# Parallelizing the Mandelbrot Set

## Enhancing Performance through Multithreading and GPU Computing

Patrick Leiser
Jared Pugh
Catherine Yaroslavtseva

June 2023

## 1 Abstract

The Mandelbrot Set is a set of complex numbers (c) that exhibits intricate fractal patterns, given the parameter that $f_c(z) = z^2 + c$ where z is iterated from 0. In this paper, we describe the investigation of various possible methods of optimizing the computation and visualization of the Mandelbrot Set, primarily through means of parallelization. The primary purpose of these experiments is to explore different parallel programming environments by expanding on the base code provided by [INSERT]. We also focused on utilizing the strengths of GPUs and CPUs to find an optimal computational balance for various problem sizes.

The source code includes a serial version, an OpenMP version, a CUDA version, and various diverging versions of the latter two.

## 2 Experiments

This project involved a number of experiments, including but not limited to checking the viability of different CUDA block sizes, load distribution methods and mathematical optimizations. While some changes were universal to all included implementations, the majority were isolated in order to accurately test their effects on the overall runtime. Timing information is included within each relevant subsection, as well as a final analysis in the conclusion. Effective changes were compiled into a cumulative final branch (main).

## 2.1 Additional Features

The contents of this section are universal to all implementations of the Mandelbrot Set discussed in this paper. The original serial version of this program rendered monotonous fractals, with pixels distinguished by black, white and shades of gray. The ability to zoom into the image or pan around it was also not included in the original program. Thus, the first stage of this project involved adding coloration to the set and implementing zooming and panning functionality.

Colors in our implementation were calculated through simple multiplication and division operations. Some versions included a colors struct and stored RGB values as 8-bit integers, but the general calculations remained the same:

```
int r = 255 * iter / maxiter;
int g = 255 * iter / (maxiter/30);
int b = 255 * iter / (maxiter/100);
```

These calculations produced primarily blue and green shades for the exterior of the zoomed out version of the set, and yellow for the interiors of the bulbs and cardioids. Ultimately, these color calculations were lower time-complexity than implementations that used trigonometric functions and created visually appealing Mandelbrot sets.

Zoom and pan functionality was added by listening specific keys of keyboard input, such as the 'i' and 'o' keys, with the former zooming in on the set and the latter zooming out on the set. Similar to many video games, the panning is implemented with the 'w', 'a', 's', and 'd' keys, mimicking the layout of a keyboard's arrow keys.

There are 2 levels of zooming and panning, with finer adjustments achieved by using the shift key along with the appropriate input key. Zoom and pan amounts can be further customized in the code as needed, and works by using the set distance or zoom ratio to transform the coordinates displayed on the screen.

The user can also restore the default position (showing the whole Mandelbrot set) with the 'r' key, or reflect the view using shift along with the 'R' key.

## 2.2 Mathematical Optimization

Initially, the sets rendered in parallel contained inconsistencies interference between thread color settings.This was expressed through incorrectly colored pixels rendered at various points of the set. The issue was rectified by creating a critical section around the lines dedicated to drawing the pixels in the window. This solution introduced overhead, which resulted in us decoupling the loops, and performing rendering after calculations were complete.

While implementing these changes, we found that the `cpow` and `cabs` operations were computationally expensive. To help rectify this, and to fix compatibility issues with CUDA and complex numbers, we created an alternative version of the `compute_point()` function. The alternative function performs complex number calculations manually, rather than relying on C complex libraries. It accomplishes this by splitting z into a real and imaginary component, then combining the components as needed for calculations. During this process, we also experimented with using floats instead of doubles throughout the program, but found that this replacement caused rounding errors. The rounding errors imposed limitations on the maximum extent of zoom functionality, so we decided to keep decimal data stored as double data types.

Mandelbrot sets are symmetrical across the x-axis. We exploited this symmetry to avoid redundant calculations through mirroring. This optimization is currently implemented in the cached CUDA version of the project. This version computes the mirror value of the coordinate currently being processed, and caches the same color value for both the original coordinate and the mirrored coordinate.

```
flip_j = (height - my_j) + 1;
```

When the mirror coordinate is processed, it is then able to fetch a color from the cache rather than recalculating it. This modification to the cache reduced the required amount of calculations by half.
In addition to being symmetrical, Mandelbrot sets contain a large interior area that contains the same fill color. This area is contained within bulbs and cardioids. We found that the outlines of these areas are defined by the following equations:

```
Cardioid: cabs(1 - csqrt(1 - 4 * alpha)) <= 1
Bulb:     cabs(1 + alpha) <= 0.25
```

The `compute_point()` function initially checks these conditions to verify whether or not the point being computed is within a bulb or cardioid. If the condition returns true, `compute_point()` simply returns the max iteration value, thus saving time spent on calculations within the function.
The following table shows the timings for the default serial runtime compared to the OpenMP runtime with optimized math, and the CUDA runtime with

optimized math.

| | Serial (no math) | OpenMP | CUDA |
|---|---|---|---|
| Run 1 | 31.32423 | 14.21342 | 0.95468 |
| Run 2 | 31.34534 | 13.1231 | 0.83748 |
| Speedup | 1.00000 | 2.38833 | 37.33530 |

## 2.3 Scheduling and Distribution

This results of this particular set of experiments varied between the OpenMP and CUDA implementations of the project. While the OpenMP version performed multiple pixel computations per thread, the CUDA version performed most optimally with one pixel per thread. Thus, this section will be split into an analysis of the OpenMP and CUDA experiments, respectively.

In OpenMP, through testing and trial and error, dynamic scheduling was deemed to be the most optimal method for scheduling. Dynamic scheduling avoids block distribution (such as the default `static` scheduling), which is prone to immense slowdowns, as some threads that must compute values close to the origin will have a significantly higher workload than others. However, a purely cyclic distribution (`static, 1`) is also suboptimal because, due to the nature of the problem, it does not give the most ideal balance of workload, as some threads may have a longer workload to finish, as others idle upon completing their assigned portion. This behavior leads to the stripe of values along y=0 tending to continue processing long after the rest of the graphic has been calculated and rendered. Dynamic scheduling, with the default blocksize of 1, as defined by the OpenMP standards, solves the flaws of a cyclic distribution while avoiding the serious slowdowns that would be presented by a block distribution with large chunksizes.

The results of our distribution experiments with Cuda are further elaborated on in subsection 2.6. We experimented primarily with assigning varying amounts of pixels per thread. After testing, we determined that one pixel per thread was the most efficient distribution method.

## 2.4 Load Balancing

As the problem size, represented by the number of pixels requiring calculation, increases, the GPU demonstrates a time-saving advantage over both serial and parallel CPU implementations. For smaller problem sizes, where the X11 window fits on a laptop or desktop monitor, a CPU that takes advantage of parallel programming paradigms is likely to outperform a GPU in pixel rendering. However, for larger dimensions, GPUs will generally outperform CPUs.

We noticed this discrepancy and investigated ways to balance the workload by splitting it between the GPU and CPU. Through various tests, we determined

that with a problem size of over 300 million pixels placed between the bounds of (-1.5, -1.0) and (0.5, 1.0) with 3,000 iterations, combining the efforts of a GPU and CPU saved over 10 seconds compared to purely GPU computing. Over 25 seconds were saved compared to the parallel CPU implementation, and several minutes compared to the naive serial implementation. These results were attained by delegating the top 60 percent of the pixels to the GPU, and the bottom 40 percent of the pixels to the CPU. These calculations were timed with an implementation that does not attempt to break out of loops early for (x,y) coordinates within a certain range. These calculations were also not timed using an implementation that takes advantage of symmetry to avoid repeated computations. In an implementation where the problem size of unique computations can be reduced, greater time improvements can be achieved by delegating more of the workload to the CPU. The percentage of the workload assigned to the GPU and CPU can thus be determined dynamically based on the problem size.

## 2.5 Multiple Kernels

Another method we attempted to use to split the workload was to combine OpenMP and CUDA to launch multiple kernels, each with on own CPU thread. In the process, we found that starting kernels in a parallelized for loop resulted in latency, and ultimately did not result in any time savings. Thus, we chose not to pursue this method further.

## 2.6 CUDA Block Sizes

After debugging the CUDA implementation, we tested the efficiency of block sizes of 1, 16 and 32 with the following definitions in place:

```
dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
dim3 dimGrid(width/BLOCK_SIZE, height/BLOCK_SIZE);
```

This was chosen based on a one pixel per thread implementation. While we tested implementations that assigned multiple pixels per thread, we found that in the CUDA version, single pixel assignments facilitated the greatest speedups. In the context of these definitions, we eventually settled on 16x16 block dimensions (256 threads), distributed across the grid in 1x1 blocks. Initially, we had hypothesized that larger block sizes would produce better results, but found that a block size of 16 consistently generated the best times.

## 2.7 Cache

When applying transformations (such as vertical and horizontal shifts) to Mandelbrot sets, many of the computations are redundant. To account for this, we implemented a cache to store the equivalent of a screenful of pixels that do not need to be recalculated. For example, in the context of a 640x480 window, the cache stores a total of 307,200 pixel settings. The cache is implemented as a struct, which stores an array of color struct objects. While the image is being

computed, the program is able to bypass the expensive `compute_point()` function if a valid value is detected for the corresponding index in the cache. Instead, it assigns the cached color value to the index currently being computed. Since each index of the array represents a pixel of the screen, values are accessed by computing the one dimensional index equivalent, given the local i and j values of each thread, resulting in an O(1) cache access time.

```
index = my_i+width*my_j
```

After transformations are applied, the local `i` and `j` values must be adjusted in order to access the correct index. This is done by storing the accumulation of the transformations on each axis along with the coordinates, and calculating the adjusted `i` and `j` values by reversing these transformations.

In addition to the currently implemented cache format, we also explored a color-mapping cache. Instead of storing the colors in association with the locations they were assigned to, this cache stores the colors associated with calculated values of `iter`. However, while this version of the cache requires significantly less storage, it requires a call to the `compute_point()` function for each index, and thus results in significantly lower time savings. This is especially true due to the overall low cost of our color calculations, so we decided not to include this cache implementation in the final version of the project.
The table below shows the difference in render time after shifts between the cached CUDA implemtation, the regular serial version, and the optimized cache version. These tests were performed for a 640x480 window size.

| | Serial | OpenMP | CUDA |
|---|---|---|---|
| Run 1 | 0.72640 | 0.45784 | 0.00134 |
| Run 2 | 0.82345 | 0.63453 | 0.00009 |
| Speedup | 1.00000 | 1.1372 | 77.234 |

# 3 Results

This section contains a detailed timing table that summarizes the speedups and efficiencies that we attained in the most efficient versions of our implementation.
OpenMP implementation - seconds per render:

| | Serial | OpenMP | | | |
|---|---|---|---|---|---|
| Threads (per block) | 1 | 2 | 4 | 8 | 16 |
| Blocks | 1 | 1 | 1 | 1 | 1 |
| Run 1 | 31.422292 | 16.402485 | 8.279810 | 4.18065 | 3.615040 |
| Run 2 | 31.43158 | 16.369762 | 8.265928 | 4.22009 | 4.090118 |
| Run 3 | 31.42617 | 16.378072 | 8.267531 | 4.23872 | 3.670874 |
| Run 4 | 31.448678 | 16.374973 | 8.267116 | 4.20696 | 4.058952 |
| Run 5 | 31.59528 | 16.371561 | 8.272475 | 4.18369 | 3.683804 |
| Minimum | 31.422292 | 16.369762 | 8.265928 | 4.18065 | 3.615040 |
| Mean | 31.4648002 | 16.3793706 | 8.270572 | 4.20602 | 3.823758 |
| Median | 31.43158 | 16.374973 | 8.267531 | 4.20696 | 3.683804 |
| Speedup | 1.000000 | 1.921002 | 3.804429 | 7.480888 | 8.228764 |
| Efficiency | 1.000000 | 0.960501 | 0.951107 | 0.935111 | 0.514298 |

Cuda_loopbreak implementation - seconds per render:

| | Serial | CUDA_loopbreak | | | |
|---|---|---|---|---|---|
| Threads (per block) | 1 | 16 | 16 | 512 | 512 |
| Blocks | 1 | 16 | 512 | 16 | 512 |
| Run 1 | 31.422292 | 3.500281 | 1.223597 | 1.035444 | 0.639993 |
| Run 2 | 31.43158 | 3.430984 | 1.179361 | 0.978722 | 0.602101 |
| Run 3 | 31.42617 | 3.430325 | 1.179367 | 0.977414 | 0.602683 |
| Run 4 | 31.448678 | 3.429313 | 1.180258 | 0.977295 | 0.602095 |
| Run 5 | 31.59528 | 3.430987 | 1.180519 | 0.977660 | 0.601976 |
| Minimum | 31.422292 | 3.429313 | 1.179361 | 0.977295 | 0.601976 |
| Mean | 31.4648002 | 3.444378 | 1.1886204 | 0.989307 | 0.6097696 |
| Median | 31.43158 | 3.430984 | 1.180258 | 0.977660 | 0.602101 |
| Speedup | 1.000000 | 9.135118 | 26.471698 | 31.804890 | 51.601130 |
| Efficiency | 1.000000 | 0.035684 | 0.003231 | 0.003882 | 0.000197 |

load balanced implementation - seconds per render:

| | Serial | load_balance (CUDA+CPU) | | | |
|---|---|---|---|---|---|
| Threads (per block) | 1 | 16 | 16 | 512 | 512 |
| Blocks | 1 | 16 | 512 | 16 | 512 |
| Run 1 | 31.422292 | 2.355774 | 0.853927 | 0.698612 | 0.464359 |
| Run 2 | 31.43158 | 2.291376 | 0.786990 | 0.657858 | 0.429514 |
| Run 3 | 31.42617 | 2.291111 | 0.787582 | 0.657679 | 0.431332 |
| Run 4 | 31.448678 | 2.291106 | 0.787354 | 0.657555 | 0.434087 |
| Run 5 | 31.59528 | 2.292387 | 0.786935 | 0.656439 | 0.430307 |
| Minimum | 31.422292 | 2.291106 | 0.786935 | 0.656439 | 0.429514 |
| Mean | 31.4648002 | 2.3043508 | 0.8005576 | 0.6656286 | 0.4379198 |
| Median | 31.43158 | 2.291376 | 0.787354 | 0.657679 | 0.431332 |
| Speedup | 1.000000 | 13.654518 | 39.303606 | 47.270806 | 71.850600 |
| Efficiency | 1.000000 | 0.053338 | 0.004798 | 0.005770 | 0.000274 |

# 4 Conclusion

In conclusion, this paper has done an in-depth analysis of the various optimizations of the Mandelbrot set problem, with an emphasis on parallelization. We were able to achieve significant speedups through a culmination of leveraging CPU and GPU cores, caching, math optimization and distribution, and ultimately produced near real-time generations of Mandelbrot set visualizations.

While we have explored various methods, there is still potential for further research and testing to be done in order to achieve even greater performance gains. Additionally, our implementation still lacks components such as mouse interaction, which could enhance its interactivity.

# Citations


Golla, Anudeep, and Paul Strode. \Discovery of the Heart in Mathematics: Modeling the Chaotic Behaviors of Quantized Periods in the Mandelbrot Set:
Journal of Emerging Investigators." Discovery of the Heart in Mathematics:
Modeling the Chaotic Behaviors of Quantized Periods in the Mandelbrot Set |
Journal of Emerging Investigators, 14 Dec. 2020, emerginginvestigators.org/
articles/discovery-of-the-heart-in-mathematics-modeling-the-chaotic-behaviors
-of-quantized-periods-in-the-mandelbrot-set.

Kraus, Jiri. \An Introduction to Cuda-Aware MPI." NVIDIA Technical Blog, 21 Aug. 2022, developer.nvidia.com/blog/introduction-cuda-aware-mpi/.

Latt, Jonas, et al. \Multi-Gpu Programming with Standard Parallel C++, Part 2." NVIDIA Technical Blog, 26 May 2022, developer.nvidia.com/blog/
multi-gpu-programming-with-standard-parallel-c-part-2/.
\Running Cuda-Aware Open MPI." FAQ: Running Cuda-Aware Open MPI, www.open-
mpi.org/faq/?category=runcuda. Accessed 13 June 2023.

"Mandelbrot Test Code." Mandelbrot Test Code - Optimizing CUDA for GPU Architecture,
selkie.macalester.edu/csinparallel/modules/CUDAArchitecture/
build/html/1-Mandelbrot/Mandelbrot.html. Accessed 14 June 2023.

Pacheco, Peter S., and Matthew Malensek. An Introduction to Parallel Programming / Peter S. Pacheco, Matthew Malensek. Morgan Kaufmann Publishers, an Imprint of Elsevier, 2022.

Pacheco, Peter, and Matthew Malensek. \An Introduction to Parallel Programming, 2nd Ed." An Introduction to Parallel Programming, 2nd Edition, 7 Mar. 2023, www.cs.usfca.edu/~peter/ipp2/index.html.

Yi, X, et al. \Cudamicrobench: Microbenchmarks to Assist Cuda Performance Programming." CUDAMicroBench: Microbenchmarks to Assist CUDA Performance
Programming (Conference) | OSTI.GOV, 27 Oct. 2021, www.osti.gov/servlets/purl/
1828124.

# Appendix

## A Source Code

Source code is available at the git repository, at `https://github.com/Patronics/Parallel-Mandelbrot`. The main files of interest are `fractal.c` (serial and OpenMP implementation), and the 3 final cuda versions at `fractal_cache.cu`, `fractal_loadbalance.cu`, and `fractal_loopbreak.cu`.

### A.1 Cuda implementations summary

The cuda cache implementation in the repository includes a cache struct that stores the colors accociated with certain indeces in the window. It works by storing the initial screen of pixels in an array of color structs, where `index = my_i +width * my_j`. When pixels that have already been calculated from the first screen are included in the transformed image, these pixels are retrieved from the cache rather than being recalculated in the `compute_point()` function.

Load balance attempts to balance the workload between the CPU and GPU, as kernel calls are asynchronous.

Loop break attempts to shorten the time needed for calculation by automatically returning the maximum iterations if a point is within a certain range. This avoids needless iterations through a for loop, and can save time for large grid sizes calculating points near the origin.

## B Contributions

### B.1 Patrick Leiser

Made and refined the initial OpenMP implementation (that the CUDA implementations were derived from), added interactivity and color, made the benchmark script, and extensive debugging and testing of many versions of the code. Also produced the timings tables.

### B.2 Jared Pugh

Tested numerous strategies for improvement for OpenMP and CUDA, ran various benchmarks to evaluate improvements, heavy involvement in debugging implementations, investigated CPU/GPU load balancing.

### B.3 Catherine Yaroslavtseva

Implemented cache, implemented cardioid/bulb math and coordinate mirroring, tested multiple kernel implementations, contributed to the report and initial

CUDA implementation, and tested block sizes.