# Mandelbrot

## Work Done

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 I launch a 2 dimensional gpu-thread-thing and get the *i* and *j* variables by calculating their index within the block.

## Create a window

Before beginning the project, the first thing I 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, then it was very easy to get the data. I simple called *CreateWindow*, and then did a for loop for the *x* and *y* of the window, and called *DrawPixel* using the red, green, and blue colour data at that part of the array.

The reason for creating a window was to increase the iteration time. I 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 I 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 I could simply; build the code, run it, and check the image on the screen. I feel like increasing the iteration time this way meant that, over the course of the project, I have saved time by sacrificing some time at the beginning.

## CPU Version

The version of the code which runs exclusively on the CPU was also slightly optimized, in order to try and make the results between the GPU and CPU versions as fair as possible. The CPU-only version launches as many threads as there are cores on the machine, which is 8 on mine. It then does the work across these cores, gaining a large performance increase compared to completely serial code.

## Optimization

The project was designed to run as quickly as possible.

<https://en.wikipedia.org/wiki/Inline_expansion#Implementation>

One of the first optimizations I did was to inline as many of the functions as possible. *Inline expansion* is an important optimization, but it is not one a compiler can guarantee, since the *inline* keyword is just a hint to the compiler. Manually inline-ing the functions meant that there was no overhead to calling them now, where the compiler would have to set up the parameters in registers then execute a *call* instruction. Instead, it could just continue to execute the code.

The next optimization I did was remove excessive memory allocations. Because the variables *cpu\_img\_data* and *cpu\_row\_ptrs* are allocated at the same time, and there is no dependence between them, I decided to change the two memory allocations into one. I simple create a third variable, *void \*mem*, which was allocated to be the size of both the memory allocations for *cpu\_img\_data* and *cpu\_row\_ptrs*. Then I set *cpu\_img\_data* to point to the start of the memory block. I then set *cpu\_row\_ptrs* to point after *cpu\_img\_data*, by setting it to *mem* plus the size of *cpu\_img\_data*. While this was a small optimization, it saved **TIME**.

## Code Style

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 the code is doing, before they fully understand what it is doing.