

William Taylor | Compute Games Technology | February 12, 2017

Mandelbrot CUDA Assignment

coursework part 1

Table of Contents

[Introduction 2](#_Toc474748661)

[Problem 2](#_Toc474748662)

[Hardware 2](#_Toc474748663)

[Software 2](#_Toc474748664)

[Goals 3](#_Toc474748665)

[CPU Analysis 4](#_Toc474748666)

[Analysis 4](#_Toc474748667)

[Timings 5](#_Toc474748668)

[Profiling 6](#_Toc474748669)

[GPU Implementation 7](#_Toc474748670)

[Port Overview 7](#_Toc474748671)

[Optimisations 7](#_Toc474748672)

[Timings 7](#_Toc474748673)

[Project Results 8](#_Toc474748674)

[Testing 8](#_Toc474748675)

[Comparison 8](#_Toc474748676)

[Conclusion 8](#_Toc474748677)

# Introduction

## Problem

A snippet of code that renders a section of the Mandelbrot set into an image was given. However, the code is sequential and the code could be much improved by using parallel compute to generate the image. In this report I set out the hardware and software I used, the steps I took to port the code to CUDA and conclude how the approach I took resulted in a more efficient program that would not only be faster but scale better with larger image sizes.

## Hardware

As I am not using university lab equipment I thought I would have a quick run through of the hardware on my laptop which is what I will be benchmarking on. My laptop is equipped with a top of the line GPU and CPU. The CPU is a sixth generation core i7 and is still one of the fastest mobile processors out right now. While it is one generation behind the newly released 7th generation intel processors this CPU is no slouch and will make sure any GPU solution gets a run for its money. My GPU is an NVidia 970m while not a workstation card like the ones found in the labs is still a very fast card beating out an NVidia 960 desktop class graphics card. So there will be plenty of power to exploit using CUDA.



## Software

Porting the given code will be done with the following software.

* Visual Studio
* CUDA Toolkit 8.0
* GitHub
* Visual Studio Profiling Tools
* NSight NVidia Profiler

## Goals

For the project I set some goals on which I could judge its success. The first goal was to output bit for bit the same image. Accuracy was of incredible important as a faster version is only applicable in real world use if it outputs the same image. Secondly a significant speed improvement is desired, this should be a certainty as the sequential CPU code will not be touched. Finally, the interface for running the program should be the same. Currently the program accepts two arguments from the program arguments. These arguments represent width and height for the outputted image, the CUDA port of this program will work in the same way for consistency. Following these goals will result in an executable that behaviors exactly as the original, produces an identical result but does so in a fraction of the time thus proving the advantage of CUDA in speeding up intensive applications.

Identical Image Output- Large Speed Improvement- Identical Interface

# CPU Analysis

## Analysis

My first task was in doing a quick analysis of the given code to look for immediate performance improvements. One of the first things identified was a double iteration.



An unneeded row vector was also identified which wasn’t really needed as all it did was point to section of the image. While this would have little effect on the original code, once I ported it to CUDA it would mean less memory to transfer on the device and fewer calls to the CUDA API.



Additionally, there was a stack allocated array which at least in debug mode where no optimizations where used made the code slower as this array was being allocated every time a color was set for the image. However as expected it was optimized out when optimization flags were turned on.



When writing the file to disk the image is reversed and written in chunks. While the image will still need to be saved like this in the CUDA version to match the output for the given code, a much better improvement would be to have the code that generates the image to do it in reverse instead and then it can be directly written to disk rather than writing it in chunks.



Finally, there was some missing if blocks which could have stop unneeded code from executing. If the first if statement validates to true it does not need evaluate the second if statement or enter the do while block. While a tiny improvement it is, it is a small improvement that could go a long way.



## Timings

For timings I took each function and benchmarked it with various image sizes and took the average from 10 runs of the function. All optimizations were turned on and was compiled as a 64bit executable. Below you can find the breakdown of the results. As we can see the most expensive parts of the application are the ***calc\_mandel*** function and the ***screen\_dump*** function as expected. With this assumption proven correct and backed up by sufficient evidence I then when to profile the code to look at the most expensive sections of the code to have an idea of the gains to be made by the enhancements I had identified earlier and with the added benefit of utilizing CUDA to parallelize the code where needed.

## Profiling

To profile the sequential code, I was relying on the profiling tools inside Visual Studio. Not only do they present a nice report breaking down the function timings but it also highlights the code in the editor which is handy. Running the given code through the profiler produced predictable results.

### Sequential Bottleneck

The biggest bottleneck identified is what I am calling the ‘Sequential Bottleneck’. The issue is that code highlighted on the right is performed sequentially for each pixel in an image. However, as it doesn’t reply on input data or adjacent elements in the image it is perfectly possible to do this operation in parallel rather than sequentially. It also explains why the program doesn’t scale well as the number of operations scale exponentially with larger image sizes. For a 256x256 image this operation is executed 196608 times but for a 4096x4096 image this is 16777216 times. With the code ported to CUDA we will be able to parallelize this operation and no longer be bound by this limitation.

### Double Iteration

Another big bottleneck that came up in the profiler was the double iteration behavior of the code that will perform two loops across a given image. The first loop calculates the index for a color from a map the next loop then goes through the image again and assigns it a color from the map based on the value currently contained within that pixel which is the index from the first loop. Obviously these two loops can be merged and by doing so we will see a large gain in performance.

### File Chunks

The final bottleneck found in the application was due to how the file was being written to disk. In the given code the image is reverse with each row being written at a time rather than the entire file being flushed in one call. It would be much faster just to write the entire image to disk rather than just write each row individual till the full image has been written.

This however brought about a problem. While the image looked the same if you were to write the entire image at once instead of reversing it, it was not. Doing a simple binary check on the output reveals that the output image is not mirrored horizontally. Thus to ensure the output remains the same while writing the image in one go, the given code will need to be adapted so it calculates the flipped image and write that to disk rather than the normal image then write it to disk flipped on its Y axis. By doing so we will remove this small bottleneck and instead of writing rows individually to disk do it all in one go which will certainly be faster.

### Action

With these bottlenecks identified they would be removed in the CUDA port through the following steps. First the sequential bottleneck would be solved by parallelizing the calculation of each pixel’s value. The double iteration would be solved by merging the two loops into one. Finally, when writing the output, it would be done in one go by calculating the image in reverse so it doesn’t need to be written in reverse.

# GPU Implementation

## Port Overview

### Memory

### Kernel

### Output

## Optimisations

### Kernel Launch

Paragraph on auto launch parameters

### Early Exit

Paragraph on early exits

### Flags

Compiler flags

## Timings

Old vs Fastest

# Project Results

## Testing

Testing the output results was done with some unit tests and a windows utility

## Comparison

Sequential vs Parallel

## Conclusion

How awesome cuda is