

|  |
| --- |
| Coursework Report |
| By B00235610 |
|  |
| This report sets out how a given C++ application was optimized using various optimization techniques and the GPU API CUDA. |
| Table of Contents  [The Problem 2](#_Toc473932083)  [Hardware 2](#_Toc473932084)  [Software 2](#_Toc473932085)  [First Analysis 3](#_Toc473932086)  [Benchmark 4](#_Toc473932087)  [Profiling 5](#_Toc473932088)  [CUDA Port 6](#_Toc473932089)  [CUDA Benchmarks 6](#_Toc473932090)  [CUDA Optimizations 6](#_Toc473932091)  [Conclusion 6](#_Toc473932092) The 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  * Visual Studio * CUDA Toolkit 8.0 * GitHub * Visual Studio Profiling Tools * NSight NVidia Profiler   C:\Users\B0023\Desktop\Work\accelerator-programming\docs\cpu-z.pngC:\Users\B0023\Desktop\Work\accelerator-programming\docs\gpu-z.png |

# First Analysis

The first part of optimizing the application was to do a quick run through looking at various areas that can be improved before the port to CUDA. The first spotted was a double iteration that was not needed. The following loop can be merged with loop that lies above as its operation does not rely on adjacent elements in the array.



The second section was to remove an unneeded row vector 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.



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.



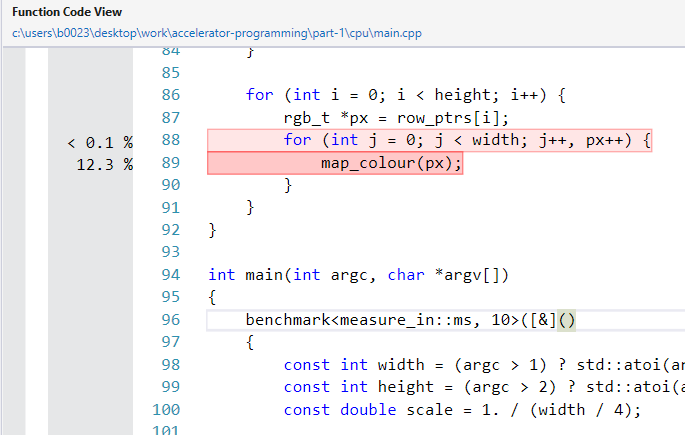
# Benchmarks

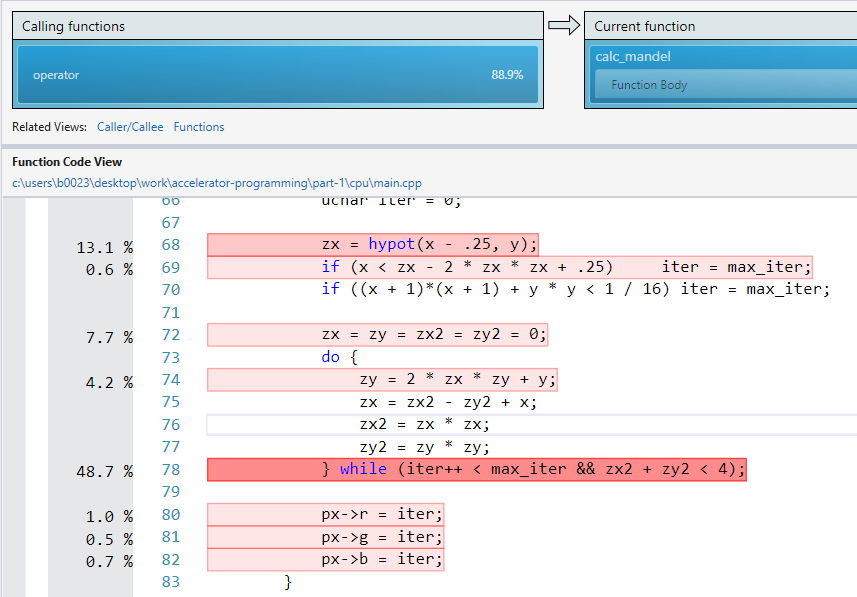
To better understand the scale of the challenge I benchmarked the given code with all optimizations on. The benchmarking was made possible with a small function I personal wrote that times a function call and outputs it to a CSV file for viewing.

Extra Small (256x256), Small (512x512), Medium (1024x1024), Large(4096x4096)

As we can see from the above tables this code does not scale well when it comes to big image sizes. The larger the image the bigger the performance penalty as should be expected with basic sequential code.

# Profiling

Before moving the code over to CUDA I thought it best to actually identify the expensive operations using Visual Studios built in profiler. Running the profiler did nothing but confirm previous suspicions. The double iteration comes up in the profile as can be seen in the below screenshot.

The call to hypot was also shown in the benchmark and this isn’t a surprise either. As the number of times this function is called is entirely based on the amount of pixels in the generated image. Ideally we should see this and the iteration loop actually fall off as a potential bottleneck once these operations are done in parallel as opposed to sequentially.

# CUDA Port

Porting the code to CUDA was actually a twostep process. Firstly, I took the sequential code and got it running in a kernel that would run across one thread and one block before breaking the code down to take advantage of the parallel compute capability of CUDA. This however was the start of a major problem thanks to the Windows operating system. Windows has a feature called TDR or Timeout Detection & Recovery. The idea is that if a GPU doesn’t respond within a giving amount of time it restarts the driver thinking it crashed. So when running my kernel with large images it crashed as TDR kicked in and stopped my kernel executing. So I went into the registry and set the TDR delay to 10 seconds which stopped this issue. After this I was able to break down the sequential code to code that could run in parallel.

# CUDA Optimizations

The first set of optimizations were to implement the bottlenecks found in my first analysis. Removing the double iteration, stack allocated pixel mappings and an unneeded vector that seems to be used out of convenience. Constants were marked as constant memory to utilize GPU caching to enable faster reads also due to the very little amount of constants it doesn’t breach the constant size limit of 64KB. Finally, I played with the block and thread size looking for the correct amount of work allocation that leads to a fast executing kernel. Below you can see how the optimization resulted in a big performance bump.

# CUDA Benchmarks

Benchmarks

# Comparison & Conclusion

As you can see from below graph the CUDA code is more optimal and scales better with larger image sizes.