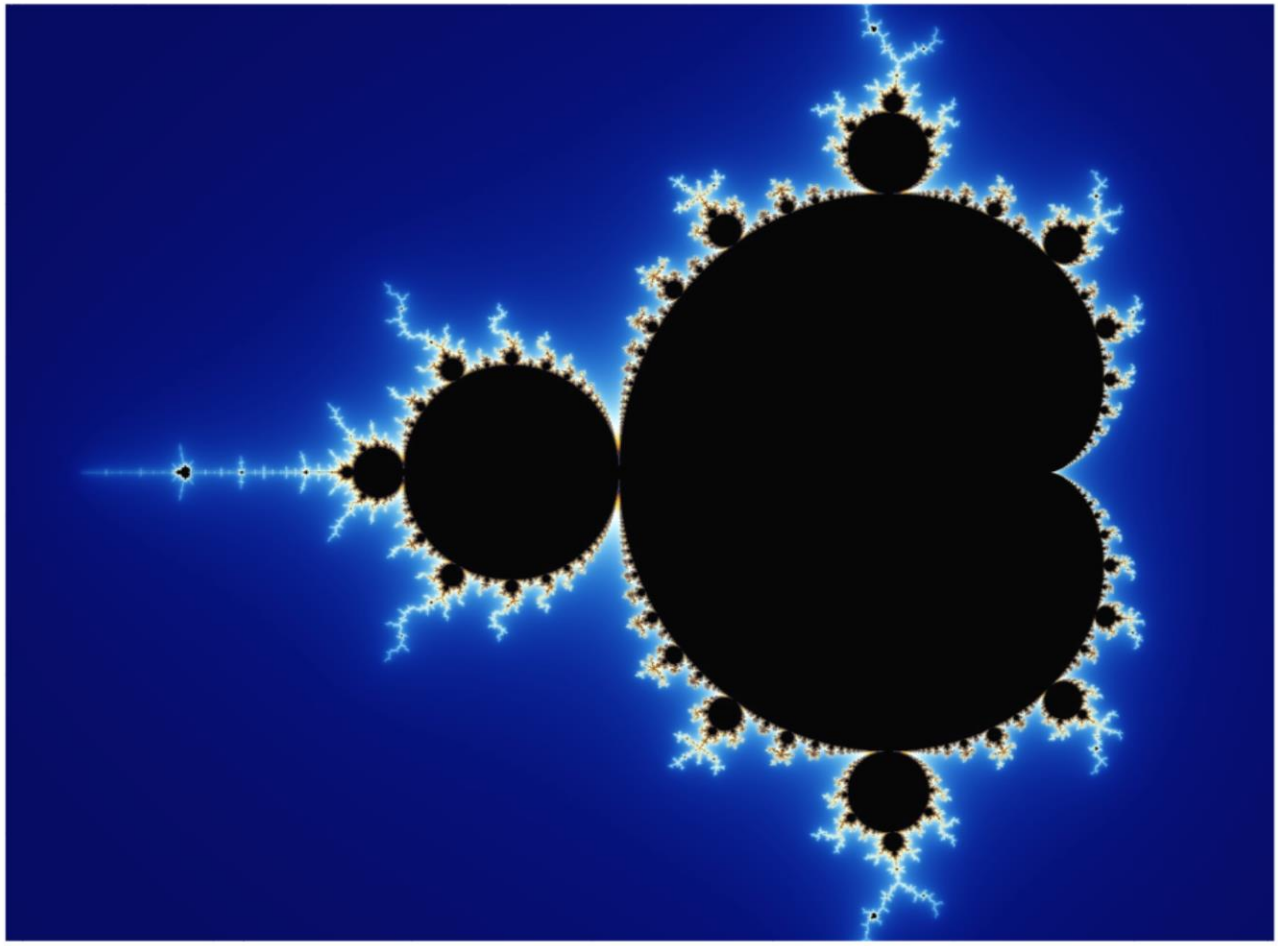


Mandelbrot CUDA Assignment

COURSEWORK PART 1

[Github Repository]

William Taylor | Computer Games Technology | February 12, 2017

Table of Contents

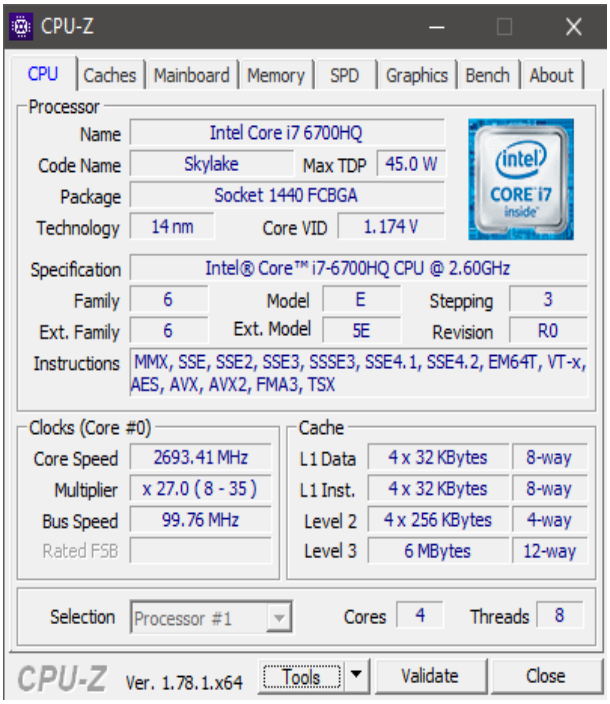
Introduction

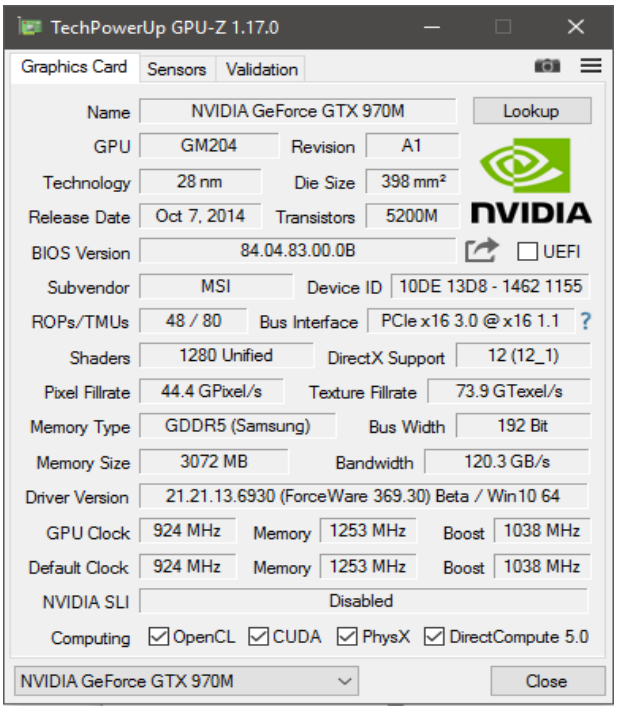
PROBLEM

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 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.

HARDWARE

I am not using university lab equipment so below are snapshots of the hardware on my laptop. 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 which is not a workstation card like the NVidia Quadro K4000 which is the card found in the labs. However, the 970m is a very fast card beating out an NVidia 950 desktop class graphics card. The 970m also comes with additional CUDA cores 1280 compared to the K4000 which has 768 allowing more computations to be done in parallel.





SOFTWARE

To write the software Visual Studio will be used as its built-in profiling tools will help identify expensive sections of the given code with relative ease. The code written will be maintained using Git for version control and is hosted online at Github so issues are tracked well. Obviously as I am writing CUDA code I will be using the latest version of the CUDA Toolkit and I will be using the NSight NVidia Profiler to help profile the CUDA kernels written.

- Visual Studio
- CUDA Toolkit 8.0
- GitBash + GitHub
- 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 which 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 behaves 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 involved a quick analysis of the given code to look for immediate performance improvements that could serve as the starting point for the CUDA version. One of the first things identified was a double iteration that was unnecessary.

```
for (int i = 0; i < height; i++) {
    rgb_t *px = row_ptrs[i];
    for (int j = 0; j < width; j++, px++) {
        map_colour(px);
    }
}
```

An unneeded row vector was also identified which wasn't really needed as all it did was point to a 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 on the device and fewer calls to the CUDA API. This is because an additional row pointers vector, would need to be allocated and transferred onto device memory.

```
rgb_t **row_ptrs;
rgb_t  *img_data;
```

Additionally, there was a stack allocated array which at least in debug mode where no optimizations were used made the code slower as this array was being allocated every time a colour was set for the image. However as expected it was optimized out when optimization flags were turned on. Either way putting this code out of the local function scope would be an improvement.

```
void map_colour(rgb_t * const px)
{
    const uchar num_shades = 16;
    const rgb_t mapping[num_shades] =
    { { 66,30,15 },{ 25,7,26 },{ 9,1,47 },{ 4,4,73 },{ 0,7,100 },
    { 12,44,138 },{ 24,82,177 },{ 57,125,209 },{ 134,181,229 },{ 211,236,248 },
    { 241,233,191 },{ 248,201,95 },{ 255,170,0 },{ 204,128,0 },{ 153,87,0 },
    { 106,52,3 } };

    // *px = mapping[index];
}
```

When writing the file to disk it writes it in reverse and writes it 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.

```
void screen_dump(const int width, const int height)
{
    FILE *fp = fopen("cpu-mandelbrot.ppm", "w");
    fprintf(fp, "P6\n%d %d\n255\n", width, height);
    for (int i = height - 1; i >= 0; i--)
        fwrite(row_ptrs[i], 1, width * sizeof(rgb_t), fp);
    fclose(fp);
}
```

In areas, there were a lot of hard written math code that could be made more readable by using standard math functions and there were some areas where values could be pre-calculated resulting in less computations. Not expecting any massive gains from this but more readable code and less operations is always advantageous.

```
// x+1 * x+1 -> pow(x+1, 2)
// 1 / 16 = 0.0625
if ((x + 1)*(x + 1) + y * y < 1 / 16) iter = max_iter;
```

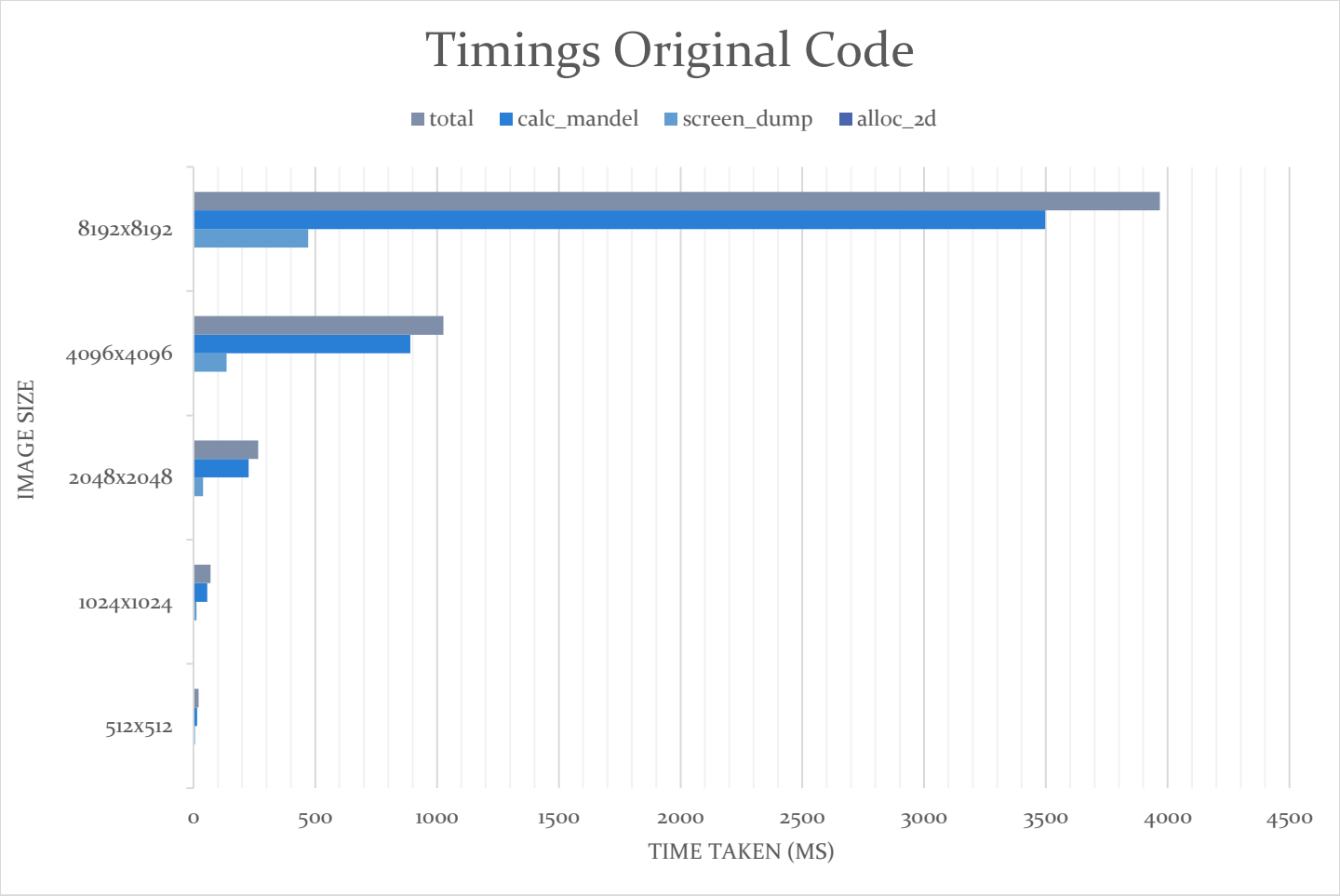
Finally, there was some missing if blocks which could have stop unneeded code from executing. In the below code if the first ***if*** statement validates to true it does not need evaluate the second ***if*** statement or enter the ***do while*** block as if ***iter*** is either 255 or 0 the output is a black pixel. So, by avoiding these unneeded computations we can make the code more efficient. While a tiny improvement it is, it is a small improvement that could go a long way.

```
zx = hypot(x - .25, y);
if (x < zx - 2 * zx * zx + .25)     iter = max_iter;
if ((x + 1)*(x + 1) + y * y < 1 / 16) iter = max_iter;

zx = zy = zx2 = zy2 = 0;
do {
    zy = 2 * zx * zy + y;
    zx = zx2 - zy2 + x;
    zx2 = zx * zx;
    zy2 = zy * zy;
} while (iter++ < max_iter && zx2 + zy2 < 4);
```

TIMINGS

For timings, I took each function and benchmarked it with various image sizes and took the average from ten 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 these assumptions proven correct and backed up by sufficient evidence I then went to profile the code. The aim was to find the most expensive sections of the code to have an idea of the gains to be made by the enhancements I had identified earlier.



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.

```
            const double x = (j - width / 2) * scale + cx;
            double zx, zy, zx2, zy2;
            uchar iter = 0;

11.2 %      zx = hypot(x - .25, y);
 0.5 %      if (x < zx - 2 * zx * zx + .25)      iter = max_iter;
            if ((x + 1)*(x + 1) + y * y < 1 / 16) iter = max_iter;

 6.2 %      zx = zy = zx2 = zy2 = 0;
            do {
 4.2 %          zy = 2 * zx * zy + y;
                zx = zx2 - zy2 + x;
                zx2 = zx * zx;
                zy2 = zy * zy;
55.3 %      } while (iter++ < max_iter && zx2 + zy2 < 4);

 0.9 %      px->r = iter;
 0.5 %      px->g = iter;
 0.8 %      px->b = iter;
```

Double Iteration

Another big bottleneck that came up in the profiler was the double iteration behaviour of the code that will perform two loops across a given image. The first loop calculates the index for a colour from a map. The next loop then goes through the image again and assigns it a colour 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.

```
 0.9 %              px->r = iter;
 0.5 %              px->g = iter;
 0.8 %              px->b = iter;
                }
            }

            for (int i = 0; i < height; i++) {
                rgb_t *px = row_ptrs[i];
< 0.1 %     for (int j = 0; j < width; j++, px++) {
13.5 %          map_colour(px);
                }
            }
        }
```

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 output image is in reverse with each row being written individually rather than the entire image being flushed in one call. It would be much faster just to write the entire image to disk rather than just write each row individually till the full image has been written.

```
rgb_t **row_ptrs;
rgb_t  *img_data;

void screen_dump(const int width, const int height)
{
< 0.1 %     FILE *fp = fopen("cpu-mandelbrot.ppm", "w");
            fprintf(fp, "P6\n%d %d\n255\n", width, height);
            for (int i = height - 1; i >= 0; i--)
 7.0 %          fwrite(row_ptrs[i], 1, width * sizeof(rgb_t), fp);
            fclose(fp);
}
```

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 writes that to disk rather than the normal image then write it to disk reversed. 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 while maintaining an identical output to the original code.

Action Taken

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 in chunks.

GPU Implementation

PORT OVERVIEW

Memory

The first part of the port involved taking data that would need to be accessed on the device via a kernel and making it available. This is because all memory that is accessed by the GPU must be in device memory, so any traditional memory in the host needs to be transferred. This was done just by making the data constant memory. This would yield benefits as constant memory is always cached allowing for fast reads on the device and it doesn't need to be transferred via ***cudaMemcpy*** calls.

```
__constant__ const uint8_t CharMax = std::numeric_limits<uint8_t>::max();
__constant__ const uint32_t ColoursSize = 16;
__constant__ const rgb_t Colours[ColoursSize]
{
    // … colour mappings
};
```

Additionally, global memory was allocated which is used to store the image that will then be written to disk. I use my own little helper class to help manage this for me. But what it does is very simple. It allocates global memory then sets a default value for it and cleans it all up once the object is destroyed. Basic C style arrays were abandoned in favour of a standard C++ vector which would handle both allocation and deletion for me on the host.

```
// Host image
std::vector<rgb_t> hostMemory(height * width);
// Device Memory
cuda::memory<rgb_t*> deviceMemory { hostMemory.size() * sizeof(rgb_t), 0 };
```

Kernel

The key section of the port was to parallelize the operation that calculates the output colour so instead of sequentially calculating each pixel it can now be done in parallel resulting large performance gains. The kernel is very simple it generates an index for the section of the Mandelbrot set it will work on. Once it has calculated the value it stores this in the device memory explained above.

```
__global__ void mandelbrot(cuda::launchInfo info, rgb_t* image, double scale)
{
    const auto j = static_cast<int>(threadIdx.x + blockIdx.x * blockDim.x);
    const auto i = static_cast<int>(threadIdx.y + blockIdx.y * blockDim.y);

    // … Calculate Output
}
```

Output

Once the output has been calculated and all values have been written to the global memory allocated we just move the device memory back into host memory so it can be written to disk. Once it has I just flush the memory to disk without reversing the data as the kernel generates the image in reverse order so there is no need to write it in reverse order. With all these components in place I had I CUDA program that would produce the image that was required. However, this implementation was very basic and further optimizations were made to ensure the code was as performant as possible before comparing it to the given CPU only code.

```
// move device memory into host memory
cuda::move(deviceMemory, hostMemory.data());

writeOutput("gpu-mandelbrot.ppm", hostMemory.data(), width, height);
```

OPTIMISATIONS

Kernel Launch

In my first implementation kernel launches were one dimensional meaning larger image sizes could not be generated as it breached the max block or grid size. To overcome this issue launching the kernel was rewritten to do a two-dimensional launch allowing the program to generate larger image sizes. Additionally, I used the CUDA Occupancy API to get an estimate for the grid and block size parameters for the given kernel. One shortfall of this API is that it is designed for one dimension workloads so I had to write some additional code that converted this estimation so it could be used in a two-dimension kernel launch.

```
cuda::launchInfo cuda::optimumLaunch(void* kernel, int width, int height, int dataLength)
{
    auto minGridSize = 0, blockSize = 0;
    auto cudaError = cudaOccupancyMaxPotentialBlockSize(&minGridSize, &blockSize, kernel, 0, 0);

    if(cudaError != 0)
    {
        throw std::runtime_error("cudaOccupancyMaxPotentialBlockSize failed");
    }

    const auto blocks = static_cast<int>(pow(2, ceil(log(sqrt(blockSize)) / log(2))));
    const auto grid = static_cast<int>((sqrt(dataLength) + blocks - 1) / blocks);

    return { dim3(grid, grid), dim3(blocks, blocks), width, height };
}
```

Early Exit

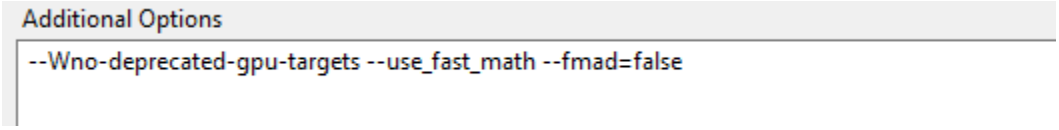
In the first implementation, the kernel would never exit early and continue to the end of the function. To stop this happening the if statements were reworked to return immediately if they were met. This stops the kernel executing a single loop of the do while loop if any of these conditions are true. Additionally, the global memory that stores the output image is initialized to black pixel values to remove the need for the kernel to write black pixels to the image. This is what makes this early exit possible.

```
if (zx - 2 * pow(zx, 2) + Unknown >= x)
{
    return;
}

if (pow(x + 1, 2) + pow(y, 2) < Threshold)
{
    return;
}
```

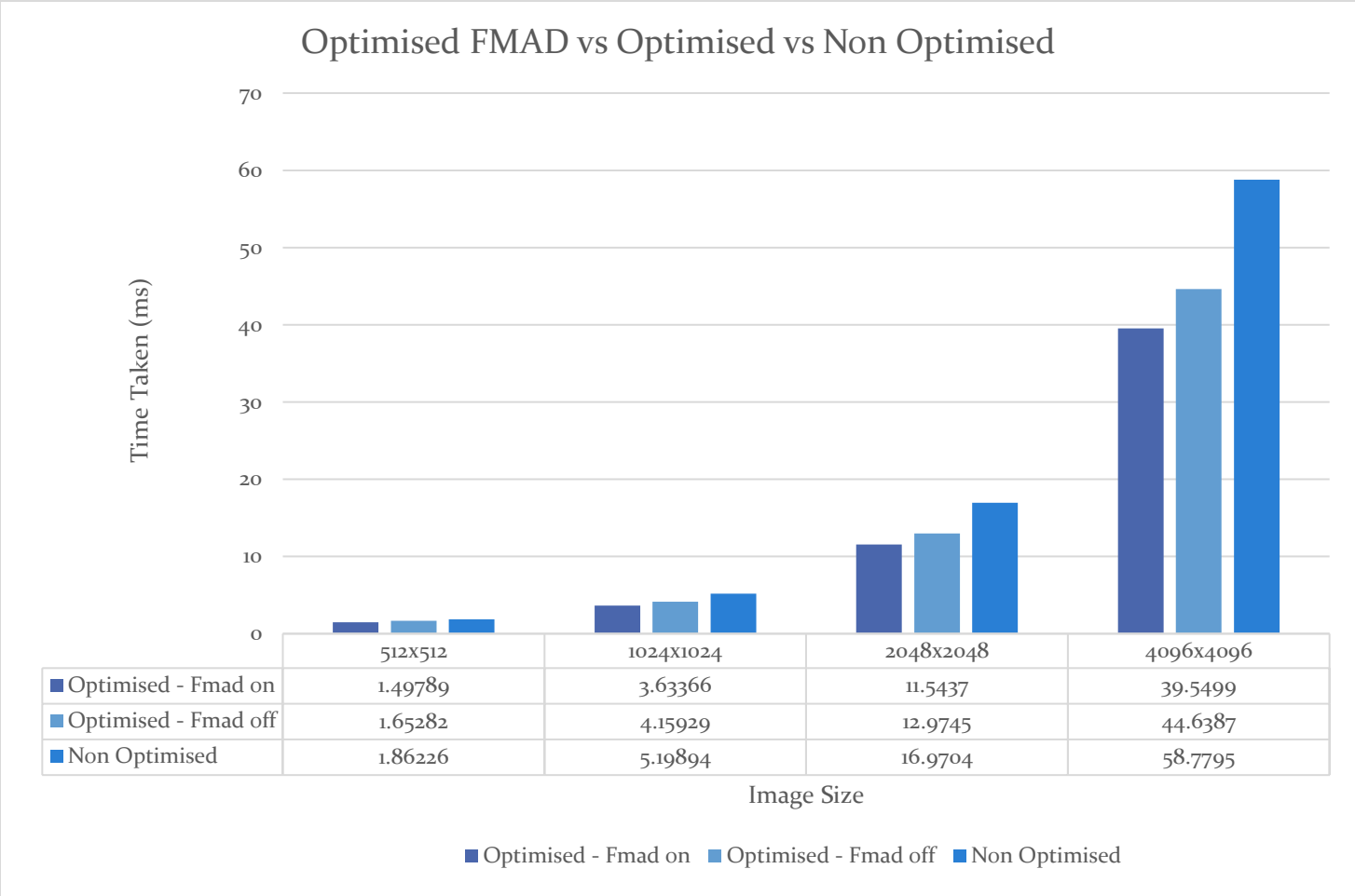
Flags

I also looked at any additional flags to increase performance, –use_fast_math was used however the FMAD optimization was turned off. This was because this optimization resulted in less accurate results as it fuses a floating point multiply and add into a single operation to improve performance. FMAD involves rounding the final sum of the operation to accomplish its goal resulting in inaccuracies. FMAD is turned on by default in release builds and if you specify -use_fast_math it is turned on as well so by providing -fmad=false I took a performance hit but I reached my goal of an identical output to the CPU version. Other flags enabled by -use_fast_math was left on. Finally, I upgraded the compute version and architecture to ensure there were no limits on active block size and active thread size.



TIMINGS

The following optimizations resulted in a significant speed up as can be seen in the below table. The given times are only for the kernels executions and timings were recorded using the CUDA event API. It is also worth noting that larger image sizes were not tested as the original version could not handle large image sizes e.g. 8k because the kernel was launched with a one-dimensional vector. As we can see turning off FMAD resulted in a slower kernel however the optimized version is still faster than the original and unlike the original it can now compute larger output sizes so it is still a welcome improvement. Another thing to note is that the larger the image the larger speed improvement we get from the optimized version of the kernel which is great as in the final section we will be comparing the CPU version with the CUDA version with larger image sizes.

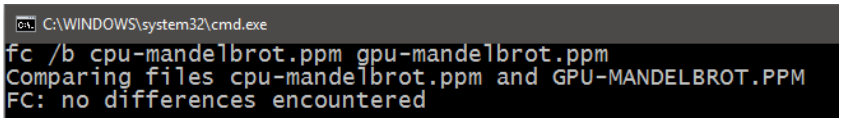


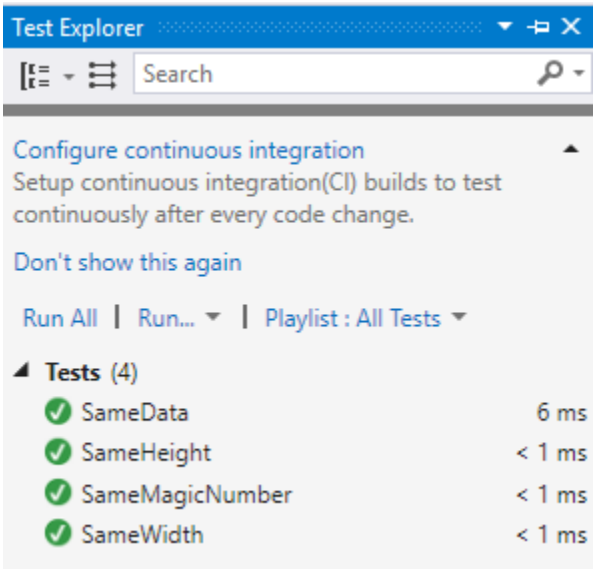
	512x512	1024x1024	2048x2048	4096x4096
Optimised - Fmad on	1.49789	3.63366	11.5437	39.5499
Optimised - Fmad off	1.65282	4.15929	12.9745	44.6387
Non Optimised	1.86226	5.19894	16.9704	58.7795

Project Results

TESTING

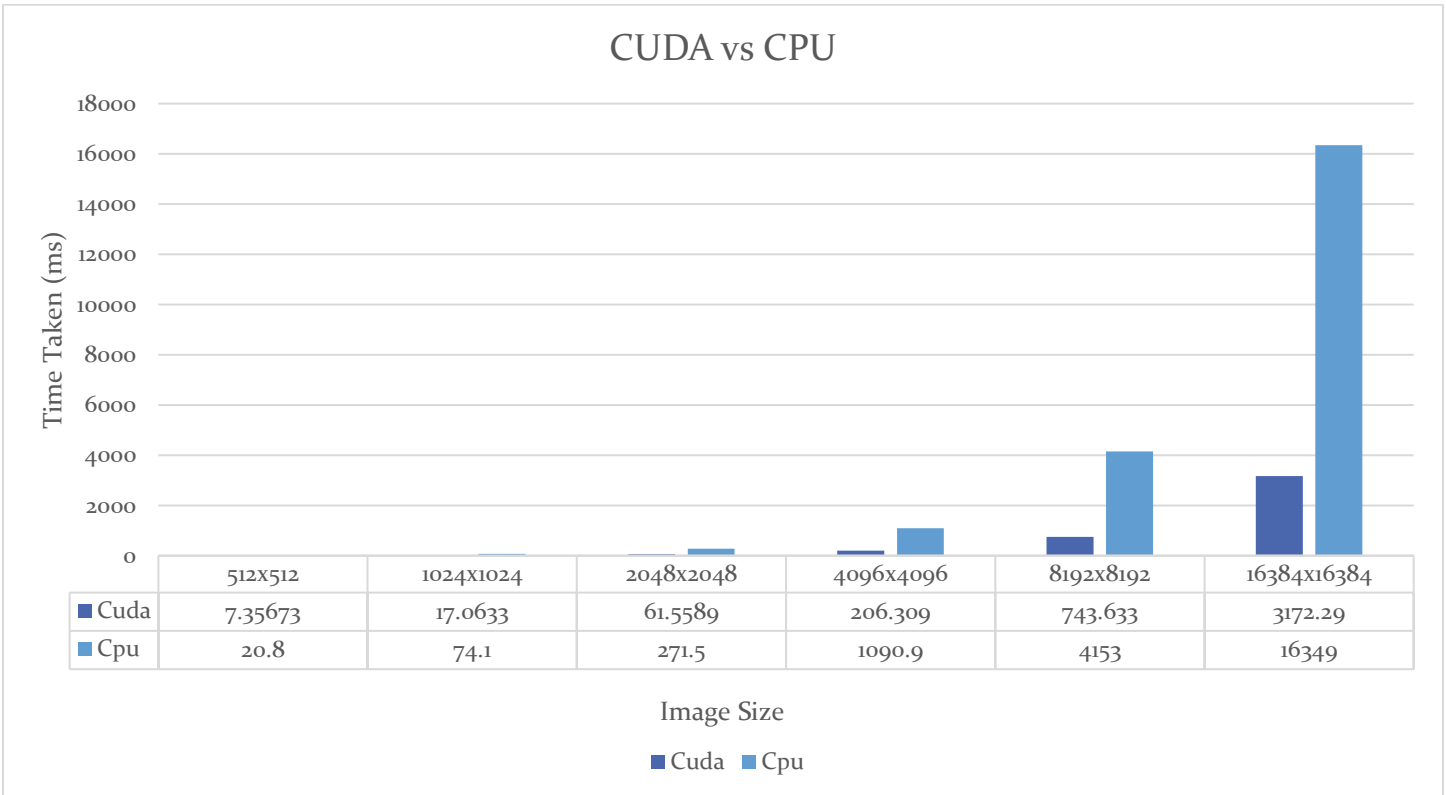
Throughout this project outputs were tested thoroughly to ensure the output is the same as the given code as it was one of my own key goals. The first test was some C++ unit tests I wrote using Visual Studio's unit testing framework to make sure the data in the files matched bit for bit. There were also additional tests for checking the size so if the same data test fails I can double check to make sure I got both programs to output the same image size. A nice feature of this testing framework is it integrates into Visual Studio very well. To ensure that a false test doesn't make its way through or in case my unit tests were broken I also used the file compare utility in binary mode found on Windows to double check my results.





COMPARISON

With the project done I decided to compare the given CPU version with my optimized CUDA version. In this comparison, I took the average total execution time of both program types and compared their performance with a set of six output image sizes ranging from 512x512 to 16384x16384. I took the average from ten runs rather than a single run to ensure a one-off timing did not corrupt the data recorded. I also threw a call to ***cudaResetDevice*** in-between each CUDA program invocation to ensure CUDA initialisation was considered. Remarkably the CUDA version of the program was faster with every output image size by a large margin which is not something I was expecting. What is more interesting however is that the CUDA version can output a 16k image faster than the CPU version can output an 8k image, as well as output two 512x512 images faster than the CPU version can output one. This just goes to show how much faster the CUDA version is thanks to its ability to take advantage of parallel compute and the various additional optimizations I added to ensure it is as fast as possible.



DISADVANTAGES

There are disadvantages both to the approach I took and the CUDA program which are worth discussing. Firstly, the written program used CUDA which will only ever run on CUDA enabled hardware. Conversely if the program was written in OpenCL it would have run on a wider range of devices as OpenCL kernels can run on any heterogeneous system. Additionally, my approach did not look at exploiting parallel compute capabilities of the CPU. There was no attempt made to optimize the CPU version to make good use of threads and SIMD instructions. The result was a comparison on an optimized CUDA version and a simple CPU version rather than a fair comparison between the fastest available version for each type of processor. Finally, when launching the kernel, I don't not optimize for each image size. This is an issue as it means the speed shown above is actually worse than could have been achieved if I calculated an optimized kernel launch for each image size. This was done so the program would work generally well with any image size given to it rather than leading to some cases where the kernel was extremely fast but only for certain data sets. I preferred a constant average rather than exceptional performance with a certain data set.

CONCLUSION

As expected the code ported to CUDA resulted in a significant performance increase thanks to the added performance found in parallelizing the code. With a large 16k output image, we saw a 5.1x speed increase and even with the smallest image I tested we saw a 2.8x performance increase. Not only is the code faster but the CUDA code will scale better with even larger images making it better than the original code in every way possible. It outputs the same image bit for bit and is considerably faster meaning I successfully achieved the objectives I set out in the introduction.

Appendix 1 – Chrono Benchmark Method

```
#pragma once

#include <fstream>
#include <chrono>

template<int C>
class csv
{
    std::ofstream output;
public:
    explicit csv(const std::string& name, const std::string& title) :
        output(name.c_str())
    {
        output << title.c_str();
        for (auto i = 1u; i < C; ++i)
            output << ",";
        output << std::endl;
    }

    template<typename V, typename... Args>
    void append(V&& first, Args&&... args)
    {
        append(first);
        append(args...);
    }

    template<typename V>
    void append(V&& v)
    {
        output << v;
    }

    void append_row(const std::string& row)
    {
        output << row.c_str() << std::endl;
    }
};

template<unsigned times, typename Functor, typename... Args>
void benchmark(Functor&& method, Args&&... args)
{
    using namespace std::chrono;
    csv<2> table("benchmark.csv", "Benchmark Results");
    table.append_row("ID, Time (ms)");

    auto total = 0.0, millseconds = 0.0;

    for (auto i = 1u; i <= times; ++i)
    {
        const auto start = high_resolution_clock::now();
        method(std::forward<Args>(args)...);
        const auto stop = high_resolution_clock::now();
        millseconds = duration_cast<milliseconds>(stop - start).count();

        table.append(i, ",", millseconds, "\n");
        total += millseconds;
    }

    table.append_row("Total (ms), Average (ms) \n");
    table.append(total, ",", total / times);
}
```

Appendix 2 – Kernel CUDA Benchmark Method

```cpp
#pragma once

#include <cuda.h>
#include <cuda_runtime.h>
#include <device_launch_parameters.h>

#include "../benchmark.h"

template<unsigned times, typename Functor, typename... Args>
void benchmark(Functor&& method, Args&&... args)
{
    csv<2> table("kernel.csv", "Kernel Results");
    table.append_row("ID, Time (ms)");

    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    auto milliseconds = 0.0f;
    auto total = 0.0;

    for(auto i = 1u; i <= times; ++i)
    {
        cudaEventRecord(start);
        method(std::forward(args)...);
        cudaEventRecord(stop);
        cudaEventSynchronize(stop);
        cudaEventElapsedTime(&milliseconds, start, stop);

        table.append(i, ",", milliseconds, "\n");
        total += milliseconds;
    }

    cudaEventDestroy(start);
    cudaEventDestroy(stop);

    table.append_row("Total (ms), Average (ms)");
    table.append("total", ",", total / times);
}
```

Appendix 3 – CUDA Kernel

```
__global__ void mandelbrot(cuda::launchInfo info, rgb_t* image, double scale)
{
    const auto j = static_cast<int>(threadIdx.x + blockIdx.x * blockDim.x);
    const auto i = static_cast<int>(threadIdx.y + blockIdx.y * blockDim.y);

    if (j >= info.height || i >= info.width)
    {
        return;
    }

    const auto y = (i - info.height / 2) * scale + CenterY;
    const auto x = (j - info.width / 2) * scale + CenterX;

    auto zx = hypot(x - Padding, y);

    if (zx - 2 * pow(zx, 2) + Padding >= x)
    {
        return;
    }

    if (pow(x + 1, 2) + pow(y, 2) < Threshold)
    {
        return;
    }

    double zy, zx2, zy2;
    zx = zy = zx2 = zy2 = 0.0;
    uint8_t iter = 0;

    do
    {
        zy = 2.0 * zx * zy + y;
        zx = zx2 - zy2 + x;
        zx2 = pow(zx, 2);
        zy2 = pow(zy, 2);
    }
    while (++iter < CharMax && zx2 + zy2 < 4.0);

    if (iter != CharMax && iter != 0)
    {
        const auto index = j + info.width * (info.height - i - 1);
        image[index] = Colours[iter % ColoursSize];
    }
}
```