Parallel L-System Generation and Rendering

Nolan Mass
Joshua Mathews

May 2022

1 Summary

We implemented a program that uses L-Systems to generate strings, then uses those strings as instructions to draw fractal trees. We ran our code on the GHC machines using OpenMP to generate the instructions in parallel and CUDA to render the images in parallel.

2 Background

2.1 L-System

A Lindenmayer system, or L-system is a grammar which generates instructions to draw fractal objects. An L-system starts with an initial string (called an axiom) and repeatedly applies rules to each of the characters in the string to map the string into a new, more complex string.

For instance, a simple L-system could start with the axiom A and the rules $A \to AB$ and $B \to A$. After one application of the rules, the initial string A is mapped to AB. Applying the rules a second time, we map the first character A to AB and the second character B to A, giving the string ABA. Applying a third iteration, we get $ABAAB$ and so on.

For our project, we chose to focus on a particular L-system which generates instructions to draw a fractal plant, although our code is adaptable to any other L-system. This L-system starts with the axiom X and consists of the two rules:

$$\begin{aligned} &X \to F + [[X] - X] - F[-FX] + X \\ &F \to FF \end{aligned}$$

After applying the rules once, we get the string:

$$F + [[X] - X] - F[-FX] + X$$

We can now interpret these instructions to draw a fractal plant. Each character corresponds to a different instruction.

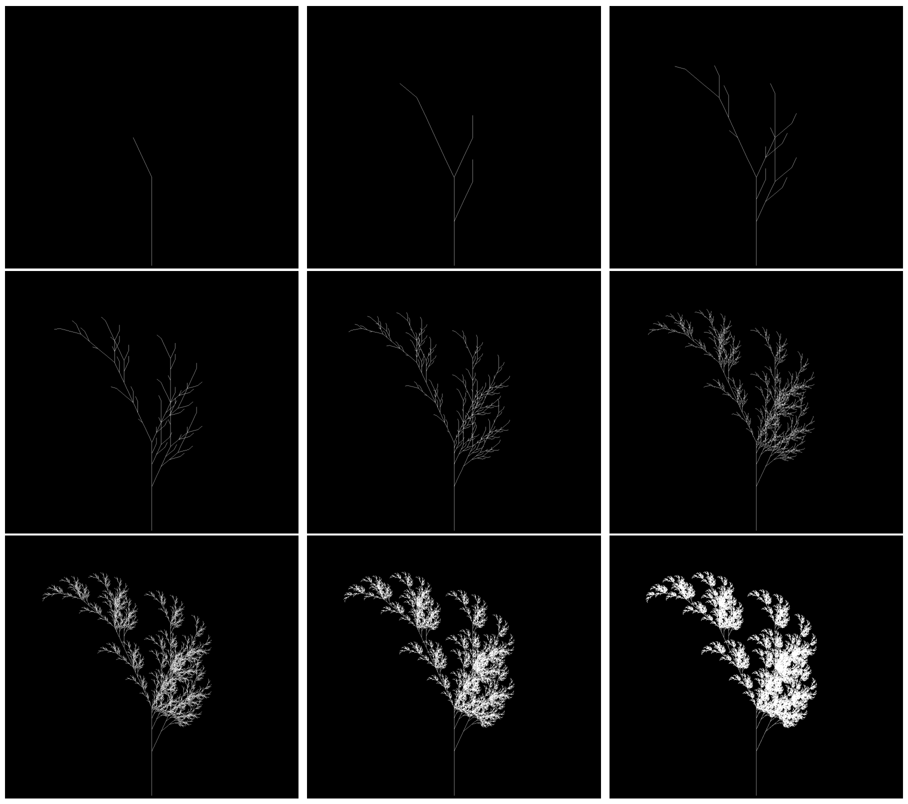


Figure 1: Tree fractal with depth 1-9

character	instruction
F	draw a straight line
+	rotate clockwise
-	rotate counterclockwise
[	store current position and angle
]	load previous position and angle

We draw a tree by reading through the entire string of instructions sequentially. We start out with a position and angle. Each time we draw a line, we draw from the current position along the current angle and update the current position to the end of the line. The line length and rotation angles are generally fixed. However, in our implementation, we randomly adjust each of the angles so that every tree we draw looks different.

2.2 Implementation Details

For our project, we draw a forest made up of many fractal generated trees using the L-system specified above. We specify the number of trees, the tree depth,

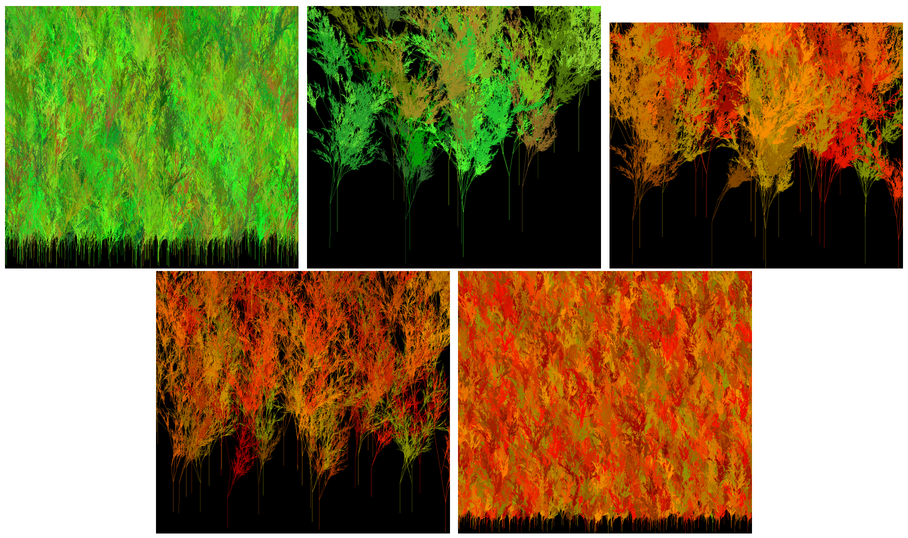


Figure 2: Sample outputs with varying tree number, depth and length

and the tree size (specified by the length of a line). There are three steps to the algorithm. First, we generate the instructions. This consists of starting with the axiom and repeatedly applying the rules depth times. Second, we read through and interpret the instructions. Finally, we render the lines generated by the instructions.

To represent an L-system, we specify a string representing the starting axiom and an unordered_map to represent the rules. Then we generate the rules and store them as a string. This information, along with information describing the position, depth, color, and other information important to each instance of an L-system is stored in an L-system class. In our initial implementation, we render the lines as we read through the instructions sequentially. In later implementations using CUDA, we first read through the instructions and store all of the endpoints of the lines in an array, then we render the lines in parallel. The four floats to specify each line are stored contiguously for better locality.

2.3 Dependencies

In the first step, L-systems are a parallel rewriting system so it is possible to rewrite each character in parallel. However, rewriting a character requires knowing the correct position in the new string, which depends on each of the previous characters in the string. Luckily, it's still possible to exploit the parallelism here using an exclusive scan to find the correct positions at each index but this involves a lot of additional overhead work.

We have to read through the instructions for a tree sequentially since the current position and angle at each point in the instructions depend on the positions and angles earlier in the instructions. There is some opportunity for parallelism across branches of the tree but we chose not to exploit it in the project because the task granularity of rendering only part of a tree is relatively small.

There is no dependency between trees other than that they are rendered onto the same scene and may overlap. Processing the trees is data-parallel since every tree stores its own instructions, lines, and other data. In our OpenMP code, we render each tree on a single thread which should give better locality in the image array. In the CUDA renderer, we group lines from each tree together, which should help with locality among the shared memory in each block.

3 Approach

3.1 Algorithm

We used both OpenMP and CUDA running on the GHC machines for our project. As mentioned before, the program has three parts: generating the instructions, following the instructions, and rendering the output.

3.1.1 Generating and Following Instructions

We parallelize instruction generation across trees but within each tree, the algorithm is sequential. We start out with the axiom and apply the rules to it depth times to get instructions to render the fractal. After completing this step, each tree has its own string of instructions.

Next, we follow the instructions in order to find the coordinates of each line to draw. We also parallelize this operation across trees but it's completely sequential within each tree. First, we allocate an array with 4 floats for each line in the tree. The number of lines in a tree is the number of F's that appear in the instructions so we can get a closed form solution for the number of lines by solving recurrences. We get that the number of lines in a tree of depth d is $3 \cdot 2^{d-1}(2^d - 1)$. Next, we traverse through the instructions and keep track of the x and y positions and angle throughout. For each line, we store the start x and y and end x and y in the array of lines. We use a stack to load and store the positions and angles whenever we encounter the [or] instructions.

3.1.2 CUDA Renderer

Once we have a list of lines for each tree, we concatenate them all into one large array of all of the lines in the image and load it onto the GPU. We also load the corresponding colors onto the GPU. Since we focused on drawing many of the same type and size of L-system (with random variations) with the CUDA renderer, we only stored one color per tree to save space and use indexing to find the corresponding tree/color of each line in order to save space. With all of the necessary data loaded onto the GPU, we run a kernel on each line in the image. The kernel functions take the start and end coordinates and color and run Bresenham's line algorithm to draw the line onto the image.

We kept all of the lines from each tree together in the lines array since they are close to each other in the image. Unfortunately, each tree typically has more that 1024 (the maximum block size) lines but lines that are nearby in the array tend to be in the same block. This helps with locality because all the kernels in one block share memory and nearby lines will write to nearby locations in the image and thus nearby locations in memory.

3.2 Parallelization Process

We benchmarked our code at each step of the process by timing the generation time and the render time on two tests: 64 trees with depth 8 and 4096 trees with depth 5.

Our initial approach was purely sequential. We were initially thinking of parallelizing the instruction generation using exclusive scan but found that converting the sequential algorithm to a parallel algorithm required extra work and space which slowed down the algorithm considerably. Further, after benchmarking the code, we found that more than 95% of the total time was spent on rendering so we decided to focus on parallelizing the rendering process first.

Our first approach was to simply use OpenMP to run the rendering in parallel. Running on 8 cores, this gave us a 1.5x speedup for the depth 5 test and 4.4x speedup for depth 8 test but the rendering was still the bottleneck in our process.

To improve the renderer performance, we decided to switch to a CUDA implementation. We wanted to maximize the amount of parallelization so we parallelized the renderer over all lines rather than just over trees. We chose to parallelize over lines instead of pixels (as we did in the circle rendering assignment) because a lines cover far less pixels than a circles. There is a lot less overlap between lines than there was with circles so each pixel may only have a few lines intersecting it. Intersecting each pixel with a long list of lines would therefore lead a lot of wasted work compared to just rendering the lines directly. Unfortunately, getting all of the lines in a tree requires reading through the tree's instructions sequentially. However, it's still much faster to iterate through the instructions if we don't have to render the lines. We therefore had to move the reading of the L-system instructions from the rendering stage to the generation stage. Instead of rendering each line as we came across it in the instructions, we now store it in the lines array and move on.

Using the CUDA implementation dropped the render time for the depth 5 test from the original implementation's 2279 ms and the OMP implementation's 1456 ms to just 171 ms. We were satisfied with the speedup from the CUDA renderer, especially considering the time taken to set everything up and move all of the data to the GPU. However, the CUDA implementation moved some of the workload from the rendering stage into the generation stage. We noticed that the generation stage now took 450 ms instead of 115 ms from the earlier implementations. This was a 3.7x speedup overall from our initial implementation but the bottleneck shifted to the generation algorithm which was still completely sequential.

Implementation	Generation Time (ms)	Render Time (ms)	Total Speedup
Initial	114.86	2233.63	1
OMP Renderer	115.15	1456.20	1.49
CUDA Renderer	459.13	171.09	3.73
Final	65.48	148.88	10.96

Table 1: Comparison of different implementations on 4096 trees with depth 5

Implementation	Generation Time (ms)	Render Time (ms)	Total Speedup
Initial	106.51	81158.19	1
OMP Renderer	106.93	18410.71	4.39
CUDA Renderer	449.19	125.45	141.42
Final	71.40	106.01	458.08

Table 2: Comparison of different implementations on 64 trees with depth 8

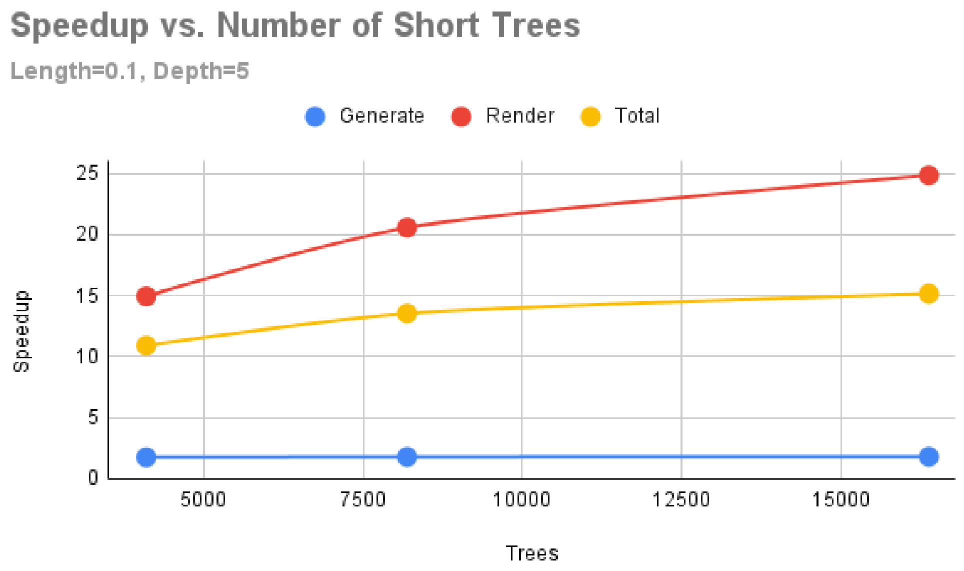
We experimented with performing the generation step in CUDA too, but generating the instructions and finding the lines involved more complex data structures such as hashmaps, strings, and stacks which are difficult to work with in CUDA kernels. The generation step is also much less work than rendering, as we can see when running both sequentially, so we decided OpenMP would be enough to parallelize it. We were expecting to simply add `pragma omp parallel for` to the loop where we generate the instructions and lines but were surprised to find that running the code with OpenMP using 8 cores made the generation step 3 times slower. We eventually discovered that the default C++ random number generator `rand()` is not thread safe and was slowing down the code. We replaced it with a thread safe random number generator and used the thread id as the seed so that each thread generates a different sequence of trees. This brought the generation time down from 450 ms to 65 ms which was nearly a 7x speedup on 8 cores.

4 Results

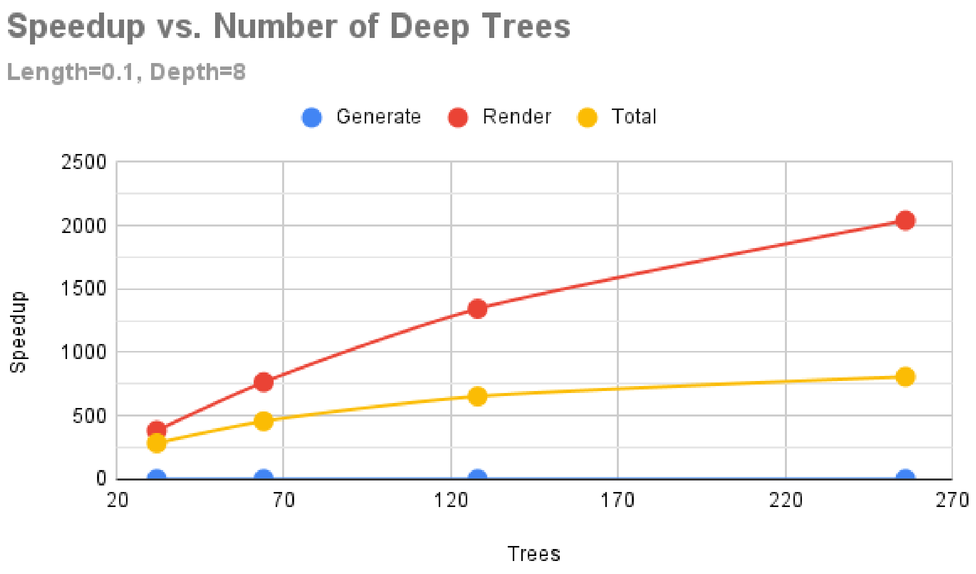
4.1 Experiments

Once we had the final parallel algorithm we ran various tests on the GHC machines to measure performance. Since our focus was mainly on the two parts of generation and rendering, we measured the generation, rendering and total speedup compared to our initial sequential implementation which we started with, which does not use OpenMP or CUDA. For each experiment, we kept all of our input variables constant except one, and measured the speedup. Namely, we varied the number of short trees, number of deep trees, length of the trees, and depth of the trees.

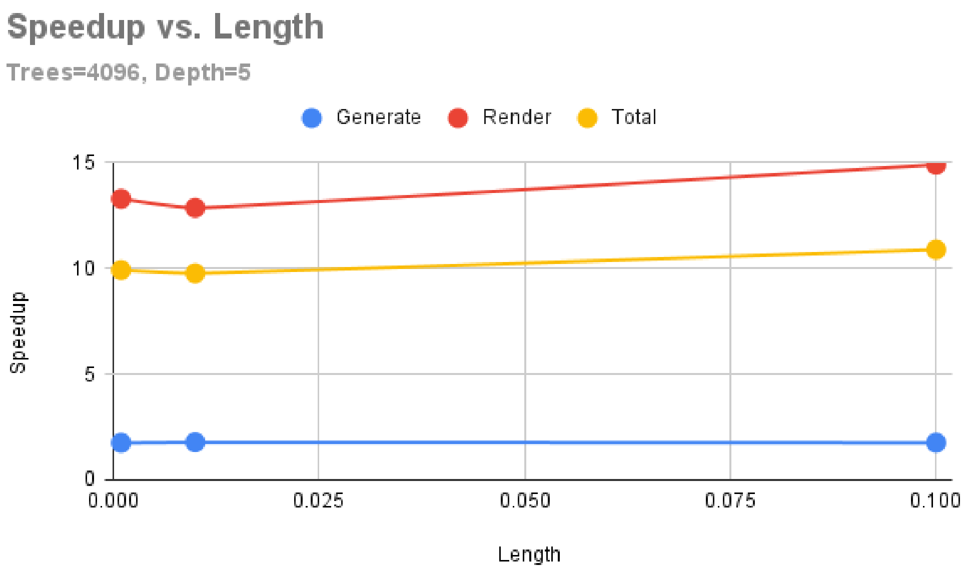
4.2 Speedup Analysis



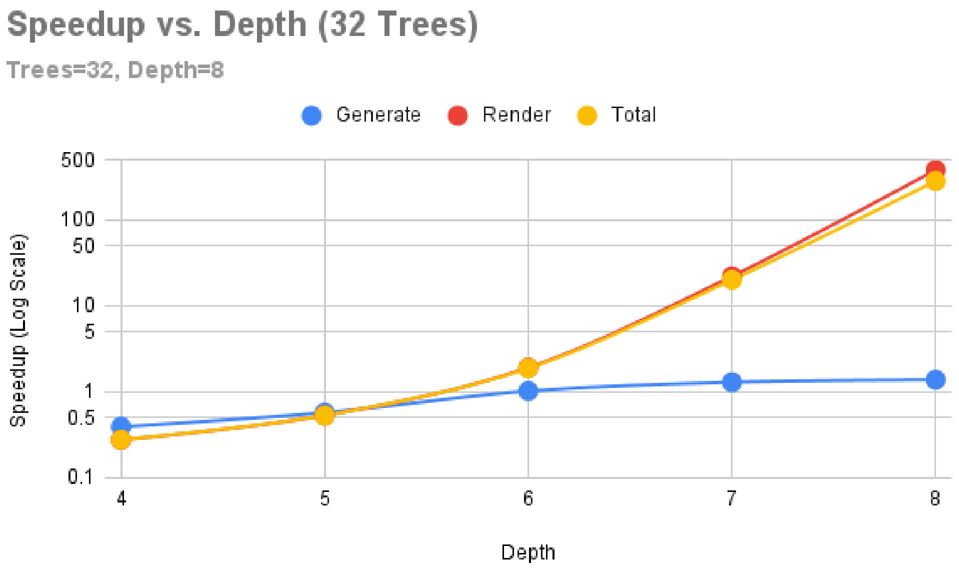
For the first experiment, we fixed the depth to 5 and the length to 0.1 and rendered number of trees = $\{4096, 8192, 16384\}$. We achieved total speedups from 10x to 15x. Since the rendering times for shallow trees were much lower to begin with, compared to the deeper trees, the speedup is lower.



For the second experiment, we fixed the depth to 8 and the length to 0.1 and rendered number of trees $= \{32, 64, 128, 256\}$. We can see that we achieved total speedups from 280x to 808x. This is due to the huge cost of rendering deep trees, which was shortened by the CUDA renderer. Rendering 256 trees of depth 8, took around 340s compared to 421ms using the CUDA renderer.

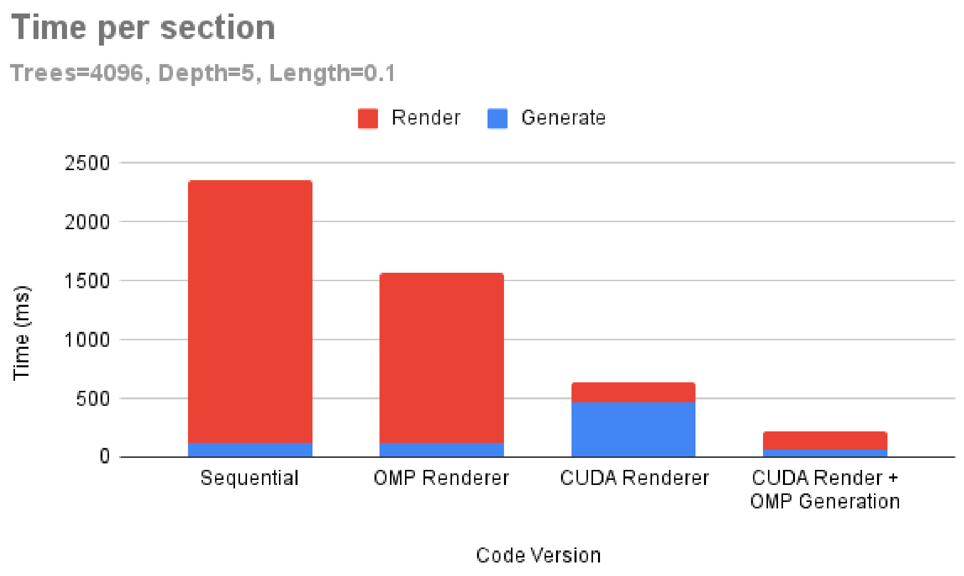


In this experiment, we fixed the number of trees to 4096 and the depth to 5 and varied the length of the trees in $\{0.001, 0.01, 0.1\}$. The length is used during rendering and affects how many pixels are shaded when drawing a tree. Firstly we see that the there is no change is speedup for the generate portion, as it does not depend on the length. Secondly, we see the the speedup for the rendering portion also stays almost constant. We hypothesize that this is because the shading of pixel is a computationally small part of the process compared to generating the actual endpoints of the lines from the instruction set. We found that most of the time spent by the CUDA renderer is on setting everything up and transferring data so the time to render stays relatively constant regardless of the quantity of rendering work.

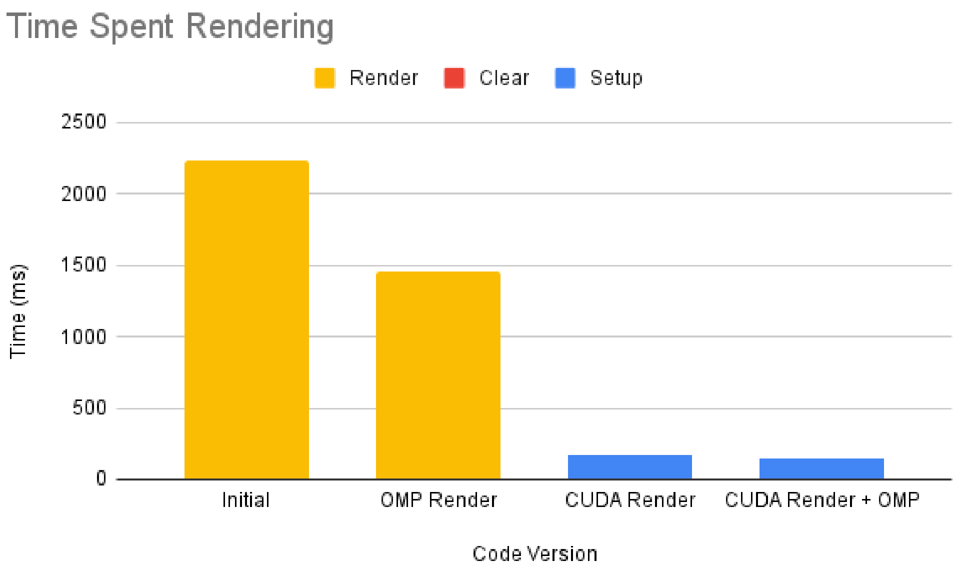


In this experiment, we fixed the number of trees to 4096 and the length to 0.1 and varied the depth of the trees in $\{4, 5, 6, 7, 8\}$. The depth affects both the generation and rendering processes, as there are more levels of updates in the generation phase, resulting in more lines to draw in the rendering phase. While the final CUDA version provides little to no speedup at first, speedup increases to 500x at depth 8. Since there are only 32 trees at depth 4 at the beginning, we hypothesize the initial lack of speedup / slowdown is caused by the lack of enough work to parallelize along with overhead of parallelization. At higher depths however, the exponentially large number of lines allows the CUDA render to achieve much better speedups.

4.3 Time Analysis



This final graph shows how we improved runtime of the program with each iteration and also how we decided where to spend our efforts. We decided to focus on parallelizing the generation only after it was the majority of the runtime of the program.



This graphs shows the distribution of time within the rendering section. For the OpenMP rendering, the setup just consisted of setting some global variables taking negligible time. The CUDA renderer on the other hand copied all the required data over to device memory during the setup, making it a large portion of the rendering runtime. This time also poses a bottleneck to further parallelization, since the actual rendering takes a negligible amount of time and optimizing it further would not improve total runtime much. If a different method of parallelization required less data to copied, then the setup costs could potentially be significantly reduced.

Given the fact that we were rendering to a display, the choice of a GPU for parallelization was a sound choice, which is confirmed by the results seen in the improvements to rendering when we started to use the CUDA kernels on the GHC GPUs.

4.4 Future Improvements

Since both the parts have around equivalent runtime now, either would be a good candidate for further parallelization. The generation could be parallelized within each tree, such that multiple cores could create the next iteration of a single string in parallel. This would be especially useful for deeper trees where the sizes of the strings exponentially increase, although they would not provide much benefit when generating many shallow trees. Another place where we could parallelize the generation is where we create the endpoints of the lines to pass to CUDA renderer. Multiple cores could work on a single string at once, giving speedups at higher depths. The rendering could also be parallelized in using other methods, such as parallelizing over each pixel, and drawing it if is on any of the lines, although it is not obvious if this would give a speedup.

5 Distrubution of Credit

Distribution - 50-50

Nolan Mass:
LSystem classes / generation
Line generation
Generation parallelization
Report approach / background
Presentation slides

Joshua Mathews:
Tree rendering
CUDA parallelization
Report summary / results / figures

References

[1] Wikipedia contributors, "L-system." `https://en.wikipedia.org/wiki/L-system#Example_7:_Fractal_plant`, 2022.

[2] Wikipedia contributors, "Bresenham's line algorithm." `https://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm`, 2022. [Online; accessed 5-May-2022].

[3] NVIDIA, "Cuda toolkit documentation." `https://docs.nvidia.com/cuda/`, 2022. [Online; accessed 5-May-2022].