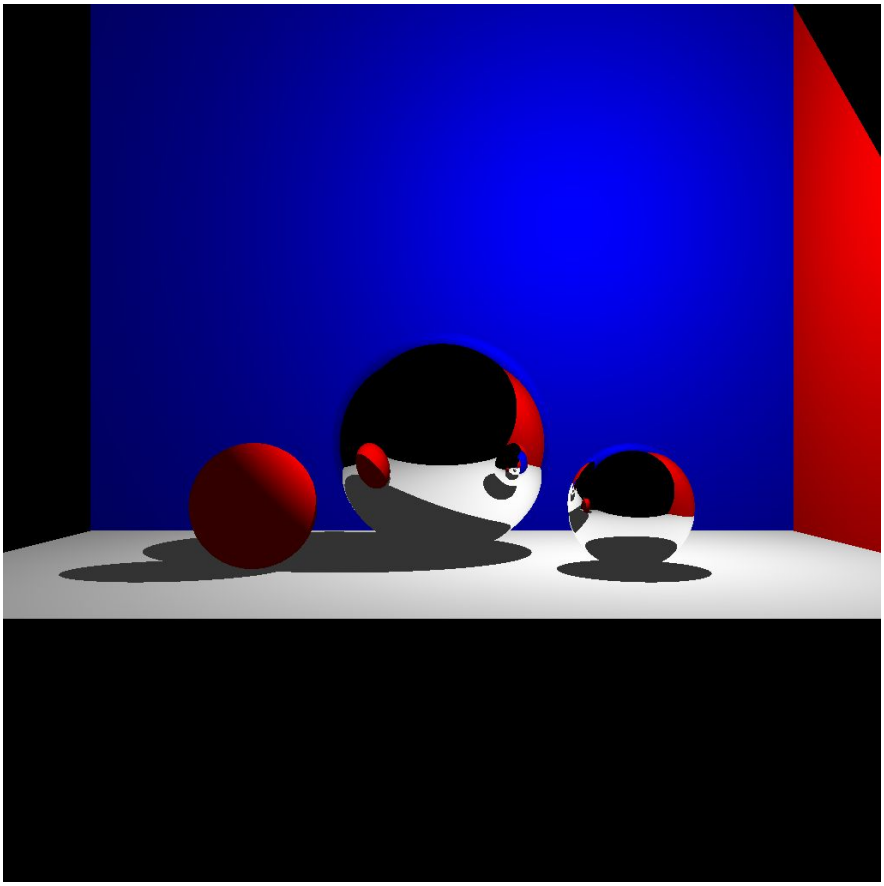
Jordan Jones
GPU & Multicore Programming

Final Report

Ray Tracing Project



1024x1024 output from CUDA ray tracer

Objective:

The goal of this project was to produce a dramatic reduction in time needed to produce a ray-traced image like the one above. To achieve this, the program was re-written in CUDA and optimized with some of the methods discussed in-class.

Implementation:

The original program made liberal use of structs to define things like Points and Rays. These structs were then used in arrays to allow for iterating over a large number of them. This made the program easier to understand due to specific struct types, but didn't lend itself well to parallelization in CUDA. To that end, the Arrays of Structures (AoS) were re-implemented as a Structure of Arrays (SoA). In the process, the "API" the ray tracer was based on (written in ray.h) was changed to use arrays rather than structs to represent things like points and vectors. This resulted in a ray tracing API and data structure that, while more difficult to understand and follow, required less overhead.

Parallelization was more complex. Thankfully, ray tracing is a somewhat-embarrassingly parallel application. Each pixel is completely independent of the other, meaning all the pixels can be ray-traced concurrently. For this application, a two-dimensional grid of three-dimensional blocks was used. Each block consists of X by Y by Z threads, where Z is the number of objects being rendered (e.g. spheres or triangles), and where X and Y are calculated based on Z such that there are not too many threads in the block. Each set of threads at (x, y) works on ray tracing for one pixel, with each thread in the set calculating the intersection of the pixel's ray and one object to be rendered. The two-dimensional grid of these blocks is made large enough such that there are enough of the blocks to cover the entire image.

Implementing reflections required multiple instances of ray tracing within each thread. Unfortunately, the CCSR server could not run such a kernel and would consistently return an error saying "Not enough resources". However, this kernel can be run on devices with more modern graphics capabilities.

Performance:
Since the CUDA program could not be run on CCSR, it was run on two devices:

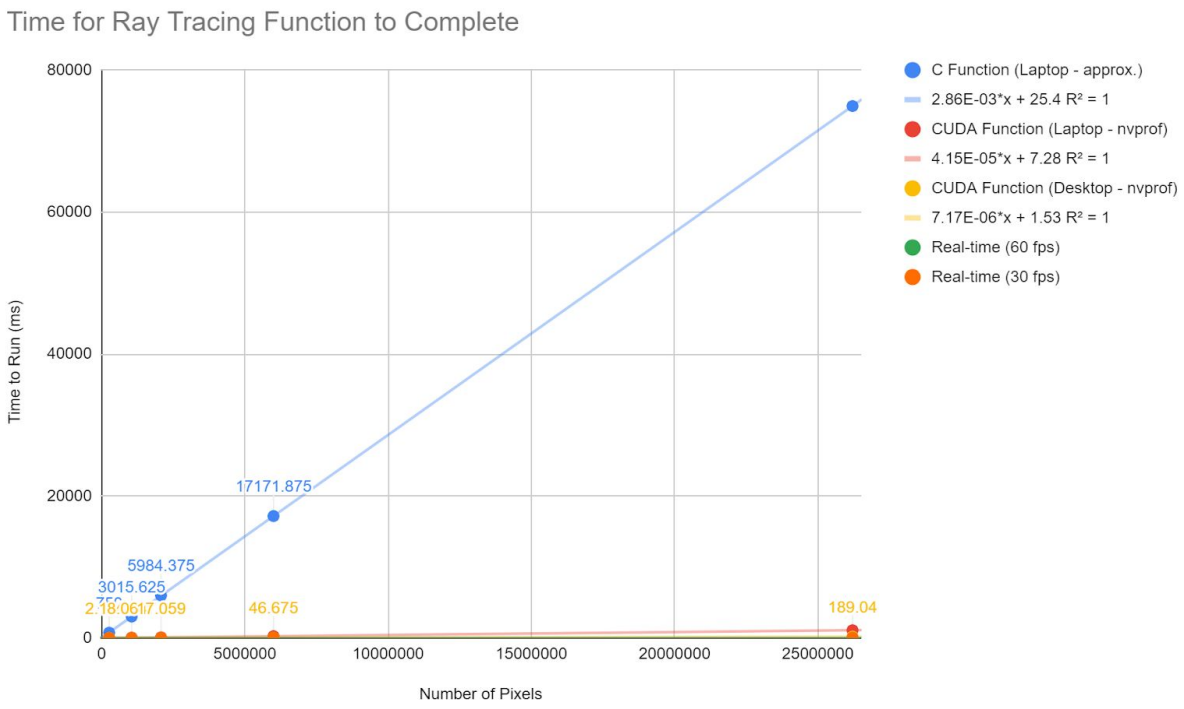
- A 1st-generation Microsoft Surface Book with Intel Core i5 and discrete Nvidia graphics
- A desktop computer with Intel Core i5-3570K and Nvidia GTX 1060 6GB graphics card

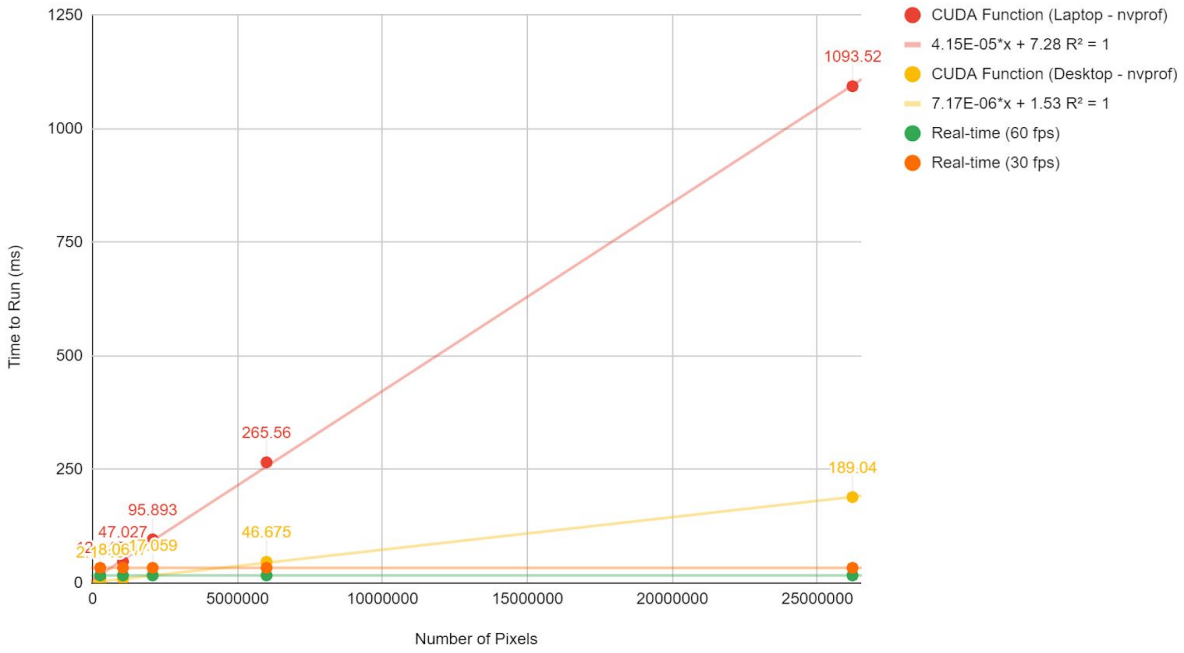
Performance was tested for only the ray tracing functions of each program. For the C program, the ray tracing functionality was copied to a separate function. In the CUDA program, the ray tracing functionality was limited to the kernel. C program performance was recorded using calls to get process clock time before and after the ray tracing function call, and the elapsed time calculated and printed to the screen. CUDA program performance was recorded using nvprof.

Each program was tested using the same resolutions (see table below), and the resulting execution times were graphed (see figures below). The time to run was charted relative to the total number of pixels being rendered, equal to the product of the x-resolution and y-resolution of the desired image.

Time to Run (in ms) for C and CUDA programs at each resolution

xRes	yRes	Total Number of Pixels	C Function (Laptop - approx.)	CUDA Function (Laptop - nvprof)	CUDA Function (Desktop - nvprof)
512	512	262144	750	12.345	2.1148
1024	1024	1048576	3015.625	47.027	8.0647
1920	1080	2073600	5984.375	95.893	17.059
3000	2000	6000000	17171.875	265.56	46.675
5120	5120	26214400	74937.5	1093.52	189.04


Time for Ray Tracing Function to Complete
80000
60000
40000
20000
0
Time to Run (ms)
0
5000000
10000000
15000000
20000000
25000000
Number of Pixels
17171.875
5984.375
3015.625
2.18:06
17.059
46.675
189.04
C Function (Laptop - approx.)
2.86E-03*x + 25.4 R² = 1
CUDA Function (Laptop - nvprof)
4.15E-05*x + 7.28 R² = 1
CUDA Function (Desktop - nvprof)
7.17E-06*x + 1.53 R² = 1
Real-time (60 fps)
Real-time (30 fps)


Time for Ray Tracing Function to Complete (w/out C Function)
1250
1000
750
500
250
0
Time to Run (ms)
0
5000000
10000000
15000000
20000000
25000000
Number of Pixels
1093.52
265.56
95.893
47.027
2.18:06
17.059
46.675
189.04
CUDA Function (Laptop - nvprof)
4.15E-05*x + 7.28 R² = 1
CUDA Function (Desktop - nvprof)
7.17E-06*x + 1.53 R² = 1
Real-time (60 fps)
Real-time (30 fps)

Analysis:
Trendlines were generated for each data set. A linear regression seemed to fit visually, and when assigned to the data resulted in an R^2 value of 1 for all trendlines, indicating they fit for the data.

As expected, the CUDA program offers a dramatic increase in performance over the C program. The rate of increase of processing time between the C program and the CUDA function when run on the laptop differ by two orders of magnitude. This difference is extreme, and to even see the differences in the runs of the CUDA program, the C program has to be omitted from the graph. With the C program omitted, the difference in performance between the laptop and the desktop is also dramatic, with the desktop outperforming the laptop by an order of magnitude in the rate of increase of processing time.

When compared to the maximum time possible to achieve 30 FPS (33.33 ms per frame) or 60 FPS (16.67 ms per frame) rendering, the laptop is only capable of either at the resolution of 512x512. Based on the trendline, the laptop would be capable of achieving 30 FPS rendering at up-to 627,791 pixels, and 60 FPS rendering at up-to 226,184 pixels. The desktop is capable of achieving the 30 FPS time at a resolution of 1920x1080, and the 60 FPS time at a resolution of 1024x1024. Based on the trendline, the desktop would be capable of achieving 30 FPS rendering at up-to 4,435,611 pixels, and 60 FPS rendering at up-to 2,111,111 pixels.

Interestingly, when looking at the y-intercept for each trendline (the predicted processing time if the program rendered zero pixels), each program and device have a different amount of overhead to ray trace. In the C program on the laptop, there is an overhead of 25.4 ms. In the CUDA program on the laptop, there's an overhead of 7.28ms, while in the CUDA program on the desktop, the overhead is only 1.53 ms. In the case of the CUDA program, comparing these values between the laptop and desktop seems to give a good approximation for their GPU performance relative to one another.

Other Considerations:
In performing these tests, it's important to note that the number of reflections were always limited to 10 per ray, and the number of objects to render was always 7. In a real ray tracing application (i.e. CGI or video games), there would likely be thousands to hundreds of thousands of objects to render, resulting in significantly lower performance than tested here.

Conclusion:
Implementing ray tracing in CUDA results in significant gains in performance over the C implementation. At specific resolutions, recent consumer-available GPUs are capable of performing the calculations fast enough to conceivably render a ray-traced scene in real-time at 30 or 60 FPS.

Possible Further Work:

- How would this ray tracing implementation run on top-of-the-line consumer and professional hardware?
- What further optimizations could be made to this implementation?
- How does this implementation compare to modern ray tracing techniques (i.e. Nvidia RTX)?
- How would this implementation perform with more objects to render?
- Do triangles or spheres have a greater performance impact than the other?
- Does the maximum number of reflections significantly affect performance?
- Does the number of reflective objects present significantly affect performance?