CS470 Final Project

Max Mancini, Seth Roper, Dylan Becker

Spring 2023

1 Team

Max Mancini, Seth Roper, Dylan Becker

2 Opening Notes

- Our repository can be found at https://github.com/Metalix53/cs470_final.
- Credit for the starting serial implementation goes to GitHub user maxmmyron. His repository can be found at https://github.com/maxmmyron/mandelbrot-visualizer. We decided to start from this code to cut down on the time of setting up our rendering window and to get a baseline for the Mandelbrot function. Since the Mandelbrot set is a widely studied subject with thousands of implementations all over the internet, we felt this was a good starting point.
- The actual calculations that are being performed to create fractal images are done with complex (imaginary numbers) which is a difficult idea to understand. We've learned enough about the Mandelbrot and Julia set to understand how they work in code and what the code is doing when it runs, however understanding how the calculations we're making form fractals is another story. For this reason, we decided not to include any detailed explanation of how performing these calculations on complex number produce the images that they do. The Mandelbrot set Wikipedia page does a pretty good job of explaining the fine details if you are interested.
- The windows version directory in our GitHub repository was used to test the program on a more powerful system however the main version is meant to be run on Linux.
- Some notes about the program:
- Decreasing the iteration count below 10 will cause the program to close, this is not a bug. Using an iteration count that low doesn't produce any meaningful visuals and inputting negative iteration counts causes the program to freeze.
- All information produced by the program will be output to the terminal, this includes the precision count as well as the runtime of each operation.
- To get the program set up to run, clone the repository and run make to build both versions. To run them use the provided run.sh script to run the CUDA version and run_serial.sh to run the serial version. Using the scripts is necessary since we need to export the graphics library or the program won't run. The scripts combine the export and the executable.

3 Functionality

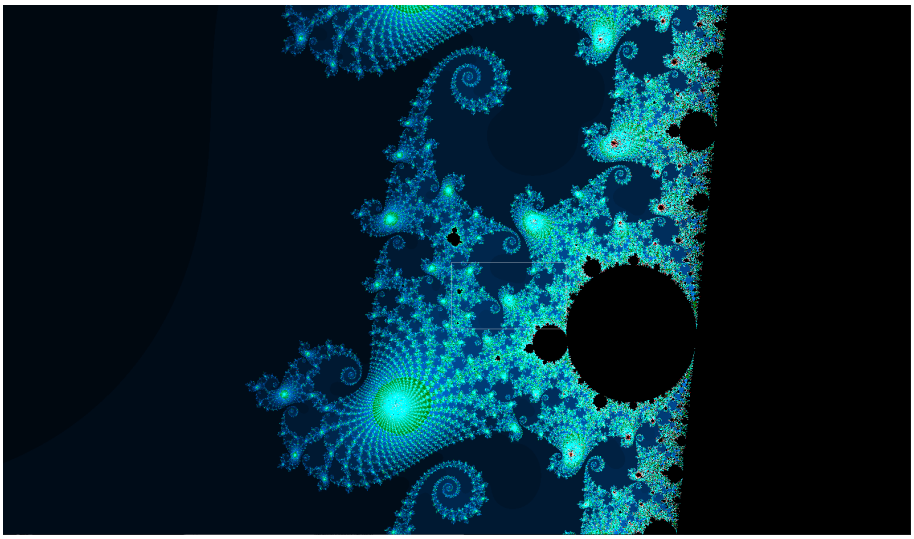
Our project consists of rendering two fractals that look distinct but are related mathematically. These two fractals are the Mandelbrot set and the Julia set. The Julia set is a subset of the Mandelbrot set. When running our application the user can switch between viewing the Julia set or the Mandelbrot by pressing the

"j" key. Both sets have some unique functionality specific to their set, but share some common functionality. Each functionality unique to a specific set is experimental and can make run-times more volatile. They are included because we felt they really exasperate the benefits of the CUDA implementation compared to the serial implementation due to the speed up allowing for more interesting image renderings. On both fractals the user can increase the precision of the fractal being rendered by using their scroll wheel. Scrolling up will increase precision, while scrolling down will decrease precision. Users can also cycle through different color transformations by pressing the "t" key. Every press of the "t" key swaps the red value of the pixels with original blue value of the pixel, the blue with the green, and the green with the red. These transformations will persist when increasing iteration count or after utilizing each fractal's unique functionality. The image can be reset to the default by pressing "O".

KEYBINDS		
	Increase Iterations	Mouse Wheel Up
	Decrease Iterations	Mouse Wheel Down
	Switch Modes (Julia or Mandelbrot)	J
	Color Transformation	T
	Zoom (Mandelbrot)	Left Click
	Shift Julia Set	W,A,S,D
	Reset View	O

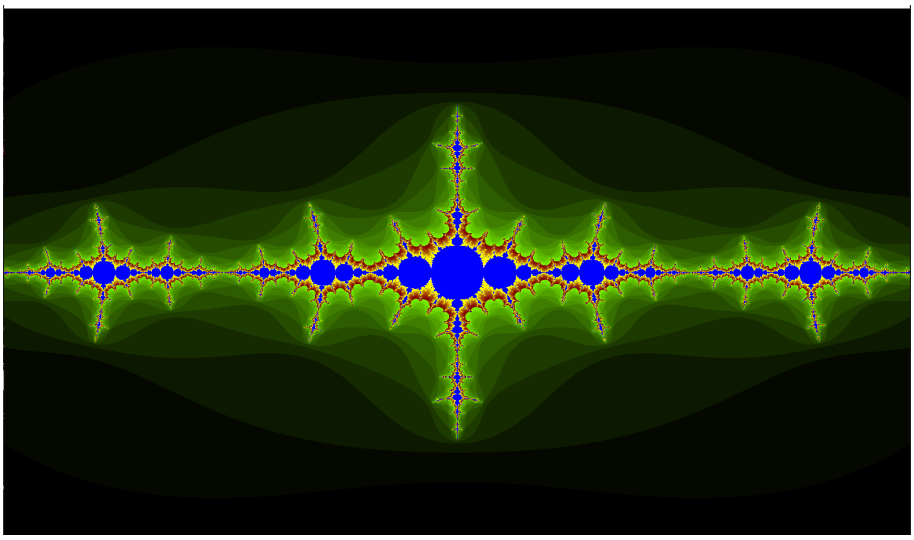
3.1 Mandelbrot

While in Mandelbrot mode, the user will see a rectangular box that follows their cursor. The user can input their mouse one button in order to zoom into a subsection of the fractal that their mouse is hovering over. Zooming into a subsection of the fractal will take the subsection covered in the rectangular and box and transform the window to render this subsection. As the precision value increases (IE, the more the user scrolls up) the more zooms the user can input and see meaningful renderings. Here is an interesting fractal we were able to produce with the Mandelbrot set:



3.2 Julia

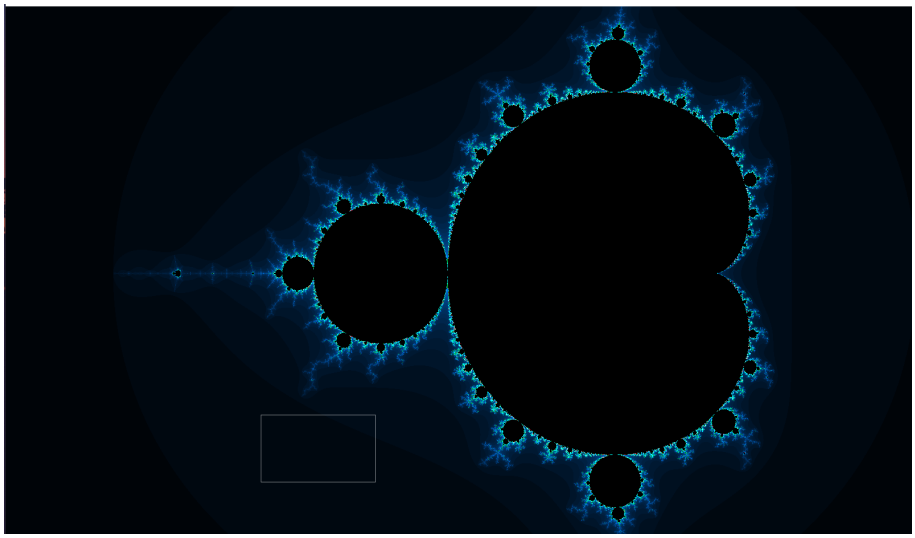
In Julia mode, the user will be presented with a Julia set fractal. To manipulate this image, use the W, A, S, and D keys on the keyboard. Performing these actions will shift the starting values for rendering the image slightly, which will shift the image slightly. By default, Julia mode renders an image that is in the middle of the set. This means that shifting in any direction will bring the image closer to the edges of the set, typically producing a slightly less interesting result towards the far edges. Experimenting with shifting the values of the Julia set can produce some extremely interesting experimental results and it's best to test it out for yourself to see the results. Here is an interesting fractal we were able to produce with the Julia set:



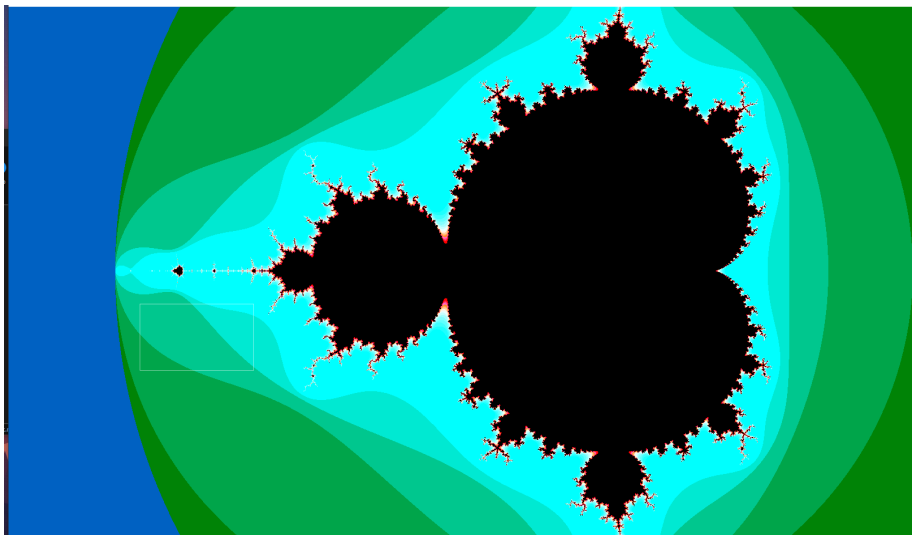
3.3 Testing Instructions

In order to run our implementation and verify speed up results, we recommend the following steps:

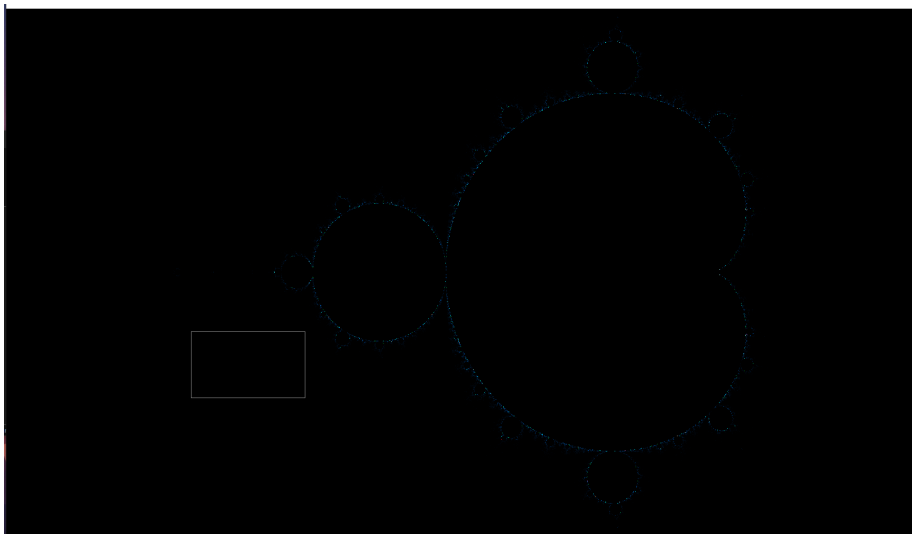
- Start the program by running the shell script (run.sh for the CUDA version and run_serial.sh for the serial version).
- When the program starts, it will be in Mandelbrot mode by default and set to 512 iterations/precision. It should look like this:



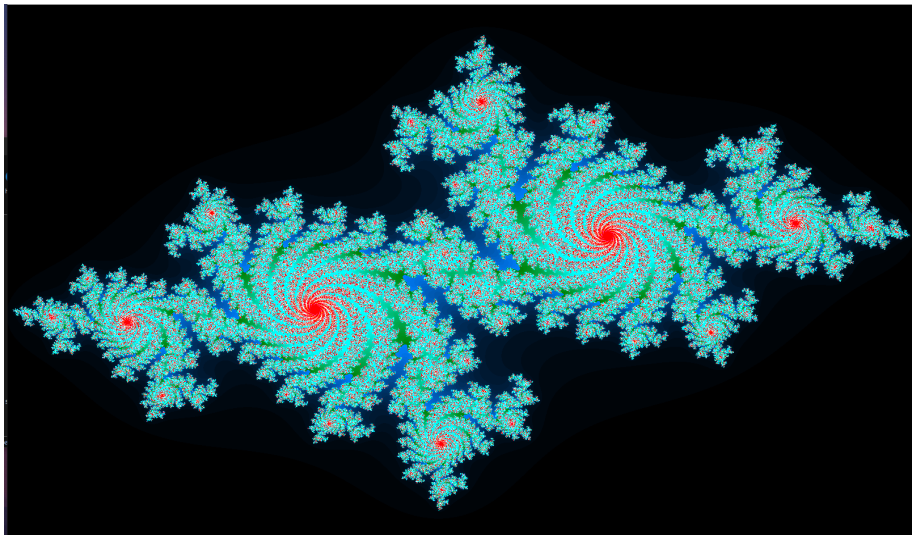
- To follow our testing, begin scrolling the mouse wheel down one scroll at a time. You will see the iteration count and timing for each rendering being output to the terminal.
- Scroll down until the precision count is at 32 (PREC: 32). This was the lowest precision count that we included in the runtime results. It should look like this:



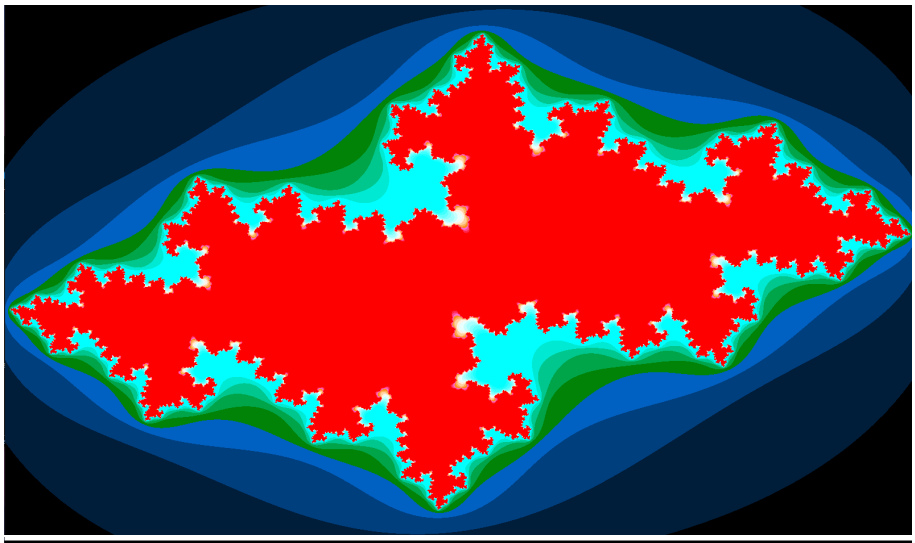
- Now, begin scrolling up until the precision count reaches 16384 (PREC: 16384). This may take a while if you are running the serial version. It should look like this:

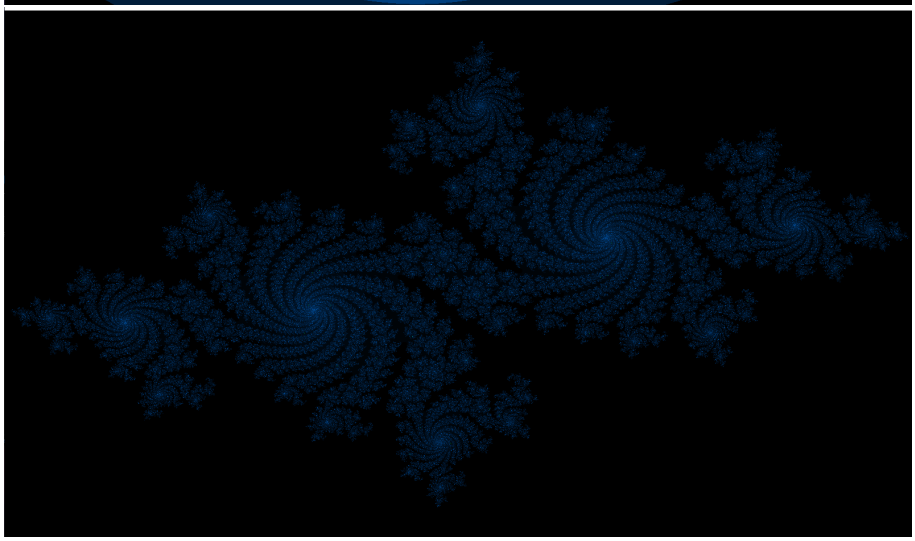


- After reaching 16384 precision, scroll back down to the starting point at 512 iterations.
- At this point, the Mandelbrot testing is concluded. Now, switch over to Julia mode by pressing "J" on the keyboard.
- The Julia set will be rendered with the same iteration count as the Mandelbrot, 512 by default.

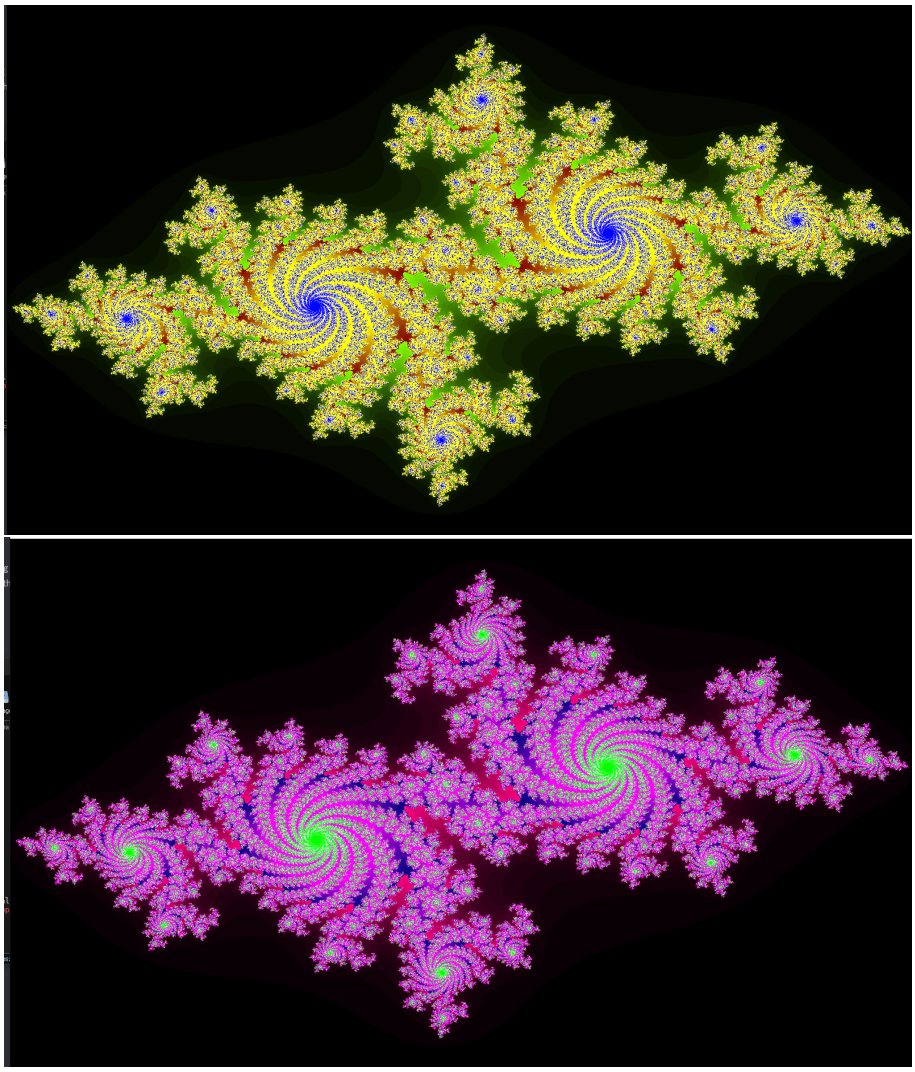


- Repeat the scrolling steps from the Mandelbrot testing on the Julia set, the results will again be output to the console. The 32 and 16834 iteration images should look something like these:





- To test the color transformation function, return the Julia set rendering to 512 iterations and press "T" on the keyboard.
- You will see the color of the image begin to change between 3 different colors and the timing will be output to the console. The transformations should look like this (Transformations run in the same time on both the Julia and Mandelbrot so either can be used to test them):



4 Kernels

Our code utilizes three kernel functions in order to utilize the GPU to perform calculations. While all of the kernel functions involve calculating pixels, the calculations are all distinct. Each kernel assigns each CUDA thread a stride. This stride represents the range of pixels that each thread is responsible for calculating. Each kernel is called in the driver code under a corresponding wrapper function in order to add consistency between the parallel and serial implementations. One important note is the block size and thread count values that we used on our kernels. We found that the best results were found by using 512 for both values. These values are the same for all kernels.

4.1 Mandel kernel

The Mandel Kernel is responsible for calculating the pixels to render the Mandelbrot set. Each CUDA thread in the kernel is responsible for calculating their portion of the pixels in the set as defined by the stride. When rendering the Mandelbrot set we must consider the precision in which we are rendering. We utilize a device function called mandelIter that the Mandel kernel utilizes in order help generate the proper pixel values based on the depth. The value returned from the mandelIter function is what determines the color of the pixel.

4.2 Julia kernel

The Julia Kernel is responsible for calculating the pixels to render the Julia set. This kernel works very similarly to the Mandelbrot kernel, using a grid stride loop to divide the pixels on the screen between the CUDA threads. Since the Julia set is essentially a subset of the Mandelbrot set, the calculations required to render it are less complex and are all done within the kernel. Still, each iteration of the grid stride loop runs its own loop on the number of iterations to determine the color of the pixel, similar to the Mandelbrot.

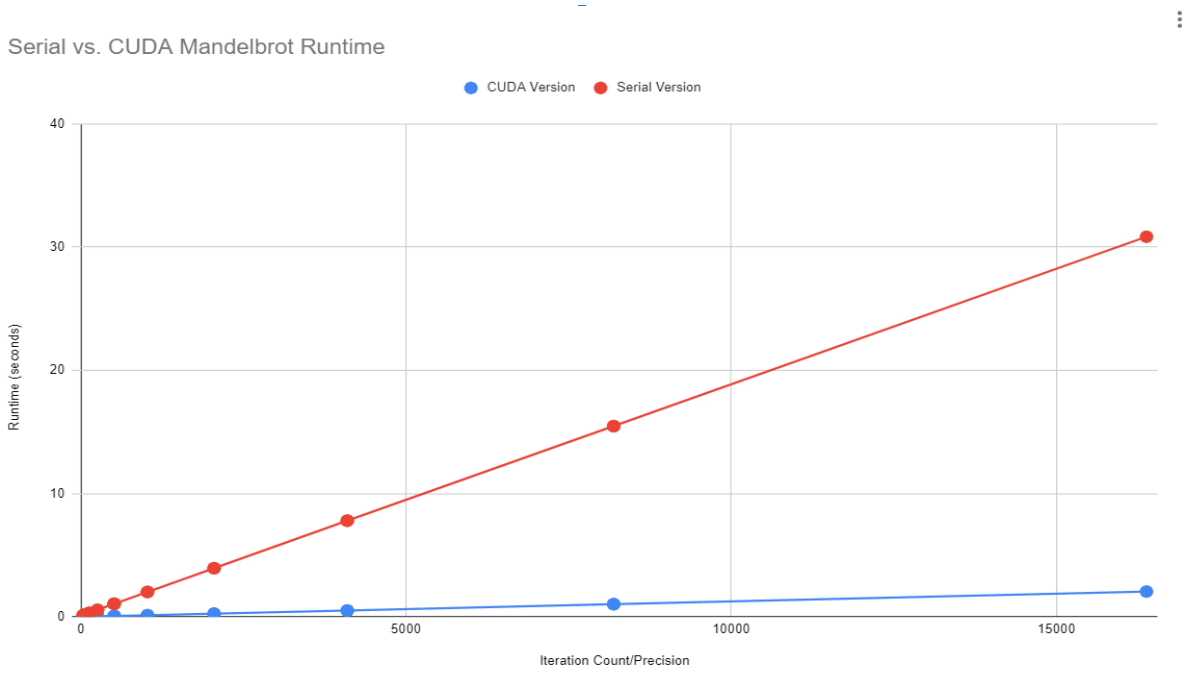
4.3 Transform kernel

The transform kernel is the simplest of the three kernels. There is a global array of pixels living on the GPU's heap (IE in vram) that is updated upon the completion of any of the three kernels. This kernel will take that array of pixels and swap pixel's red value with the pixel's green value, the green value with the blue, and the blue with the red. This kernel affects this global array directly. In effect, this kernel cycles through swapping RGB values of all of the pixels calculated from other kernels.

5 Performance Analysis

5.1 Mandelbrot

To test the runtime of the Mandelbrot function in the CUDA version versus the Serial version, we used the method described in the testing instructions section above. Beginning at 512 iterations, scrolling down to 32, up to 16834, and back to 512. For consistency of results, we did not include the use of the zoom function since attempting to render the same exact scene on both versions of the program using the zoom function would likely not be possible. The Mandelbrot function performed very well and provided the best speedup results out of the three kernels.



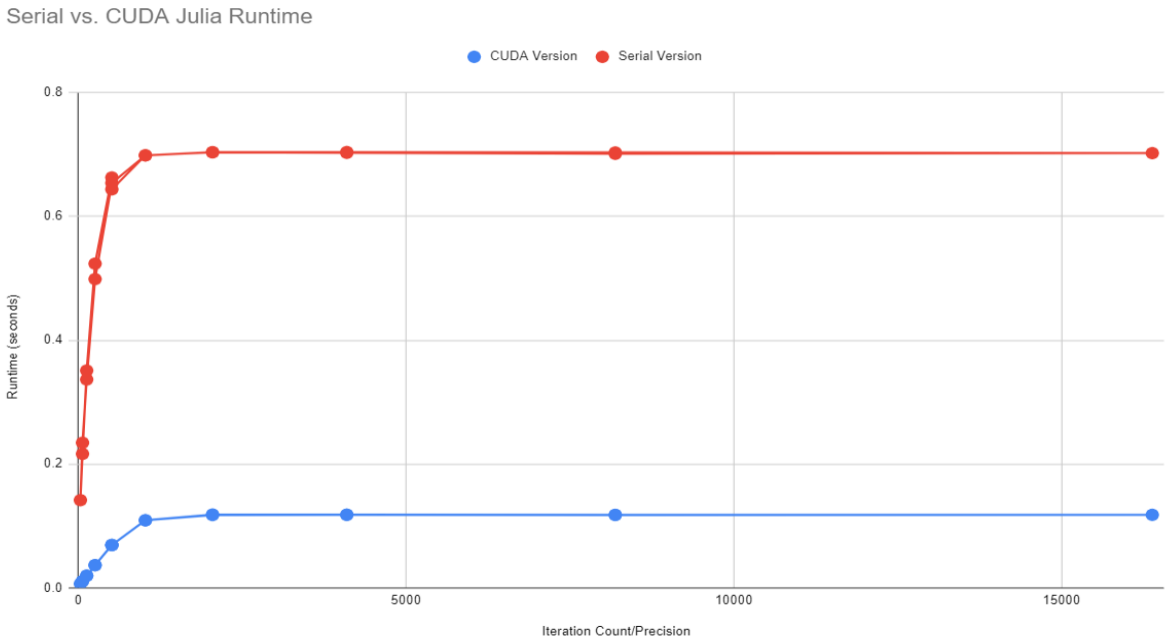
Iteration count	Cuda Mandel Runtime	Serial Mandel Runtime
512	0.0851s	1.0802s
256	0.0456s	0.5711s
128	0.0258s	0.3261s
64	0.0134s	0.2042s
32	0.0089s	0.1377s
64	0.0142s	0.2033s
128	0.0221s	0.3266s
256	0.0392s	0.5695s
512	0.0731s	1.058s
1024	0.1332s	2.0172s
2048	0.2606s	3.9483s
4096	0.5152s	7.8062s
8192	1.0276s	15.4864s
16384	2.0533s	30.8498s
8192	1.0349s	15.4693s
4096	0.5139s	7.8078s
2048	0.2607s	3.9461s
1024	0.1339s	2.025s
512	0.0696s	1.0526s

These results show a graph of the serial version runtime plotted (in red) against the CUDA version runtime (in blue), along with the table of data that was used to create the graph. On average, the Mandelbrot kernel produces between a 10 and 15x speedup (and sometimes higher) with larger speedup values being more noticeable at higher iteration counts. Overall, these results are pretty stellar, and they allow for a much smoother experience of zooming through the image and finding interesting fractals. One slightly interesting thing about these results is when we scroll the iteration count back down to 512 from 16384, the runtime seems to have decreased from the initial rendering at 512 iterations. We speculate that this has something to do with the GPU caching some sort of information on it that is useful in the calculations however we

can't confirm this. It seems like the program has a sort of "warm-up" period and after scrolling through some different iteration counts, it gets faster. Other than this, the runtime results for the Mandelbrot code are relatively uninteresting. It seems to be a rather consistent 15x speedup through all iteration counts and there is little variance even when applying the color transformation or zooming.

5.2 Julia

To test the runtime on the Julia function in the CUDA version versus the Serial version, again, we used the same method described in the above instructions. Beginning at 512 iterations, scrolling down to 32, up to 16384, and back down to 512. We did not apply any transformations to the Julia set and left the image as the default for consistency between both versions, although it would likely not make much of a difference in terms of scaling. The Julia set results were less impressive than the Mandelbrot results however they were still acceptable and allowed for a much smoother experience.

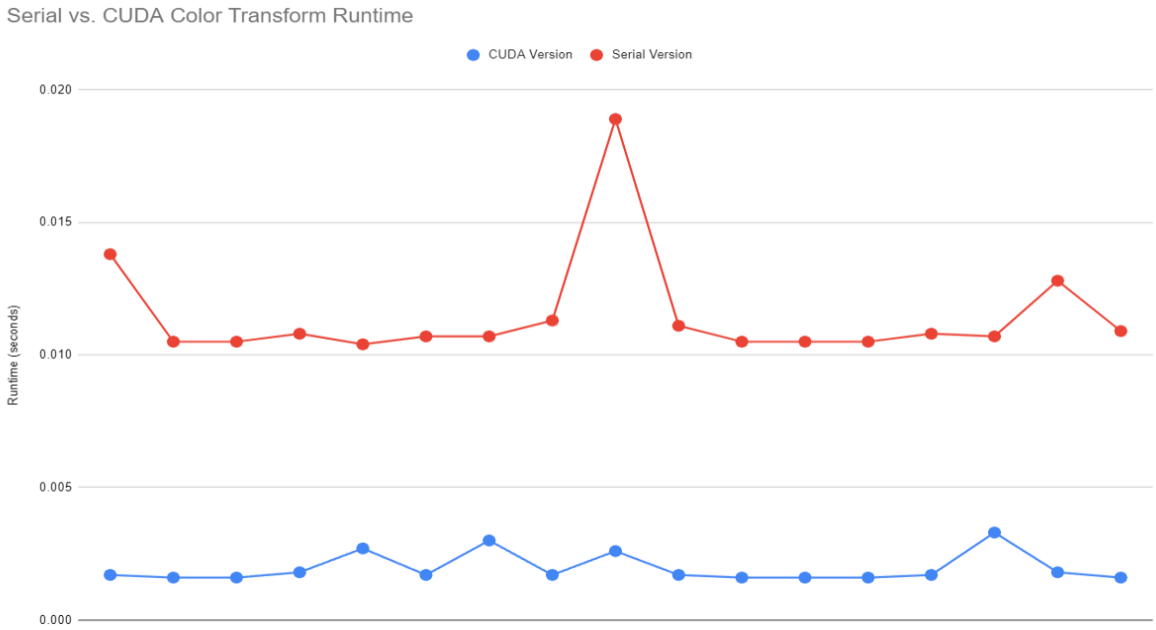


Iteration count	Cuda Julia Runtime	Serial Julia Runtime
512	0.0692	0.663
256	0.0364	0.5237
128	0.0196	0.351
64	0.0107	0.2163
32	0.0064	0.1415
64	0.0107	0.2343
128	0.0194	0.3365
256	0.0365	0.4988
512	0.0688	0.654
1024	0.1089	0.6987
2048	0.1175	0.7039
4096	0.1177	0.7041
8192	0.1176	0.7034
16384	0.1178	0.7023
8192	0.1177	0.7012
4096	0.1181	0.7028
2048	0.1183	0.7036
1024	0.1091	0.6988
512	0.069	0.6438

These results follow the same pattern as the Mandelbrot results, with the CUDA version plotted in blue and the Serial version plotted in red on the graph as well as the accompanying table of data. The results for the Julia set are much more interesting than the Mandelbrot set and there's a little bit more to analyze here. Firstly, the speedup results start initially at around 10x, increasing as we lower the iteration count, and decreasing slightly as we increase the iteration count. Perhaps the most obviously strange thing about this data is that the runtime seems to cap out at around 2048 iterations. Since the Julia set is a subset of the Mandelbrot set, increasing the iteration count past this point can no longer produce any higher levels of detail since there is no depth to the Julia set like there is in the Mandelbrot set. This is why you can zoom through the Mandelbrot set and find some fractals that look extremely similar to the Julia set. Rendering the Julia set is essentially extracting a specific formation in the Mandelbrot set and rendering it on its own. Visually, when we continue to increase the iteration count on the Julia code, the image continues to get more defined until around 2048 iterations after which it begins to become washed out. At the maximum run-time on the default Julia image, the speedup is right around 7x, however it's likely that different images could produce different results.

5.3 Color Transformation

To test the color transformation function, we simply start the program with the default images (on either the Julia or Mandelbrot set) and press "T". Since this function only relies on a copy of the pixels array, there are no other values to change. The results for this function were relatively straight forward and the function doesn't take all that long to finish in the serial version anyways however we were still happy with the results.



Cuda Transform Runtime	Serial Transform Runtime
0.0017	0.0138
0.0016	0.0105
0.0016	0.0105
0.0018	0.0108
0.0027	0.0104
0.0017	0.0107
0.003	0.0107
0.0017	0.0113
0.0026	0.0189
0.0017	0.0111
0.0016	0.0105
0.0016	0.0105
0.0016	0.0105
0.0017	0.0108
0.0033	0.0107
0.0018	0.0128
0.0016	0.0109

The graph shows the CUDA version plotted in blue and the serial version in red and the data points are the runtime over 17 runs of the function. The speedup lands at around 7x on average which isn't mind blowing however this is likely due to the complexity of the function. The function simply loops through all of the pixels and shifts the values by 1 position which isn't a very intense operation. Still the CUDA version is able to make this function take nearly no time at all to run. There are a few anomalies in the data that we wanted to include since they seemed to be occurring pretty consistently in both the CUDA and serial version. Occasionally, the runtime seems to spike to nearly double every few runs of the function. Since this is consistent between the serial and CUDA version, we aren't entirely sure what could be causing it. The

occurrence of this spike seems completely random, sometimes occurring twice in a row and sometimes not at all in 10 runs or so. Since this operation takes so little time to complete, we speculate that this could be caused by a hardware related issue that causes a slight variance in run-time that is not noticeable on more complex functions.

5.4 Windows version

The Windows version of the code was created to be able to test the capabilities of CUDA with better GPU's available to the team. Small changes had to be made in order for it to be compatible with windows like the timing code. Downloading the windows_package.zip in the windows version folder and running the executable should work on Windows Intel x86 architecture. The performance metrics included are running on a NVIDIA 4070ti. The last iteration count on the graph is the highest possible without overflowing the iteration count floating point variable, indicating strong scaling. The last two values for the Mandel set were omitted due to the speed limitations of the serial implementation.

Iteration Count	Cuda Julia Runtime	Cuda Mandel Runtime
64	0.0015s	0.0017s
1024	0.0151s	0.0328s
16384	0.0158s	0.2646s
262144	0.0159s	4.1476s
4194304	0.0160s	72.2739s
67108864	0.0161s	N/A
1073741824	0.0176s	N/A

6 Future Project Extensions

In terms of future features to add to the application, the biggest would be resolution modifier. This would allow the user to change how many pixels are actually being rendered onto the screen, allowing for more intensive tests with higher resolutions. This could help for testing higher resolutions of the Julia set specifically, which sees only small run time increases across iterations. Another addition is a larger floating variable to store the iteration counts to allow for greater amounts of precision before overflow occurs. These would be the biggest two in term of testing further capabilities of Serial vs. CUDA run time. We also could add renderings of other fractals to the application and add more features for each fractal such as a zooming functionality for the Julia set. A final potentially interesting extension to this project would be creating a new version of the program that utilizes CPU parallelism to speed up the program on systems that cannot take advantage of CUDA (i.e they don't have a Nvidia GPU). This could be done using OpenMP to create parallel loops for looping over all pixels. We could also experiment with a distributed version of this project that uses a technology such as MPI to utilize multiple nodes (such as on the cluster). It would be extremely interesting to see what kinds of results we would be able to produce with a network of powerful machines.

7 Retrospective and Conclusion

In summation, the CUDA implementation performs significantly than the serial implementation. These types of pixel calculations are what GPUs were designed for the and the performance increases we documented serve as an exemplum to this. Although we would have liked to have more significant speedup results for each kernel, we are still happy with the results. All of the kernels and serial implementation functionality was

developed by us except for the serial implementation of the Mandelbrot function and SFML code for the rendering window, which can credited to this repository https://github.com/maxmmyron. In conjunction with linking the SFML library locally, refactoring this code to utilize CUDA was the most time consuming task of this project. Overall we found CUDA to be very intuitive and are satisfied with our final implementation.