where in the arrays the value of the results should be stored. Each row of our matrix is of width "`stride`," so if we want to go to the `aidx`'th row, we multiply stride by `aidx` to find the starting point of that row. Then add "`xidx`" to find the location of the entry in the `xidx`'th column.

Finally we store our computations in the arrays. First we store the difference between the original and iterated value. Recall that this difference will be 0 if the starting value `xval` is of period `numIt`. These arrays reside in the video card's "global" memory, which we discuss below.

Device and Host Memory
Multiprocessors have a large number of 32-bit registers: 8k for devices of compute capabilities 1.0 and 1.1, 16k for devices of compute capability 1.2 and 1.3, and 32k for devices of compute capability 2.0 or above. (See Appendix F of **[2]** for a description of compute capabilities.) Here we describe the various kinds of memory available on a GPU.

Registers	Registers are the fastest memory, accessible without any latency on each clock cycle, just as on a regular CPU. A thread's registers cannot be shared with other threads.
Shared Memory	Shared memory is comparable to L1 cache memory on a regular CPU. It resides close to the multiprocessor, and has very short access times. Shared memory is shared among all the threads of a given block. Section 3.2.2 of the Cuda C Best Practices Guide **[1]** has more on shared memory optimization considerations. Each multiprocessor has on the order of 32k of shared memory.
Global Memory	Global memory resides on the device, but off-chip from the multi-processors, so that access times to global memory can be 100 times greater than to shared memory. Devices typically have between 1 and 6 gigabytes of global memory. All threads in the kernel have access to all data in global memory.
Local Memory	Thread-specific memory stored where global memory is stored. Variables are stored in a thread's local memory if the compiler decides that there are not enough registers to hold the thread's data. This memory is slow, even though it's called "local."
Constant Memory	64k of Constant memory resides off-chip from the multiprocessors, and is read-only. The host code writes to the device's constant memory before launching the kernel, and the kernel may then read this memory. Constant memory access is cached — each multiprocessor can cache up to 8k of constant memory, so that subsequent reads from constant memory can be very fast. All threads have access to the constant memory.
Texture Memory	Specialized memory for surface texture mapping, not discussed in this module.

Before launching a CUDA kernel, the host program (the program on the computer, not the device) may allocate memory on the video card using cudaMalloc(). This function sets a pointer to the allocated memory, which can be used in three important ways: 1. For use in copying data from the host computer to the device at that location; 2. To send to the kernel program running on the device, so that the kernel can read from and/or write to that memory, and 3. For use in copying data back off of the device. The memory thus allocated resides in the device's global memory.

The last two commands in our `recompute()` code given earlier are shown below:

```
// Copy the results off of the device onto the host
cudaMemcpy(hResults, dResults, stride * acount * sizeof(double),
          cudaMemcpyDeviceToHost);
cudaMemcpy(hDenomVals, dDenomVals, stride * acount * sizeof(double),
          cudaMemcpyDeviceToHost);
```

Here is an explanation of the parameters in the first copy:

- `hResults` is a pointer to memory on the host. This is the destination
- `dResults` is a pointer to memory on the device. This is the source
- `stride * acount * sizeof(double)` is the amount of memory, in bytes, that we are copying
- `cudamemcpyDeviceToHost` is an enumerated constant indicating the direction of the copy.

When we launched the kernel earlier we had passed the pointer `dResults` to the kernel, so that it would know where to place the results of its computations.

Finally, here is how we allocated the memory on the device in the first place, in `main()`. (Note that we allocate this memory only once, but then use it over many kernel launches.)

```
 err =  cudaMallocPitch((void**)&dDenomVals, &stride, 1088*sizeof(double),
1088*sizeof(double));
```

There is a simpler command `cudaMalloc`, which has only two parameters: pointer and size, and we could have used it here. But since we intend to use our memory as a two-dimensional array, we use `cudaMallocPitch()` instead. When a CUDA kernel is running, if the threads in a block all make a simultaneous access to global memory then the CUDA runtime can group those accesses into a single read or write operation to global memory, greatly increasing performance. Global memory is read in 32-, 64- or 128-byte transactions, where each block must start at a memory location that is a multiple of its size. (So a 128-byte transaction can start at 0, 128. 256, etc...). What `cudaMallocPitch()` does is set a value (`stride` in our example) that is a multiple of the optimal block size (32, 64 or 128) so that each row of our array begins at one of these multiples. See Section 5.3.2.1 of **[2]** for more on stride and memory alignment.

Running the Code

Two Makefiles are included with this module: One in the main directory and one in the assessment directory. The latter is shown below.

```
se : squareExplore.c squareExplore.h
    cp squareExplore.c squareExplore.cu
    /usr/local/cuda/bin/nvcc -arch=sm_20 -o se -lglut -lGL -lGLU \\
       -lXext -lXmu -lX11 -lm  squareExplore.cu

seNoCuda : squareExploreNoCuda.c squareExploreNoCuda.h
    g++ -o seNoCuda -lglut -lGL -lGLU -lXext -lXmu -lX11 \\
       -lm  squareExploreNoCuda.c
```

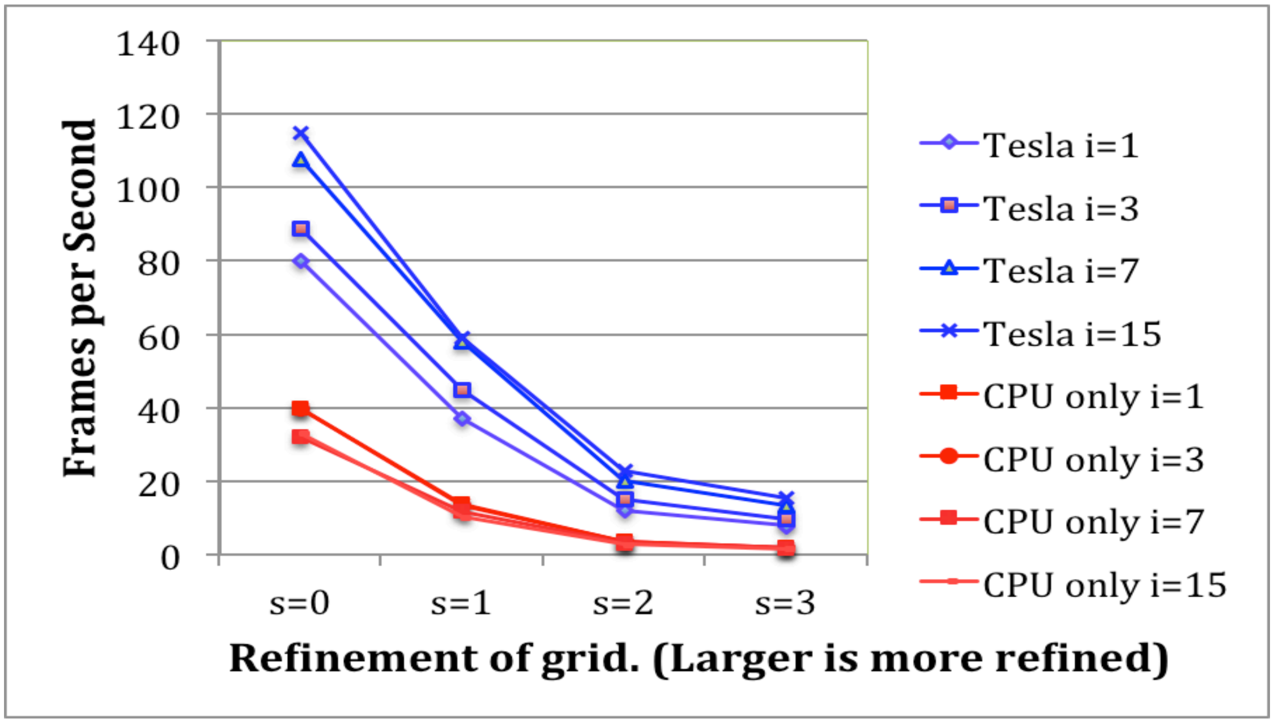
To build the version that uses CUDA, type "`make se`" at a command line. To build the one that does not use CUDA, type "`make seNoCuda`". These create executables "`se`" and "`seNoCuda`" respectively, which can be run by typing "`./se`" and "`./seNoCuda`" respectively. Note that several libraries related to OpenGL and CUDA need to be installed, and these will be specific to your system.

Also note the "`-arch=sm_20`" flag. This tells the compiler that the code should be targeted for a CUDA device of compute capability 2.0 or higher. You may need to change the "20" to 11, 12 or 13 if you are running on a device of compute capabiltiy 1.1, 1.2 or 1.3, respectively.

IMPORTANT NOTE: Many text editors facilitate writing C code by automatically indenting code and highlighting the code by context, and does so by recognizing the ".c" extension of the file. Most of these editors do not recognize the "`.cu`" extension. I've therefore been writing the code as "`.c`" files, and then having the Makefile copy them to "`.cu`" files before compilation. This is because the nvcc compiler requires the "`.cu`" extension. There are other ways to deal with this discrepancy... Pick your own. But be aware that this is the workflow that the included Makefiles expect.

Timing for CUDA vs. non-CUDA versions

The chart below gives a comparison of the number of frames per second that can be rendered with and without making use of CUDA. For our tests we used an NVIDIA Telsa C2075 with 448 CUDA cores, and an AMD CPU running at 3.6 GHz. The "i=…" numbers refer to the number of iterations of the "square and add" operation that were performed when computing each grid element.



Frames per Second	s=0	s=1	s=2	s=3
Tesla i=1	80	37	12	7.7
Tesla i=3	89	45	15.2	9.8
Tesla i=7	108	58	20.2	13.6
Tesla i=15	115	59	23	15.5
CPU only i=1	40	13.7	3.5	2
CPU only i=3	40	13.6	3.6	2.1
CPU only i=7	32	11.7	3	1.8
CPU only i=15	33	10.1	2.6	1.5

Interesting Observation: The frame-rates shown in the table above are higher in all cases when we make use of CUDA. Notice that the frame rate *increases* when we make use of the GPU, as we increase the number of iterations. This is because with more iterations, fewer quads fall below the threshold for rendering to the screen, so the image may be drawn more quickly by OpenGL. But in the CPU-only case, more iterations leads to a *lower* frame rate, even though exactly the same number of quads will be drawn. This is because more iterations means more computation for each quad --- exactly where massive parallelism helps.

Explorations

The following exercises are designed to give the student an opportunity to practice with the ideas presented in this module. Solutions to all the problems are included with the module, in an appropriately-named pair of files.

1. Scenes can be viewed in different ways. Currently our Scene is with a Model View. Render the scene as a wireframe view. Refer to the glBegin() parameters. Explain why it visually looks like it does, instead of one big outlined wire.

2. Create a keyboard shortcut that switches between original rendering and wireframe rendering. The rendering state should stay constant even if other callbacks are triggered.

3. Currently the application only Rotates in 'x' (g/G key) and 'a' (f/F key) direction. Implement a 3rd key 'h' and 'H' to rotate in the 'z' direction.'x' and 'a' both rotate in the center of the screen, please follow the same rule with the 'z' direction. Refer to `glTranslatef()` and `glRotatef()` for help.

4. Rotation via keys f/F, g/G and h/H are fine, but using the mouse for rotation would be even better. With the same idea as the above keys, write a new function callback that registers to `glutMouseFunc()`, `glutPassiveMotionFunc()` and/or `glutMotionFunc()`. Creating rotation for 'x' and 'a' is enough. For assistance on these callbacks, view the Registering Callbacks section from the OpenGL introduction PDF.

5. Write a version that will graph general functions instead of our iterated quadratic. For example, how would you change the kernel so that it graphs 10*sin(xy) instead?

Bibliography

[1] The NVIDIA Corporation. The CUDA C Best Practices Guide v4.0. NVIDIA Corporation, 2011.
[2] The NVIDIA Corporation. The CUDA C Programming Guide v4.0. NVIDIA Corporation, 2011.