ME 766 Project Report

Anubhav Bhatla, Arya Vishe, Rishabh Ravi, Sarvadnya Desai
200070008, 21D070018, 200260041, 210040138

Accelerating RayTracing

December 10, 2024

Contents

System Configuration

Model	AMD EPYC 9554
Cores	64
Threads per core	2
Memory	128GB
OS	Ubuntu 24.04.1
GPU	NVIDIA A6000
Memory	48GB

Table 1: System configuraton used for collecting all results.

1 Introduction & Motivation

Raytracing is a technique used to model light transport in graphic rendering algorithms to generate high quality digital images. In the late 2000s, ray-tracing started gaining action in the video games industry for real-time graphics. Ray tracing is capable of simulating realistic optical effects such as reflection, refraction, scattering soft shadow, motion blur, depth of field, etc.

Ray tracing works by simulating how light travels and interacts with objects in a virtual scene. It starts with a virtual camera that "shoots" rays of light into the scene, tracing their paths as they bounce off surfaces, collect color and lighting information, and ultimately determine the color of each pixel displayed on the screen.

Ray tracing also takes into account properties of material such as opacity, nature of material, texture and geometry and performs calculation of on the scattering of the image light.

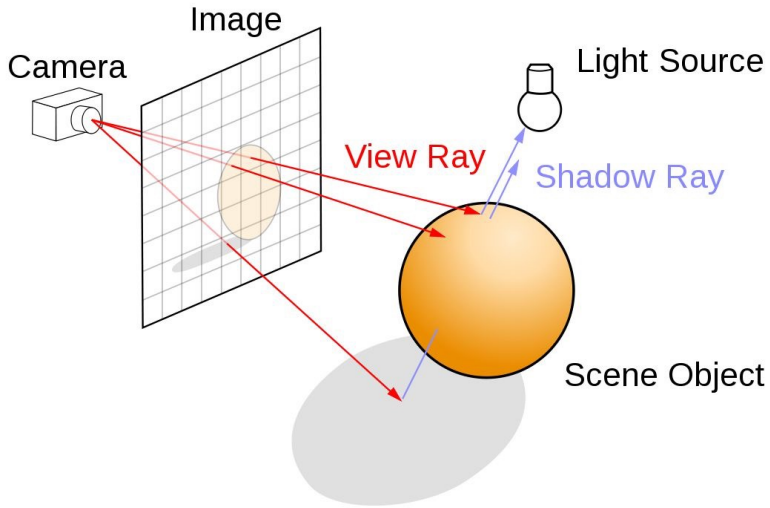

Figure 1: Physics Simulation behind Raytracing

2 Code Analysis

Below is a call graph of the serial ray tracing program named 'inOneWeekend' which can be found here.

```
79.87%    0.00% inOneWeekend inOneWeekend        [.] main
   |
   ---main
      |
       --79.85%--camera::render(...)
             |
              --79.06%--camera::ray_color(...)
                    |
                    |--49.05%--camera::ray_color(...)
                    |         |
                    |         |--24.34%--camera::ray_color(...)
```

```
                    |         |
                    |          --23.13%--    hittable_list::hit(...)
                    |
                     --28.02%--hittable_list::hit(...)
                           |
                            --14.13%--sphere::hit(...)
```

We observe that a large amount of time is spent inside the function 'render' which is responsible for rendering the 2D scene. Let us analyze the code snippet for the render function and see if we can parallelize it anyway.

```
void render(const hittable& world) {
initialize();

color* pixel_color = (color*)malloc(image_width*image_height*sizeof(color));

for (int j = 0; j < image_height; j++) {
    for (int i = 0; i < image_width; i++) {
        for (int sample = 0; sample < samples_per_pixel; sample++) {
        ray r = get_ray(i, j);
        pixel_color[i + image_width*j] += ray_color(r, max_depth, world);
        }
    }
}
```

We observe a chain of nested for loops rendering pixels based on the ray data and the world objects. We can parallelize this section with multiple threads running at once as the data dependence between executions is not serial in nature and can be executed simultaneously.

2.1 OpenMP

2.2 Code

```
void render(const hittable& world) {
initialize();

color* pixel_color = (color*)malloc(image_width*image_height*sizeof(color));

#pragma omp parallel for firstprivate(image_height,image_width,
    samples_per_pixel,max_depth) shared(world,pixel_color)
for (int j = 0; j < image_height; j++) {
    for (int i = 0; i < image_width; i++) {
        for (int sample = 0; sample < samples_per_pixel; sample++) {
        ray r = get_ray(i, j);
        pixel_color[i + image_width*j] += ray_color(r, max_depth, world);
        }
    }
}
```

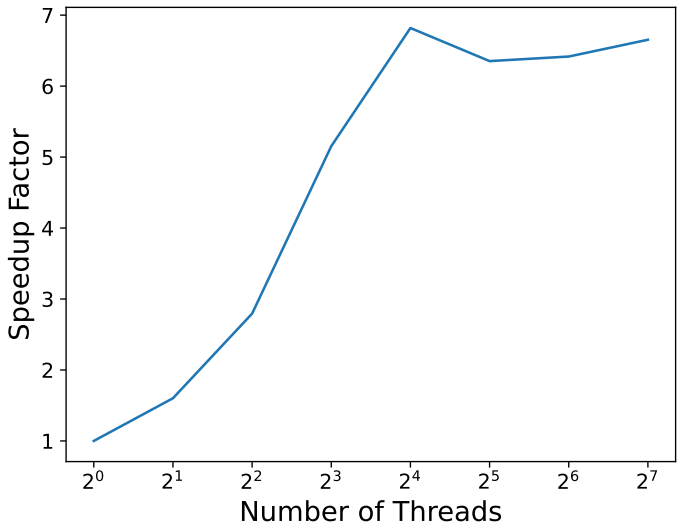

Figure 2: Speedup vs threads for OpenMP for 1920×1080.

2.3 Results

#Threads	1st Run	2nd Run	3rd Run	4th Run	5th Run	Avg. Runtime	Speedup (Ψ)
1	381.601	384.011	384.000	383.067	383.224	383.181	1.00000000
2	240.721	238.748	238.878	240.031	237.752	239.226	1.60175315
4	137.471	137.259	137.081	137.236	136.575	137.124	2.79441236
8	74.150	74.129	74.709	74.506	74.289	74.357	5.15326062
16	56.382	55.425	55.932	56.322	56.922	56.197	6.81853124
32	60.105	60.400	60.220	60.466	60.414	60.321	6.35236485
64	59.324	59.846	59.870	59.635	59.905	59.716	6.41672249
128	57.835	57.138	57.297	57.931	57.766	57.593	6.65325647

Table 2: Execution time for OpenMP for image size 1920×1080.

3 CUDA

3.1 Code

Ray tracing requires a camera and world of spheres to be created, using which it renders the pixels. We start with performing these operations on the CPU and then share the camera and world with the GPU to render the pixels. Figure 3 shows the flow for ray tracing. Rendering involves two functions `get_ray` and `ray_color` which are now device functions called by the GPU. These functions are part of `camera.h` which further calls functions from various header files, all of which need to be made compatible with running on the GPU.

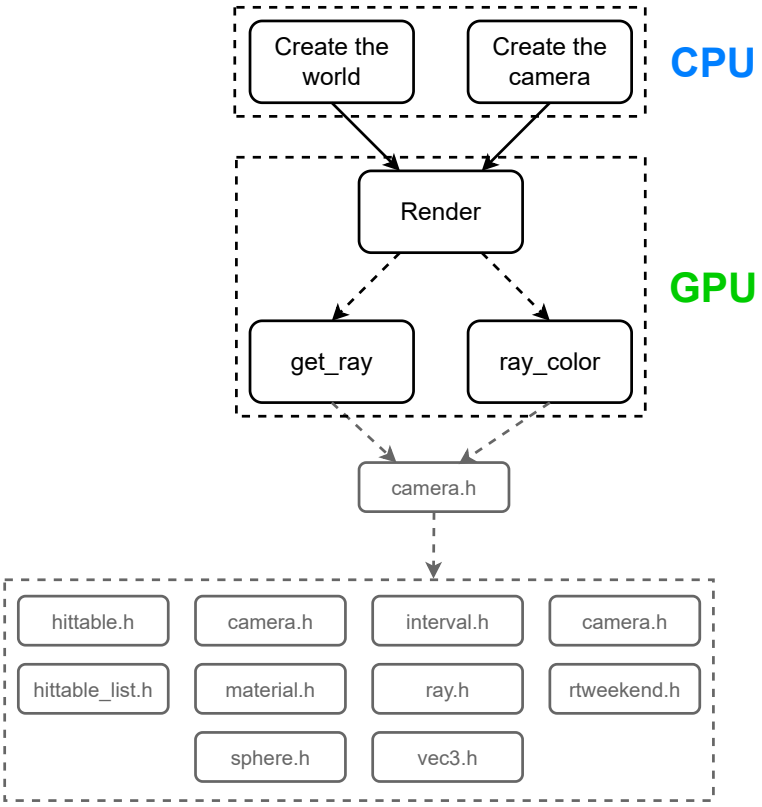

Figure 3: Flow of ray tracing when GPU is used to render only.

3.2 Optimization

What if we perform the creation of the world and camera on the GPU instead of the CPU? Figure 4 shows the new flow with all operations being performed on the GPU. In order to create these entities on the GPU, we need to use CUDA's own `cuRand` function, which is then used to create the world and initialize render. This world and camera can now directly be used by the GPU to render pixels and there's no communication overhead from the CPU to GPU.

3.3 Results

#Threads	1st Run	2nd Run	3rd Run	4th Run	5th Run	Avg. Runtime	Speedup (Ψ)
1	14.348	14.529	14.696	14.839	14.935	14.670	1.00000000
2	5.186	5.180	5.183	5.185	5.186	5.184	2.82986111
4	2.213	2.212	2.203	2.199	2.215	2.208	6.64402174
8	2.399	2.397	2.387	2.399	2.407	2.398	6.11759800
16	2.455	2.438	2.436	2.465	2.452	2.450	5.98775510
32	2.823	2.844	2.839	2.799	2.814	2.824	5.19475921

Table 3: Execution times for CUDA for image size 1920×1080.

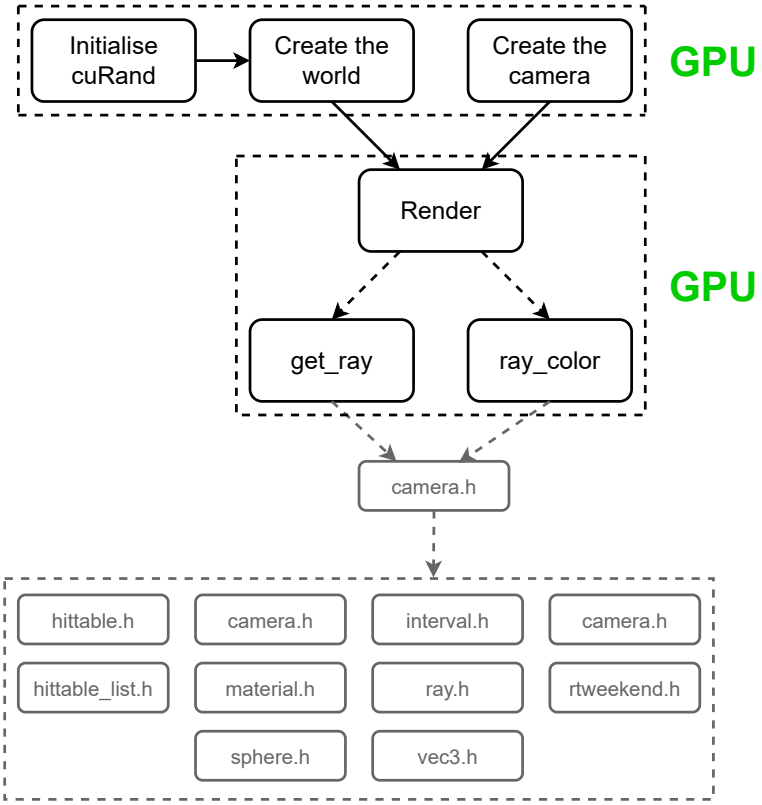

Figure 4: Flow of ray tracing when GPU is used to perform all operations.

#Threads	1st Run	2nd Run	3rd Run	4th Run	5th Run	Avg. Runtime	Speedup (Ψ)
1	25.344	25.764	25.763961129	26.561	26.626	26.012	1.00000000
2	9.224	9.224	9.127	9.115	9.102	9.130	2.84906900
4	3.816	3.763	3.746	3.755	3.756	3.770	6.89973475
8	4.104	4.055	4.050	3.974	4.019	4.041	6.43702054
16	4.033	4.282	4.249	4.237	3.992	4.165	6.24537815
32	4.829	4.827	4.990	4.998	5.015	4.899	5.30965503

Table 4: Execution times for CUDA for image size 2560×1440.

#Threads	1st Run	2nd Run	3rd Run	4th Run	5th Run	Avg. Runtime	Speedup (Ψ)
1	58.383	59.727	60.153	60.283	60.142	59.736	1.00000000
2	20.462	20.430	20.448	20.400	20.421	20.432	2.87518355
4	8.390	8.300	8.219	8.244	8.261	8.283	7.21497585
8	9.126	9.146	9.281	8.870	9.051	9.094	6.57205721
16	9.130	9.296	8.822	9.008	9.110	9.073	6.58654906
32	11.374	10.362	11.055	11.418	11.356	11.113	5.37713771

Table 5: Experiment runtimes for CUDA for image size 3840×2160

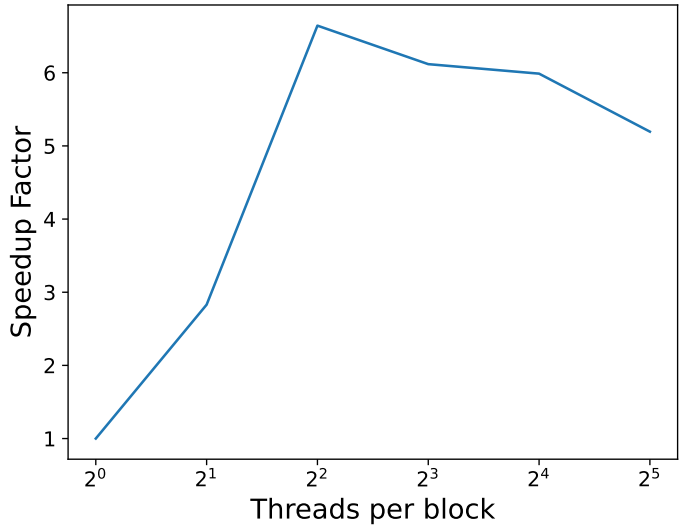

Figure 5: Speedup vs threads for CUDA for image size 1920×1080.

#Threads	1st Run	2nd Run	3rd Run	4th Run	5th Run	Avg. Runtime	Speedup (Ψ)
1	237.962	240.295	240.118	240.191	240.288	239.771	1.00000000
2	81.015	81.100	81.403	81.121	81.131	81.154	2.95451857
4	32.787	32.766	32.598	32.769	32.662	32.716	7.32886050
8	37.020	37.093	37.273	37.160	37.149	37.139	6.45604351
16	36.799	36.795	36.754	36.718	36.677	36.749	6.52455849
32	47.694	47.394	47.985	47.443	47.346	47.573	5.04006474

Table 6: Experiment runtimes for CUDA for image size 7680×4320

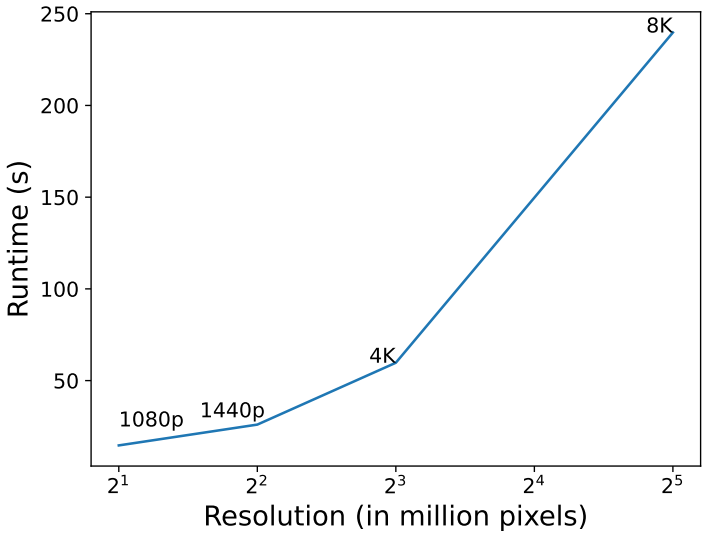

Figure 6: Runtime for CUDA for various common resolutions.

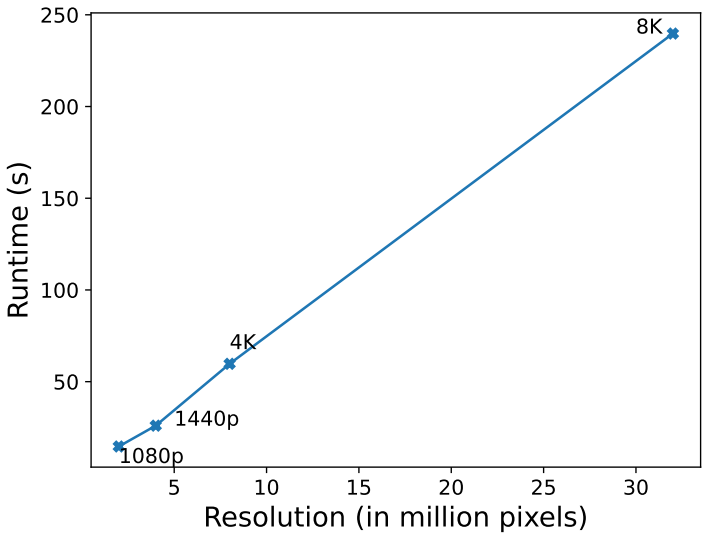

Figure 7: Runtime for CUDA for various common resolutions.

4 Analysis of results

$$\Psi(n,p) = \frac{T(n,1)}{T(n,p)}$$

$$e(n,p) = \frac{\frac{1}{\Psi(n,p)} - \frac{1}{p}}{1 - \frac{1}{p}}$$

We compute the speedup and the serial fraction of the CUDA and OpenMP parallelization of the render function. The serial fraction indicates the effect of the parallelization overhead in the program. A steady increase in the serialization fraction in the larger number of threads indicates greater contribution of parallel overhead in the parallel program.

#Threads	Speedup (Ψ)	Serial fraction (e)
2	1.60175315	0.24863185
4	2.79441236	0.14380932
8	5.15326062	0.07891645
16	6.81853124	0.08976976
32	6.35236485	0.13024174
64	6.41672249	0.14244348
128	6.65325647	0.14361180

Table 7: Speedup and Serial fraction for OpenMP for image size 1920×1080.

#Threads	Speedup (Ψ)	Serial fraction (e)
2	2.82986111	0.13783231
4	6.64402174	0.09387866
8	6.11759800	0.15018448
16	5.98775510	0.16374086
32	5.19475921	0.19171236

Table 8: Speedup and Serial fraction for CUDA for image size 1920×1080.

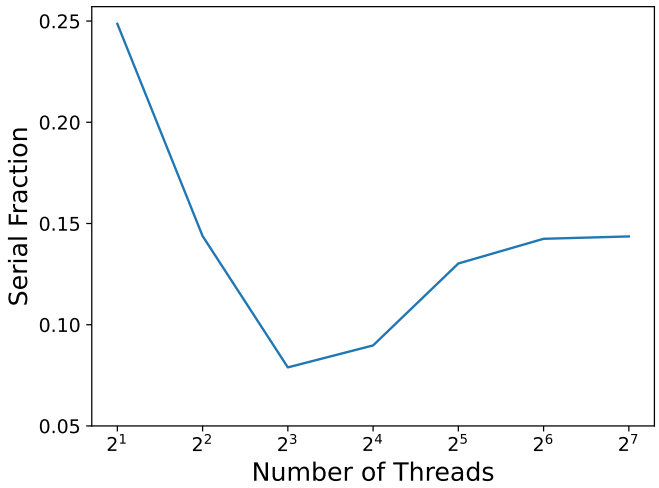

Figure 8: OMP Serialization fraction trend

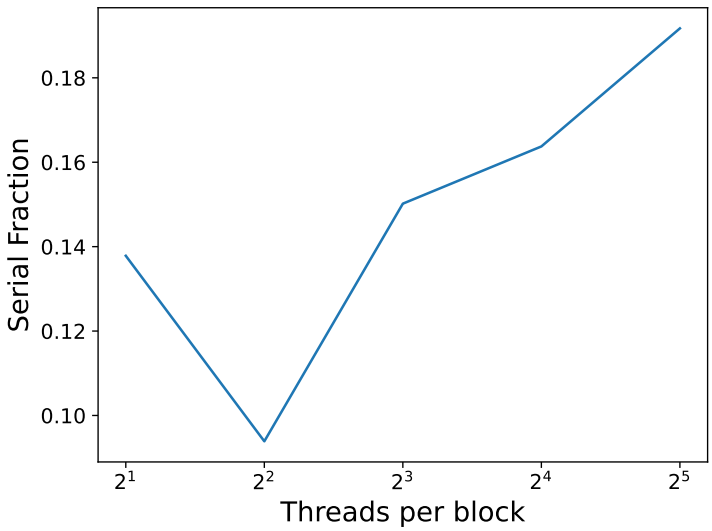

Figure 9: CUDA Serialization fraction trend