# Mandelbrot

## Introduction

We were given a sample of code which calculated a Mandelbrot and output it to a .ppm file. The code given was completely sequential, and we were given the task of modifying it to work with NVidia’s CUDA to see if it could be used in order to get a performance boost.

For the project, the coding style was designed to be closer to C than C++. The reason for this was so that the paper could be more readable to the layman. C is a much simpler language compared to C++, and since the language has a lot less features, then it is much easier to read as the person can always be sure semantically what each line is doing in the code, before they fully understand what is happening in it.

|  |
| --- |
| err = cudaMalloc((void \*\*)&gpu\_img\_data, img\_data\_size);  assert(err == cudaSuccess); |

*Figure 1: Example of the primitive error handling used*

The return call for each CUDA function was checked using a *cudaEvent\_t* variable. It was then asserted to make sure it was equal to *cudaSuccess*. While this error handling was very primitive, it did serve the function of letting the programmer know when something went wrong in the CUDA code, so it could be fixed. This simple error checking had the benefit of not polluting the code to much, and it didn’t introduce complicated error handling that would make the program more difficult to understand.

## Hardware

The CPU used was an Intel i7 chip with 8GB memory.

The GPU was a NVIDIA Quadro K4000 with approximately 11 GB of memory.

## Software

* Visual Studio 2015
* CUDA Toolkit 8.0
* Git
* Github

## Create a window

Before beginning the project, the first thing we did was write some code which would display a window and draw the Mandelbrot to it. Because the Mandelbrot is generating a *.ppm* file, which is really just writing out the red, green, and blue colour data to disk, it was very easy to get the colour data in order to draw the Mandelbrot to the window. To draw it to the window, a simple *for* loop was created that would loop through the data, and call *DrawPixel* to place colours the pixel at that position to the same as the *.ppm* file.

The reason for creating a window was to increase the iteration time. We feel it’s important to be able to make a change in the code, run it, and test it quickly. Having the code write to disk meant we would have to; build the code, run it, open the image in an image editor, and then check it. While adding the window was not a part of the assessment, it meant that we could simply; build the code, run it, and check the image on the screen. We feel like increasing the iteration time this way meant that, over the course of the project, we have saved time by sacrificing some time at the beginning.

There is also a console version of the code, without the window. The console and windowed version of the code can be toggled by changing the line *#define DRAW\_WINDOW 0* at the top of the code to *#define DRAW\_WINDOW 1*. After that, you have to right click on the project in the solution explorer and go to *properties*. Then go to *Configuration Properties* -> *Linker* -> *System,* and changed *Subsystem* from “Windows (/SUBSYSTEM:WINDOWS)” to “Console (/SUBSYSTEM:CONSOLE)”.

## Thread CPU code

Once the window was done, which could output the generate *.ppm* file, we decided to thread the CPU version of the code. There were two reasons for this.

The first was so that there was a serial starting point for doing the CUDA version. Serializing the code on the CPU, which is much easier to debug, meant that we could sort out any nasty surprises and figure out how the Mandelbrot algorithm worked at a deeper level. It also let me be sure that the code could be done parallel more easily.

The second was so that the CUDA version of the code had a strong CPU version to compare against. Comparing optimized CUDA code againt un-optimized CPU code would not be very useful, as it does not give a true representation of how much faster the GPU version would be compared to the CPU version.

Due to the nature of the assignment, and since it was tailored towards learning GPGPU programming and the CUDA API, we did not add any SIMD elements to the CPU version of the code. While the x64 architecture does support wide instructions, and the computer used supported SSE4 instructions, which can do 4-wide vector operations, we did not feel it would be a good use of my time to implement this.

## CUDA Code

There are two kernels called within the CUDA version of the code. The first is very simple, and only sets the row pointers for the *gpu\_img\_data* variable.

The second function generates the Mandelbrot data on the GPU. The original code was within two *for* loops, but instead we launched a 2 dimensional kernel and get the *i* and *j* variables by calculating their index within the block.

The function to map the colours was put into the *calculate\_mandelbrot* function. Initially, we were hesitant to do this, because it adds some branches into the code which may hurt the performance of the CUDA code. However, it avoided the overhead of a kernel launch, which saved about0.02ms. Had there been more time to spend on the project, we may have tried to remove the branch completely, which could have possibly sped up the performance of the CUDA code. Because of the nature of GPGPU programming, each branch may end up getting taken, then some of the values which did not take that code path are discarded. This can often damage a lot of the performance benefits of programming on the GPU, and should be avoided as much as possible.

In order calculate the size of the occupancy for the block being created, the function *cudaOccupancyMaxPotentialBlockSize* was used. This function, which was added in CUDA 6.5, greatly simplifies the block size. Prior to the introduction of this function, many developers would use the *Occupancy Calculator Spreadsheet* included with the CUDA Toolkit to find good block sizes on supported GPUs. The function has the obvious benefit of being done on the fly, so it is more robust to code changes within the function being launched.

|  |
| --- |
| minGridSize = 0, blockSize = 0;  dataLength = width \* height;  err = cudaOccupancyMaxPotentialBlockSize(&minGridSize, &blockSize,  calculate\_mandelbrot, 0, 0);  assert(err == cudaSuccess);  blocks = (int)(pow(2, ceil(log(sqrt(blockSize)) / log(2))));  grid = (int)((sqrt(dataLength) + blocks - 1) / blocks); |

*Figure 2: Sample code for calculating occupancy size*

As you can see in figure 2, once we have the *blockSize* and *minGridSize* from the CUDA function, it is then possible to calculate the blocks and grid that are going to be used. The blocks are calculated by taking the block size given back by the function and rounding it up to the nearest power of two. This was done for performance reasons, as it is generally more efficient to launch kernels with a block size that is a power of two. The grid size is rounded up based on the input data size.

|  |
| --- |
| calculate\_mandelbrot<<<dim3(grid, grid),  dim3(blocks, blocks)>>>(width, height,  scale, gpu\_row\_ptrs); |

*Figure 3: How the calculate\_mandelbrot kernel is launched.*

|  |
| --- |
| int i = threadIdx.x + blockIdx.x \* blockDim.x;  int j = threadIdx.y + blockIdx.y \* blockDim.y; |

*Figure 4: How indexes are calculated*

Figure 3 shows how the Mandelbrot calculation function is launched. The part in the triple chevrons is the grid size and block size. The *dim3* part means launch a 3 dimensional kernel, but since the *z* is left blank, and it gets defaulted to 1, only a 2D kernel is getting launched.

Figure 4 shows how the indexes to access the pixel data are calculated. Adding *blockIdx* and *blockDim* will calculate the index within a thread block, which will be unique for each index in the block. However, this can cause different threads to find the same index, so the number must be added to by the *threadIdx* variable. This gives a unique value to each index.

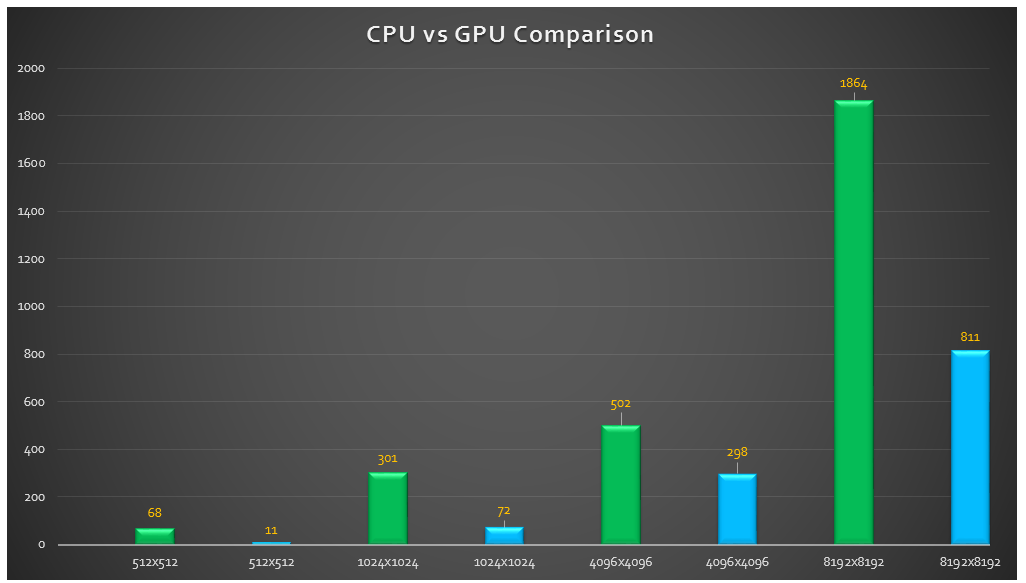
Because we launched a 2D kernel, you can simply changed the *.x* to *.y* and to access the values for the *y*. This is useful because the *Mandelbrot* calculation works on pixels in a screen, on an *x* by *y* plane. The CPU version of the code uses the *i* and *j* variables within a double *for* loops to access the *x* and *y* pixels. Due to the nature of the GPU code, it instead calculates the *i* and *j*, avoiding the need for the *for* loops completely.

## Calculate Mandelbrot

The CUDA and CPU-only executions of the code are shown below. All times are based averages from ten runs of the code.

|  |  |  |
| --- | --- | --- |
| **Image size** | **CPU** | **CUDA** |
| 512x512 | 68ms | 11ms |
| 1024x1024 | 301ms | 72ms |
| 4096x4096 | 502ms | 298ms |
| 8192x8192 | 1864ms | 811ms |

*Figure 5: CPU vs. CUDA comparison data.*



*Figure 6: Figure 5 data presented in a bar graph. Green is CPU code, blue is CUDA code. Times are in milliseconds.*

In order to get the timing information, we used *cudaEventElapsedTime* for the GPU version, and *QueryPerformanceCounter* on the CPU version. In CUDA, in order to get accurate timing information, you must first create two CUDA events, one for the start time and one for the end. Once the events are created, you must call *cudaEventRecord* with the start time. After the work is done, you should call *cudaEventRecord* again with the end time. Then we call *cudaEventSynchronize* to make sure the call to *cudaEventRecord* has finished. Finally, we subtract the end time and the start time, and output that number in order to get the milliseconds elapsed.

## Conclusion

As presented in figure 6, the CUDA code is significantly faster than the CPU version. This is because of the nature of the problem, which is calculating a lot of similar data which doesn’t have a lot of interconnectivity between the pieces. This allows the data to be mass calculated on the GPU, which can run a significantly higher number of threads than the CPU concurrently.

The CUDA version could have been even faster than it was. The main issue with the CUDA version was the transferring of memory back and forth between the GPU and CPU. There is also some overhead to setting up the *img\_data* pointer information for the GPU.

Even despite some limitations from the CUDA code, it still easily outperformed the CPU version for this task.

## Appendix 1 – Threaded CPU Mandelbrot function

|  |
| --- |
| typedef char unsigned uint8\_t;  typedef struct rgb { uint8\_t r, g, b; } rgb\_t;  struct mandelbrot\_data\_t {  int start, thread\_cnt;  double scale;  rgb\_t \*\*row\_ptrs;  int width, height;  };  static DWORD \_\_stdcall thread\_proc(LPVOID param) {  mandelbrot\_data\_t \*mandelbrot\_data = (mandelbrot\_data\_t \*)param;  int width = mandelbrot\_data->width;  int height = mandelbrot\_data->height;  int thread\_cnt = mandelbrot\_data->thread\_cnt;  double scale = mandelbrot\_data->scale;  rgb\_t \*\*row\_ptrs = mandelbrot\_data->row\_ptrs;  double cx = -0.6, cy = 0.0;  for (int i = mandelbrot\_data->start; (i < height); i += thread\_cnt) {  double y = (i - height / 2) \* scale + cy;  for (int j = 0; (j < width); ++j) {  rgb\_t \*px = row\_ptrs[i] + j;  double x = (j - width / 2) \* scale + cx;  double zx, zy, zx2, zy2;  //px += j;  zx = hypot(x - .25, y);  uint8\_t iter = 0;  if (x < zx - 2 \* zx \* zx + .25) {iter = 0xFF;}  if ((x + 1) \* (x + 1) + y \* y < 1 / 16) {iter = 0xFF;}  zx = zy = zx2 = zy2 = 0;  do {  zy = 2 \* zx \* zy + y;  zx = zx2 - zy2 + x;  zx2 = zx \* zx;  zy2 = zy \* zy;  } while ((iter++ < 0xFF) && (zx2 + zy2 < 4));  px->r = iter;  px->g = iter;  px->b = iter;  if (px->r == 0xFF || px->r == 0) {  px->r = 0; px->g = 0; px->b = 0;  } else {  uint8\_t uc = px->r % num\_shades;  \*px = mapping[uc];  }  }  }  return(0);  } |

## Appendix 2 – Calculate Mandelbrot on 2D CUDA kernel

|  |
| --- |
| \_\_global\_\_ void calculate\_mandelbrot(int width, int height, double scale, rgb\_t \*\*row\_ptrs) {  int i = threadIdx.x + blockIdx.x \* blockDim.x;  int j = threadIdx.y + blockIdx.y \* blockDim.y;  double cx = -.6, cy = 0;  double y = (i - height / 2) \* scale + cy;  rgb\_t \*px = row\_ptrs[i];  px += j;  double x = (j - width / 2) \* scale + cx;  double zx, zy, zx2, zy2;  uint8\_t iter = 0;  zx = hypot(x - .25, y);  if (x < zx - 2 \* zx \* zx + .25) {iter = 0xFF;}  if ((x + 1)\*(x + 1) + y \* y < 1 / 16) {iter = 0xFF;}  zx = zy = zx2 = zy2 = 0;  do {  zy = 2 \* zx \* zy + y;  zx = zx2 - zy2 + x;  zx2 = zx \* zx;  zy2 = zy \* zy;  } while ((iter++ < 0xFF) && (zx2 + zy2 < 4));  px->r = iter;  px->g = iter;  px->b = iter;  if (px->r == 0xFF || px->r == 0) {  px->r = 0; px->g = 0; px->b = 0;  } else {  uint8\_t uc = px->r % num\_shades;  \*px = mapping[uc];  }  } |