Comparing and Investigating a Ray Tracing Workload on CPU and GPU Systems

Jamin Valick

CSE 594, The Pennsylvania State University

1. Abstract:

This paper investigates ray tracing workloads and the performance differences between CPU and GPU implementations. The experiment for this research uses the AMD Ryzen 7 3700X CPU and a Nvidia RTX 3060 Ti GPU. The workload is written in C++ and CUDA and is based on Peter Shirley's "Ray Tracing in One Weekend". The performance is based on the runtime for a single image. Multiple images are used to find trends in varying resolution and samples per pixel. The results of this experiment show GPU speedups of up to 20x improvement over the CPU. This study shows how much more efficient the GPU is for parallel computing and computer graphics and the trade off in quality of images versus runtime costs.

2. Introduction:

Since the beginning of computer graphics, a paramount goal has been to produce more realistic images. Computer graphics are used in industries such as video games, film and animation, computer aided design, medical imaging, and graphic design. The prevalence of computer graphics is ever increasing as the quality improves. Such an improvement in computer generated images can be attributed to the advancements of both algorithms and hardware.

Games traditionally render frames utilizing rasterization. Although this method is very efficient for older and less powerful hardware, it is not ideal for producing highly realistic scenes. The ray tracing algorithm is an algorithm that has seen a surge in usage, especially in video games and animation. The algorithm simulates the transportation of light as rays to interact with a virtual environment and render an image. This can simulate reflections, refractions, and shadows with high accuracy. The concept of ray tracing is not new, but the technology to support such an intensive algorithm has recently matured enough to allow real time rendering with such a technique (Nvidia, 2018).

Central processing units are the standard device that many programmers develop for. CPUs have very high clock rates, are easy to program, and are used for completing general tasks. Unfortunately, they lack in completing a lot of very small tasks in parallel. Even a high-end multi-core CPU only has 24 cores. Alternatively, graphics processing units (GPUs) are powerful devices that have many simple cores (up to 1000s of cores) which can all work in parallel.

Images are made of many pixels. The color for each pixel can be independently calculated. This means that each core in a CPU or GPU can handle the calculation of a pixel in parallel. This paper will explore the differences in performance between a multi-core AMD CPU and an Nvidia GPU using a ray tracing workload written in C++ and CUDA and based on Peter Shirley's "Ray Tracing in One Weekend" and the GPU adaptation of Roger Allen's "Accelerated Ray Tracing in One Weekend in CUDA".

3. Background:

3.1 Ray Tracing:

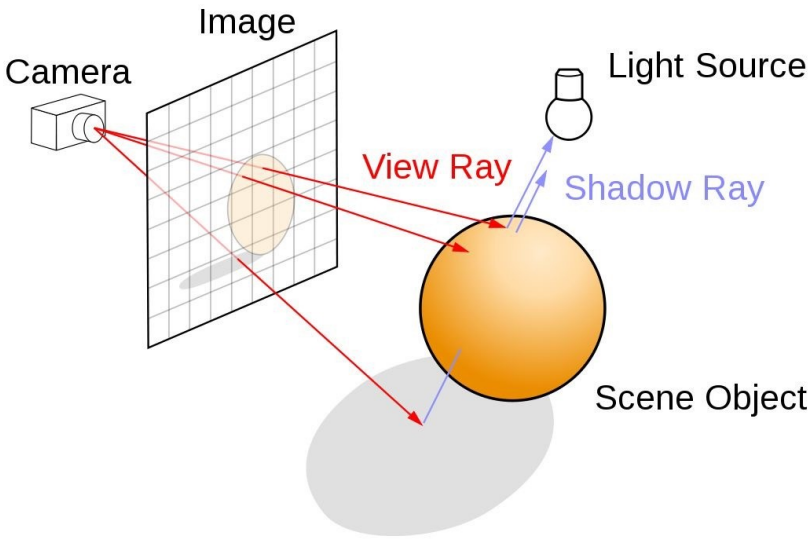

Figure 1: Ray Tracing Depiction (Nvidia, 2018).

Ray tracing is the simulation of light transportation. Typically, light is emitted from a source and enters the camera or viewer. The opposite is true in ray tracing as simulating all the rays from a light source (even ones that do not make it to the camera) is inefficient. Instead, rays are projected from a camera behind a through each pixel of a virtual plane that represents the real life screen the image will be displayed on. These rays continue into a scene of objects. The color of each pixel will be determined by the color of the location that the corresponding ray intersects with. The types of objects in this workload are diffuse, metal, and dielectric. Each material has its own separate class and models for coloring, reflection and refraction. This means that an object can reflect or refract rays to continue multiplying to a pixel's final color. These rays are recursively projected throughout the scene until it reaches a maximum recursion depth (this workload will have a maximum depth of 50) or fails to intersect with another object. At this point the pixel's final color is determined and is added to the overall image. For the sake of simplicity there are no light sources in the workload for this paper and thus no shadow rays. Instead, each object is lit with an ambient light and apparent shadows are handled by the material classes and properties (Shirley, 2018).

3.2 Variables:

Sampling is a factor in the quality of a ray traced image. Taking only one sample per pixel will produce an aliasing effect where edges are sharp and can appear irregular. Coloring is also less accurate as an object can be illuminated by many rays coming from many directions. These problems can be mitigated by taking multiple samples per pixel at slightly randomized directions and averaging them, but this may add noise if too few samples are taken. If enough randomized samples are taken, then a clean and accurate image will be produced. This experiment will use varying sampling rates, from 1 to 128 samples per pixel, to find patterns in performance.

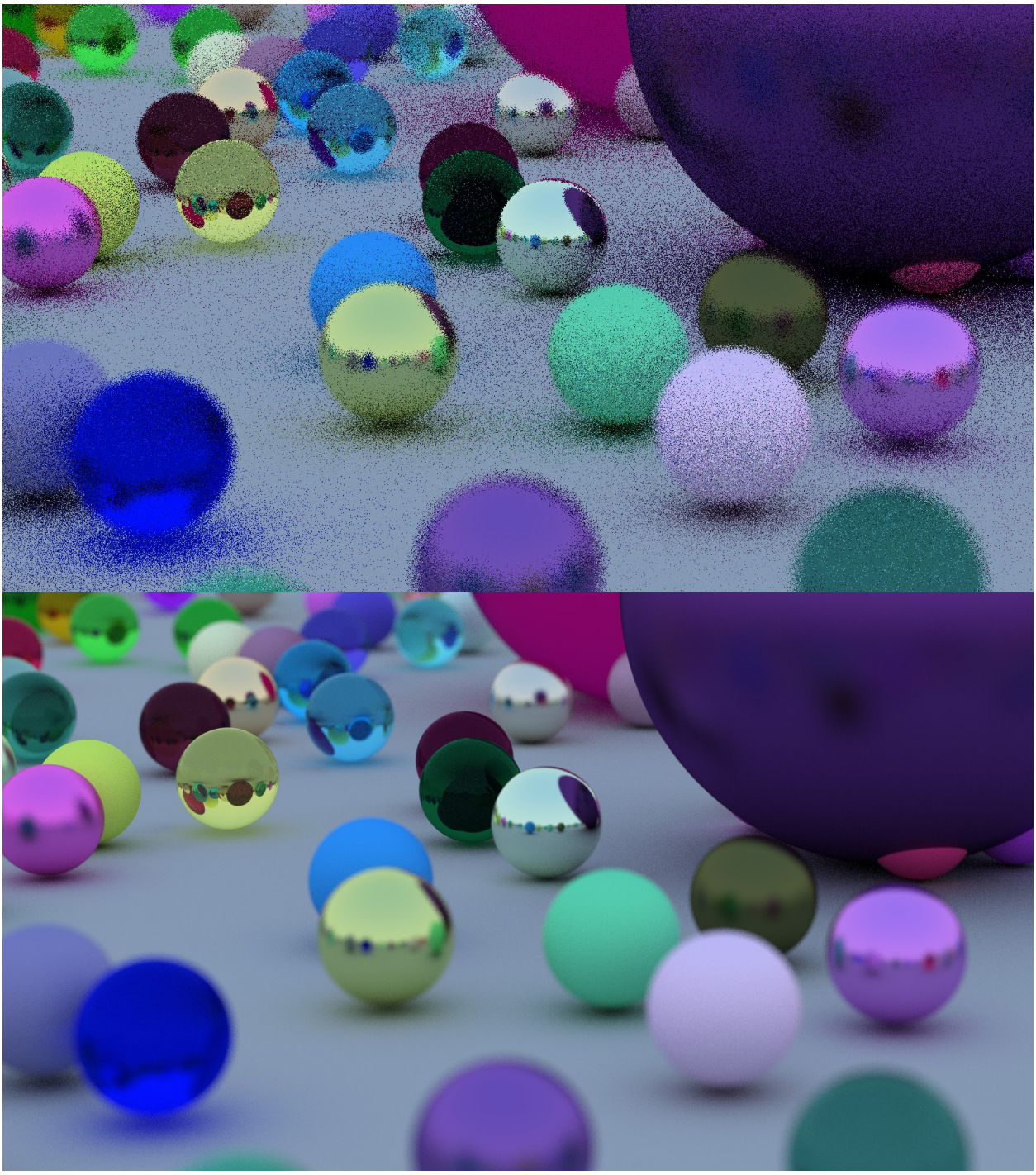

Figure 2: An image with 1 sample per pixel (top) versus an image with 128 samples per pixel (bottom)

Resolution is the second factor of quality for ray tracing. Increasing resolution will increase the number of pixels in the image. This will improve quality but will also increase the workload. This experiment will use resolutions from 144p to 2160p (4k).

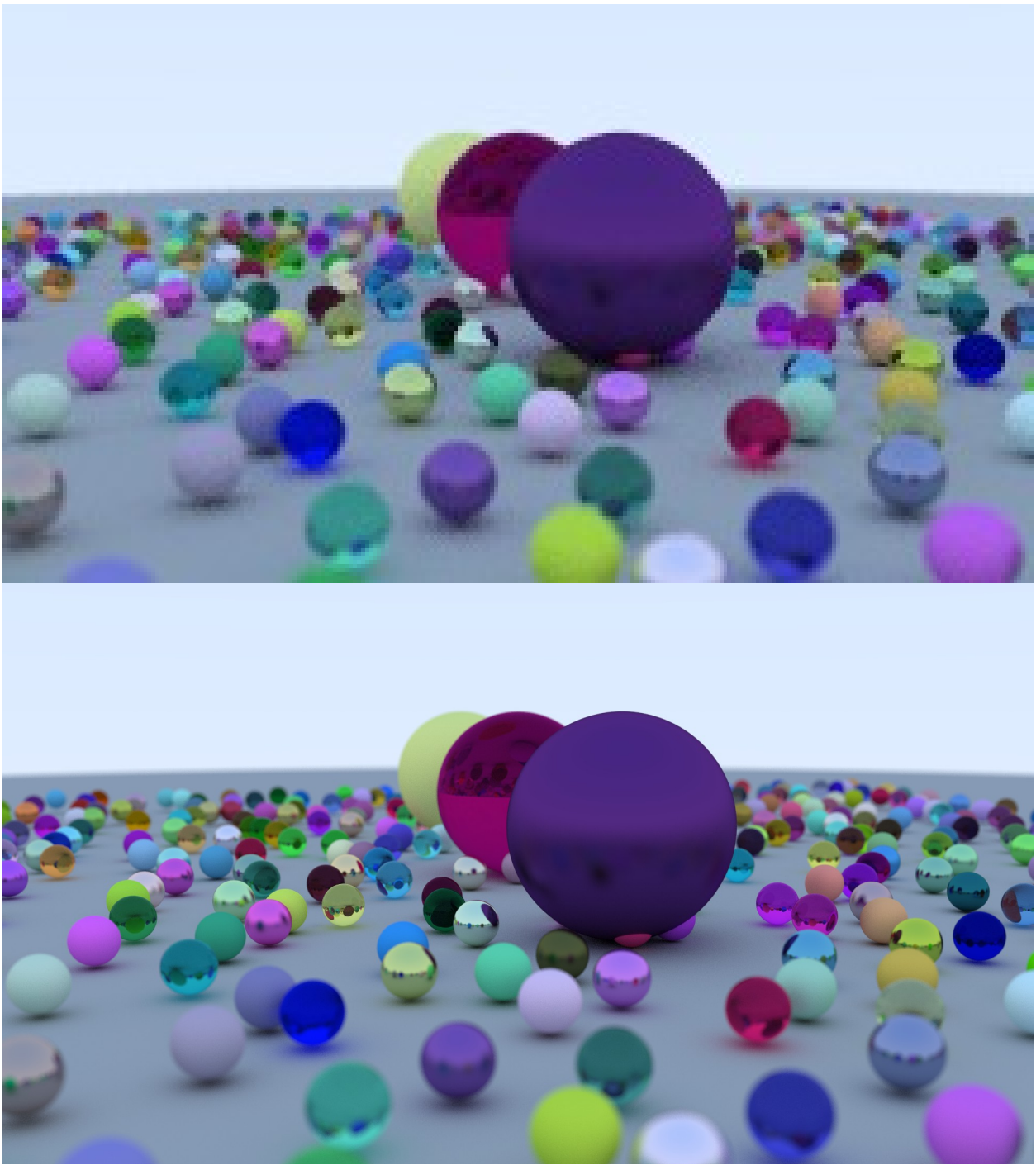

Figure 3: An image with 144p resolution (top) versus an image with 4k resolution (bottom)

note: The full resolution cannot be appreciated without the image being extended on a proper screen.

3.3 Parallelization:

The workload for even a simple ray tracer can be a lot for just a single thread on a CPU. Doing recursive path tracing, calculating the intersection of objects and their color, and taking multiple samples for each pixel in an image of thousands of pixels will show ever increasing runtimes. Fortunately, the calculation of each pixel is entirely independent from the rest of the image. This means that multiple threads can concurrently run on multiple pixels. An easy solution is to utilize multiple cores on a CPU. The overall algorithm of calling the ray tracing program and adding the color to an image can be represented with this nested for loop:

Figure 4: Parallelizable code (Valick, 2024).

OpenMP is an API that allows for C++ code to run on multi-core CPUs. The loops that progress through the pixels of the image can be parallelized with OpenMP pragmas. This will distribute the workload to multiple cores on a CPU and decrease runtime dramatically.

3.4 GPUs and CUDA:

Although improvements are made with parallelizing the workload on a CPU, there are still relatively few cores on a CPU compared to the numbers on a GPU. GPU cores are simple with small caches. CPUs are complex and highly optimized for single thread performance and latency. GPUs have the advantage of data processing with far more cores and lower context switching latency. If a program has many repetitive and mathematical based operations that have minimal dependencies, then a GPU is the better option (Gupta, 2020a).

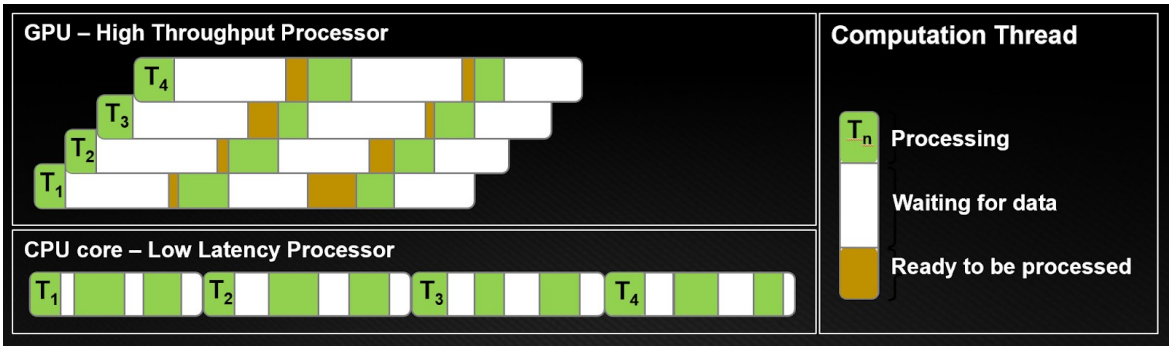

Figure 5: Latency comparison (Gupta, 2020a)

Nvidia GPUs are programed with a C++ API called CUDA. This allows programmers to take advantage of GPU parallelism with relative ease. Instead of using for loops, iterations are handled by a CUDA kernel. The kernel will run a group threads withing groups of blocks to perform a task. In this case each thread will be a pixel on the rendered image. Each thread is assigned to a CUDA core. Threads are grouped into blocks which are assigned to groups of cores

called streaming multiprocessors (SM). The entire image will be assigned to groups of blocks called kernel grids. This kernel grid is run on groups of SMs on a single device (Gupta, 2020b).

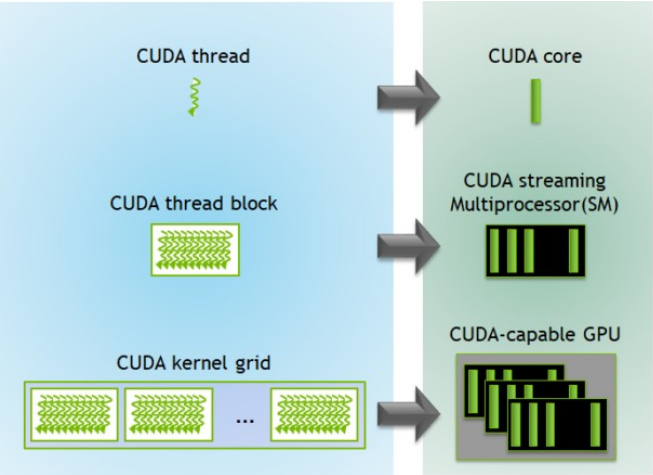

Figure 6: Thread groupings and hardware parallels (Gupta, 2020b)

Another major change that must be made is converting the ray tracing recursion into iterative calling. Since each ray that intersects with an object can refract or reflect, the ray tracer recursively calls on itself until the recursion depth is reached or the ray fails to intersect. GPUs have limited memory sizes and will end with stack overflows if the recursion is too deep. Fortunately, the simple solution is to turn the recursive calls into an iterative for loop (Allen, 2018).

3.5 Experiment

This experiment uses the AMD Ryzen 7 3700X with 8 cores and 16 threads as the CPU and the Nvidia RTX 3060ti with 4864 CUDA cores as the GPU. It tests the C++ based ray tracing workload on the CPU and GPU independently. A scene of spheres and their properties is generated as the input to the ray tracer to test for each image iteration. The performance is measured as runtime in seconds. This runtime is for calculating pixel values with raytracing, shading, and adding them into an array of pixels which represents the image itself. I took these measurements with basic time libraries from start to finish of the algorithm and found trends in runtime and speedup between devices using Excel and MATLAB. Each device rendered 64 images where each image has the following combinations of variable values:

Resolution (pixels):

- 256 x 144 (144p)
- 426 x 240 (240p)
- 640 x 360 (360p)
- 854 x 480 (480p)
- 1280 x 720 (720p)
- 1920 x 1080 (1080p)
- 2560 x 1440 (1440p)
- 3840 x 2160 (2160p)

Samples per pixel:

- 1
- 2
- 4
- 8
- 16
- 32
- 64
- 128

4. Results:

CPU Runtime (seconds)								
Samples/Resolution	**1**	**2**	**4**	**8**	**16**	**32**	**64**	**128**
144p	0.040	0.080	0.156	0.286	0.684	1.172	2.208	4.356
240p	0.097	0.185	0.388	0.729	1.559	3.099	5.978	11.859
360p	0.210	0.436	0.839	1.728	3.502	6.612	13.500	27.473
480p	0.399	0.773	1.510	3.110	6.121	11.923	23.752	47.589
720p	0.853	1.723	3.324	6.666	13.560	27.137	53.234	107.868
1080p	1.910	3.766	7.569	15.403	30.151	59.938	120.826	234.835
1440p	3.548	8.288	13.938	27.265	53.360	107.559	212.177	423.193
2160p	7.473	15.033	29.847	60.260	118.628	238.288	473.701	972.572

Figure 7: CPU Runtime as a function of samples per pixel and resolution.

GPU Runtime (seconds)								
Samples/Resolution	**1**	**2**	**4**	**8**	**16**	**32**	**64**	**128**
144p	0.01	0.02	0.042	0.074	0.132	0.256	0.494	0.996
240p	0.013	0.024	0.043	0.088	0.185	0.348	0.671	1.331
360p	0.02	0.037	0.07	0.136	0.302	0.526	1.056	2.022
480p	0.029	0.056	0.102	0.199	0.401	0.791	1.575	3.18
720p	0.054	0.104	0.203	0.392	0.776	1.536	3.072	6.178
1080p	0.111	0.209	0.411	0.815	1.638	3.238	6.532	13.217
1440p	0.185	0.366	0.722	1.449	2.859	5.723	11.318	22.8
2160p	0.406	0.809	1.594	3.138	6.266	12.474	24.992	49.695

Figure 8: GPU Runtime as a function of samples per pixel and resolution.

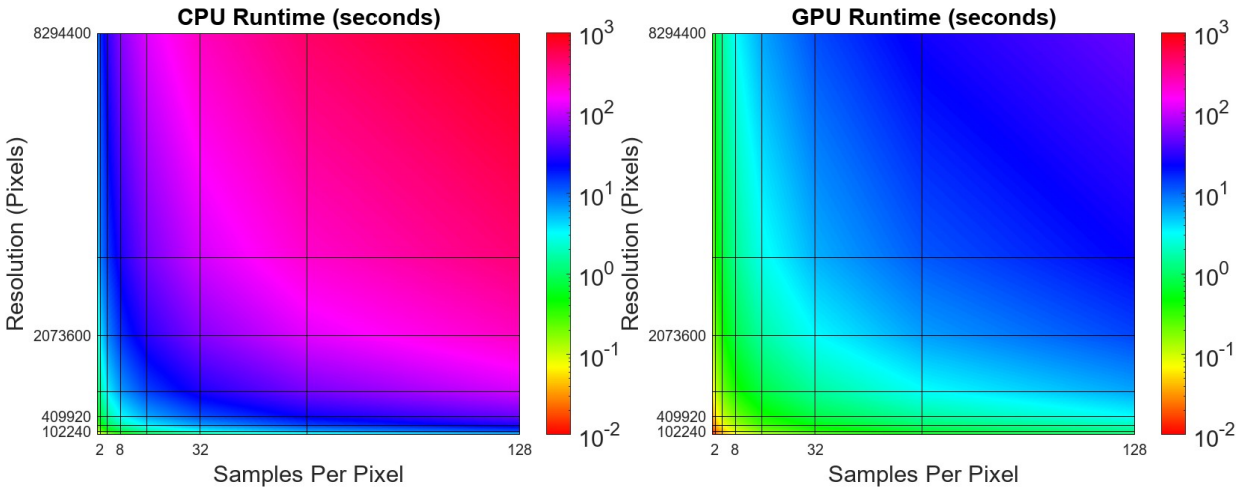

Figure 9: Color map of CPU and GPU runtimes on the same scale as a function of samples per pixel and resolution.

The tables and graphs above show the raw runtime for varying image qualities. It is evident that the GPU outperforms the CPU for all image qualities. The color map illustrates the scale of the difference between the two devices.

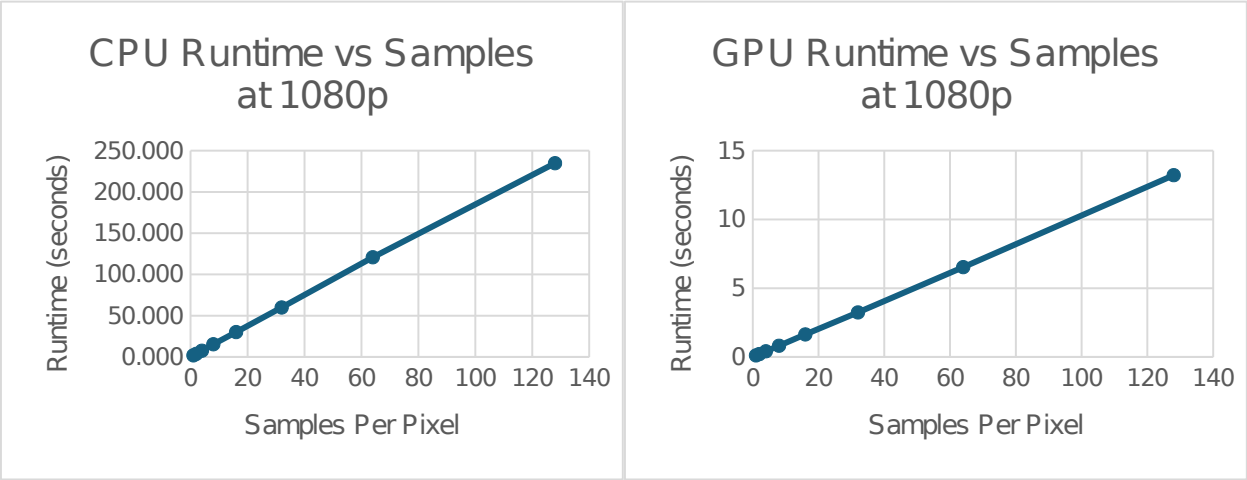

Figure 10: CPU and GPU runtimes as a function of only samples per pixel

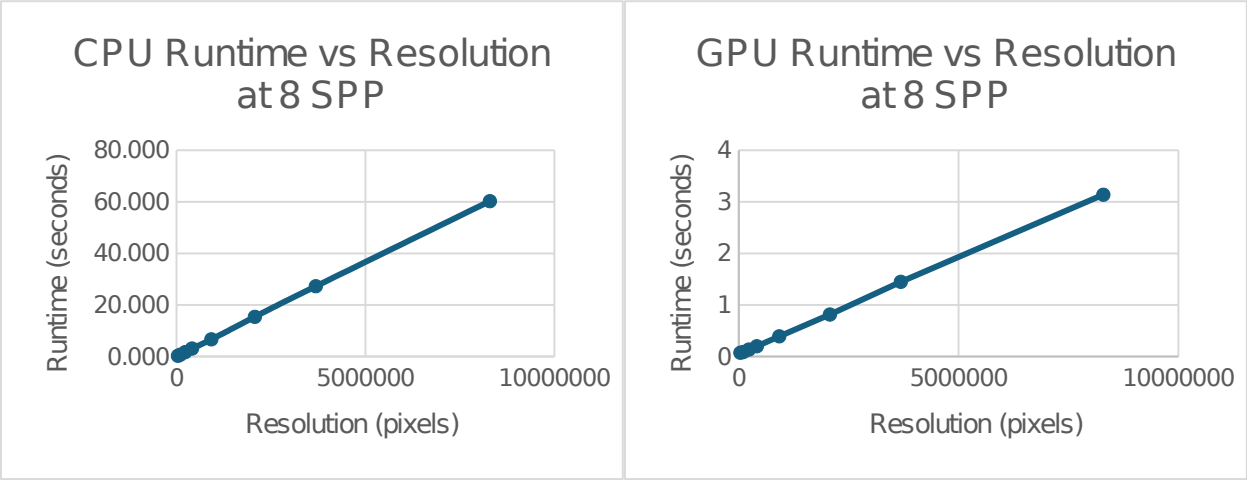

Figure 11: CPU and GPU runtimes as a function of only resolution

Both samples per pixel and resolution increase runtimes linearly.

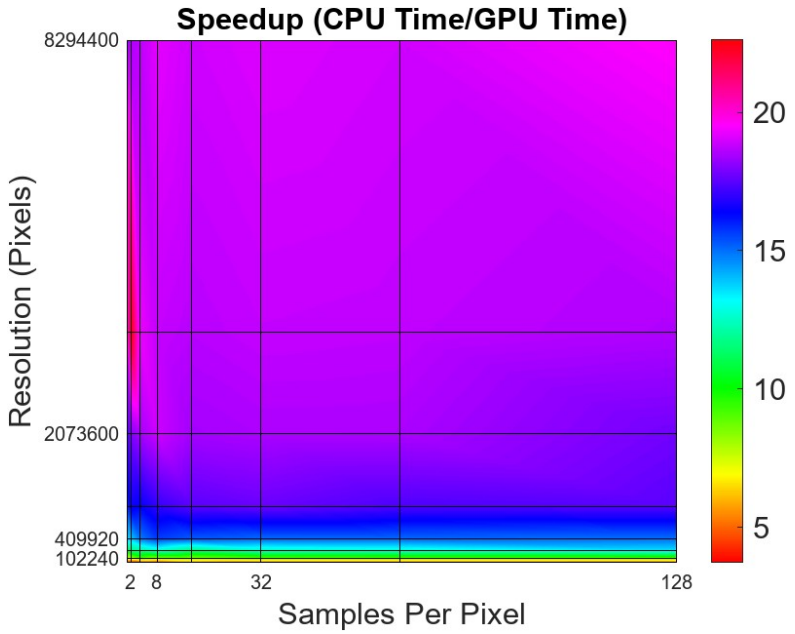

Figure 12: Speedup as a function of samples per pixel and resolution.

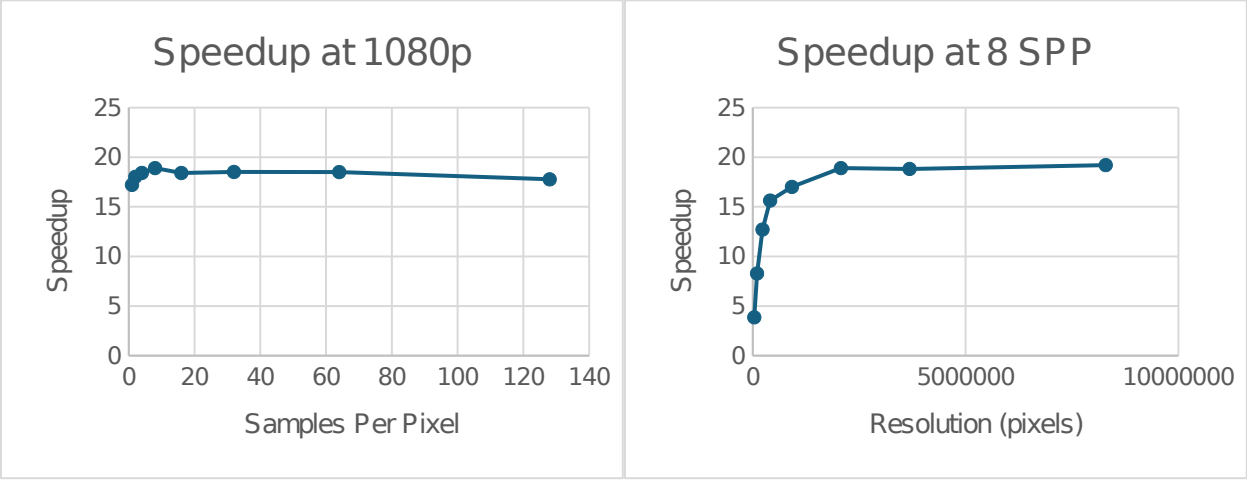

Figure 13: Speedup as a function of individual variables.

The above tables and graphs show speedup factors as CPU Time/GPU Time. Speedup is nearly constant with samples per pixel. The amount of work done per thread or the samples per pixel does not affect speedup. On the other hand, speedup is logarithmic for change in resolution. As resolution increases the factor between performance plateaus close to a 20x improvement.

5. Conclusions:

The goal of this project was to implement a ray tracing workload on both the CPU and GPU and explore performance differences between the two. The GPU has up to a 20x improvement over a CPU. CUDA and parallel processing are powerful tools but even these have limits. No matter how powerful a device is there is a delicate balance of quality and performance when

making decisions on rendering with ray tracing. Overall, this was a learning experience with many facets. This research project covered ray tracing, GPU architecture, parallel programing, and CUDA programming.

Although the GPU is a powerful device for ray tracing, improvements can still be made to optimize the ray tracing algorithm. Denoising algorithms, ray-box intersection, adaptive sampling, and machine learning are all topics of improving quality and performance in ray tracing. Multi-frame rendering is also important for measuring performance as many ray tracing applications are for videogames and animation. Finding optimizations for rendering images frame by frame is important for increasing frame rates, especially in video games where players demand high FPS.

References:

Allen, R. (2018, November 5). *Accelerated Ray Tracing in One Weekend in CUDA*. NVIDIA Technical Blog. https://developer.nvidia.com/blog/accelerated-ray-tracing-cuda/

Gupta, P. (2020a, April 24). *CUDA Refresher: Reviewing the Origins of GPU Computing*. NVIDIA Technical Blog. https://developer.nvidia.com/blog/cuda-refresher-reviewing-the-origins-of-gpu-computing/

Gupta, P. (2020b, June 26). *CUDA Refresher: The CUDA Programming Model*. NVIDIA Developer Blog. https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/

Nvidia. (2018, August 14). *Ray Tracing*. NVIDIA Developer. https://developer.nvidia.com/discover/ray-tracing

Shirley, P. (2018). *Ray Tracing in One Weekend*.

Valick, J. (2024, April 22). *Final Project: Parallelization of Ray Tracing Algorithm*.