Parallelizing the Path Tracing Algorithm

Parallel Programming Project Report Group 17

Kai-Chieh Hsu
StudentID: 0816117
kaidz.cs08@nycu.edu.tw

Chia-Hao Chang
StudentID: 0716021
andychang.cs07@nycu.edu.tw

Bo-Wei Lin
StudentID: 0816152
alanlin9006.cs08@nycu.edu.tw

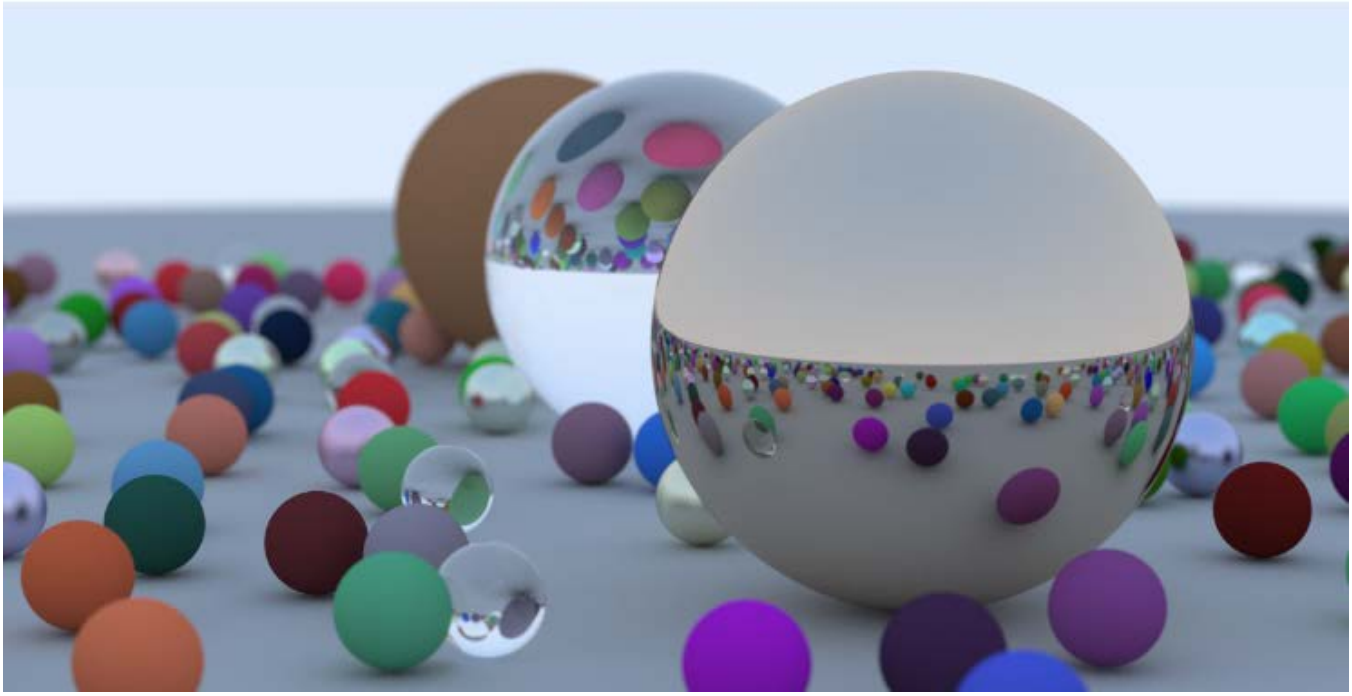

Figure 1: Ray Tracing Rendered Image

ACM Reference Format:
Kai-Chieh Hsu, Chia-Hao Chang, and Bo-Wei Lin. 2023. Parallelizing the Path Tracing Algorithm: Parallel Programming Project Report Group 17. In *Proceedings of Parallel Programming (NYCU).* ACM, New York, NY, USA, 7 pages.

1 ABSTRACT

Path tracing has been increasingly dominant in the world of computer graphics, for it having the ability of generating photorealisitic images by simulating the physics of light. However, due to its requirement on repetitive and massive computations, performance remains an issue to apply it on applications such as gaming and real-time rendering. In this project, we explore the methods on how the algorithm could be speed up by parallelization. We revise a simple C++ ray tracer from [3] with OpenMP and CUDA, and obtain a 3.2x and 15.6x speed up respectively. Extensive experiments are conducted to evaluate the effectiveness of our proposed solution.

NYCU, Parallel Programming 2022 Fall, Hsinchu, Taiwan

2 INTRODUCTION

In computer graphics, ray tracing is a method to generate photorealistic 3D images. It derives a pixel-based rendering approach that generally models light transport, and is capable of simulating optical effects such as reflection, refraction, soft shadows, and scattering. This method ultimately solves the global illumination issue, which is complex and hard to realize in traditional object-based rendering approach. Owing to its outstanding and high fidelity results, ray tracing is widely used in generating still images and film-making, and especially useful for applications which require vivid and delicate scenes.

However, efficiency is the main obstacle to apply the algorithm to real-time applications where rendering speed is critical. To render a frame, a ray tracing algorithm might take minutes or hours to complete because it requires to simulate the propagation of light and perform massive mathematical computations to detect ray-object intersection. The more objects in a scene, the more computation there is. To generate an image with higher quaility, more computations also have to be done.

In this project, we explore different ways to exploit *parallelism* to speed up the ray tracing process. Since generating an image

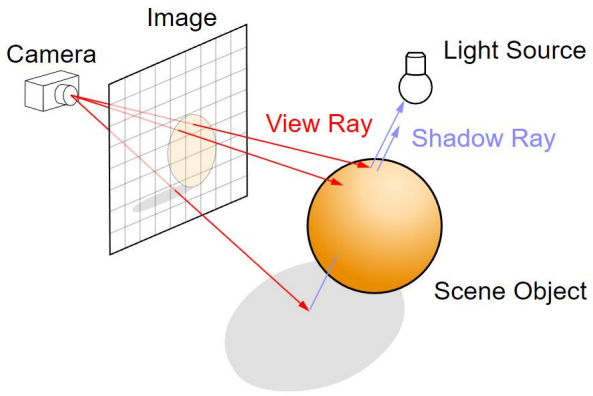

Figure 2: A diagram of ray tracing and image generation. The color value of each pixel in an image is approximated by extending rays from the viewer (camera) into the scene, then bouncing them off surfaces and approaching the light source. This image is cited from [1].

with ray tracing algorithms requires repeated computation and numerous loops, we argue that parallelism is a descent way to speed up. We focus on a specific ray tracing method, namely ***path tracing***, and analyze the bottlenecks that affect the performance.

We first follow the book *Ray Tracing in One Weekend* [3] to implement a serial path tracer from scratch using C++. Then, we analyze potential sections that could be parallelized. Two methods of parallelization are used, multithreading (OpenMP) and GPU acceleration (CUDA). Through analysis, we realized that parallelizing the rendering of each individual pixel is the most effective method. We conduct extensive experiments and analyze the effectiveness of both parallelization methods under different input settings. Lastly, we discuss the performance challenges behind implementing a parallel path tracing algorithm.

3 STATEMENT OF THE PROBLEM

In this section, We first give a background of path tracing, then describe the components and sections that could be sped up.

Ray tracing mainly consists of 3 stages: *Ray Generation*, *Ray Intersection* and *Shading*. We describe each stage in detail in the following.

3.1 Ray Generation

Each pixel in an image corresponds to an area in the scene (as shown in the grid in figure 2). Thus for each pixel, we randomly sample s points within the area, then for each point i, we generate a ray by starting from the camera e and pointing towards the sampled point with direction $\vec{d}_i$. The generated ray can be denoted by a parametric equation 1:

$$p_i(t) = e + t \cdot \vec{d}_i \tag{1}$$

3.2 Ray Intersection

Then in the ray intersection stage, for each ray, we determine whether a ray will intersect with an object in the scene. If it does, randomly select a direction of reflection to recursively generate another ray until it will not collide with any object or it exceeds the maximum limit of the number of reflection.

Take a sphere for example, we can represent a sphere with equation 2. O is the origin (center) of the sphere, and r is the radius of the sphere. If a random point P in 3D space happens to be on the sphere, then clearly $f(P) = 0$.

$$f(x) = |(x - O)|^2 - r^2 = 0 \tag{2}$$

If a ray intersects with a sphere, it implies that equation 3 has a positive t solution.

$$f(p(t)) = |(e + t\vec{d} - O)|^2 - r^2 = 0 \tag{3}$$

Thus for each ray, we will have to loop through all objects to check whether or not it will intersect with them.

3.3 Shading

Finally, if a ray reaches a light source in the end, it will contribute to the color of the pixel on the image which it corresponds to. The color of a single ray is determined by all objects it has reached, and the color of a pixel is the average of all sampled rays. Figure 3 shows the whole process from ray generation to shading.

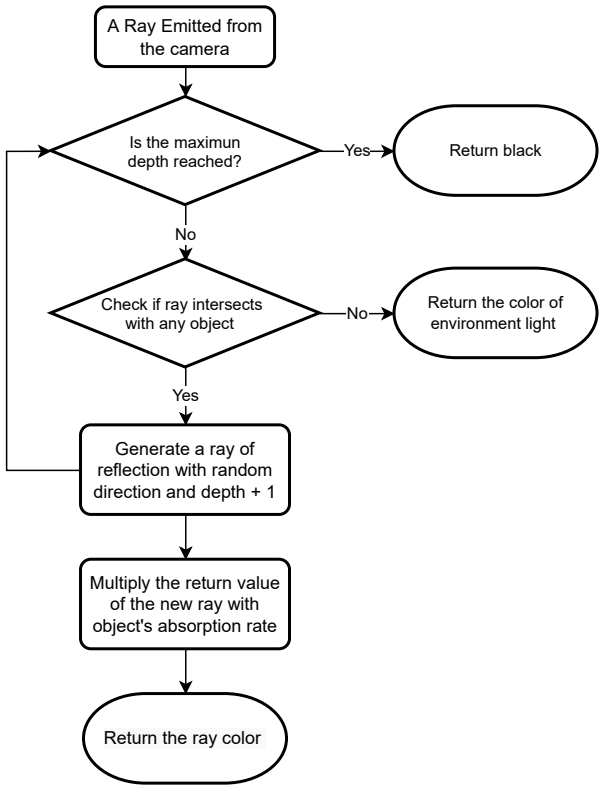

Figure 3: 3 stages flow of a sampling ray for single pixel

3.4 Problem Analysis

The time complexity of the path tracing algorithm can be determined by the following factors:

- p: number of total image pixels
- s: number of sampled rays per pixel
- o: number of objects in scene
- r: number of maximum reflections

If everything was performed sequentially, the time complexity should be: $O(p \cdot s \cdot o \cdot r)$. To render an image with higher quality and more details, we would have to increase the size of each factor. It is obvious that certain parts could be sped up.

4 PROPOSED SOLUTION

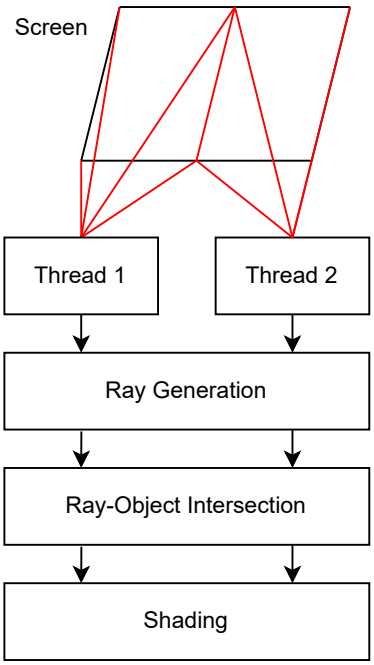

Figure 4: A Stucture of Data Parallelism Model
Computing pixels' color is mutually independent, so we can do data parallelism by assigning the task of computing pixel in leftside to thread 1, while the rightside to thread 2.

In this section, we discuss possible ways to parallelize each section of the algorithm, in other words, we discuss how to reduce the execution time of each factor in the analysis in section 3.4. Then we select the most *effective* and *viable* factor to parallelize.

By analyzing the time complexity in the section above, we can see that each factor contributes to the final execution time.

r: To reduce the execution time of factor r, we will have to parallelize the computation of each reflection. This could be completed by first allocating a thread pool. Whenever a reflection occurs, the newly spawned ray is assigned a thread in the thread pool to perform the corresponding computation, e.g., object intersection. Although this method keeps all threads busy at all times, it suffers from heavy synchronization overhead because each ray is dependent on the previous ray during reflection.

o: To reduce the execution time of o, we can parallelize the task of determining which of all objects is the nearest that would be intersected by the ray. Although we could largely reduce the overall time of the intersection detection for all objects, that is, to distribute the task to compute the intersection and distance to each thread, the additional overhead to synchronize all threads and an additional loop to determine the nearest object is still inevitable. Thus, this solution is rather infeasible.

s: The computation of each sampled ray within one pixel could be parallelized by distributing the computation of each ray to a corresponding thread. This is a relatively feasible and effective factor to parallelize compared to the previously mentioned factors.

p: This is also a common factor to parallelize since each thread would just be in charge of all computations within a single pixel. Since all pixels are mutually exclusive, no additional synchronization has to be done.

Whether to do a pixel-wise parallelization or ray-wise parallelization (parallelize factor s or p) isn't immediately obvious, therefore we conducted a simple experiment to measure the effectiveness of parallelizing each factor by OpenMP. The experiment is conducted by rendering a 300×200 image with 486 objects, 50 rays are sampled for each pixel with a maximum reflection number of 50, a sample of the scene can be seen in Figure 1. The relative speedup to the serial implementation of each thread number is listed in table 1.

Table 1: Relative speedup to serial implementation of pixel-wise and ray-wise parallelization

Threads	2	4	8
Pixel-wise	1.53×	2.63×	**3.25**×
Ray-wise	1.72×	2.71×	2.44×

We can see that as the number of threads increases, the pixel-wise parallelization method outperforms the ray-wise parallelization. We suspect that the reason why the ray-wise method becomes ineffective as the number of threads grow is that, the values computed by all rays within a single pixel has to be aggregated and averaged to form a single value (the color) for the corresponding pixel. thus resulting in an additional synchronization overhead, where threads have to wait until other threads to complete.

As a result, we decided to parallelize the rendering of each pixel, which is to reduce the execution time of factor p. The illustration of the data parallelism model is shown in figure 4. By performing data parallelism in this way, as the computation of each pixel completes, we can dynamically distribute more pixels to that thread for computation during run time, which maximizes the overall throughput.

5 EXPERIMENT METHODOLOGY

In our experiments, we compare the effectiveness and the speedup relative to the serial implementation attained by the two parallelization methods: CUDA and OpenMP under different **input settings**.

Both versions parallelize the serial code following the pixel-wise parallelization approach described in section 4.

5.1 Experiment Setting

According to the time complexity listed in section 3.4, the execution time of the path tracing algorithm is determined by four factors: Image size (pixel number), samples per pixel, maximum reflection number, and the number of objects in the scene. The larger the size of each factor, the higher the quality of the rendered result. Therefore, we came up with 4 different input settings. We first define a *basic image* with the following configuration:

- Image size: 600 × 400
- Ray samples per pixel: 50
- Number of maximum reflections: 50
- Scene: 1 (486 objects placed on the ground)

A sample of this image can be seen in the middle of figure 6. In each experiment, we modify only **one factor** (the listed factor) and keep other factors the same as the basic image, then measure the **execution time** of each program. We list each setting below:

(1) Image size:
  (a) 300x200
  (b) 600x400
  (c) 1200x800
(2) Ray samples per pixel
  (a) 10
  (b) 50
  (c) 200
(3) The number of maximum reflections
  (a) 10
  (b) 25
  (c) 50
(4) Different scene
  (a) Scene 1
  (b) Scene 2

The effect of each setting will be described in detail in section 6.

5.2 Implementation Details

The serial implementation of the ray tracer follows the book *"Ray Tracing in One Weekend"* [3]. The program is compiled with compiler flag '-O3' enabled for maximum efficiency. For the CUDA implementation, we have rewritten parts of the code containing recursion in an iterative approach, since CUDA limits recursion. For all experiments, we set the number of threads for the OpenMP implementation to 8. For CUDA, we set the block size to (8,8) and the grid size to (8,8) empirically. We noticed that the speedup efficiency is insensitive to the block size.

The experiment environment and hardware specifications are given in table 2.

Table 2: Platform

OS	Ubuntu 20.04.5 LTS
CPU	AMD® Ryzen 7 3750h with radeon vega mobile gfx × 8
GPU	GeForce GTX 1650 Mobile / Max-Q
MEM	15.4 GiB
g++	9.4.0
OpenMP	4.5
nvcc	12.0

6 EXPERIMENT RESULTS

6.1 Size

In this experiment, we render 3 images with different sizes. The results are shown in table 3. We plot the relative speedup to the serial implementation in figure 5. We can see that as the image size increases, the cuda version attains a higher speed up. This is reasonable because as the computation go large, it utilizes more GPU hardware resources, and the execution time of each pixel becomes more evenly distributed, better utilizing the hardware. On the other hand, the speed-up of the openmp implementation do not differ much as the image size increases.

Table 3: Performance over different image size

Size	-	serial	openmp	cuda
300x200	Time(s)	17.7	6.50	1.48
	Speedup	1.0	2.72	11.96
600x400	Time(s)	70.42	21.77	4.83
	speedup	1.0	3.23	14.58
1200x800	Time(s)	285.28	90.1	18.33
	Speedup	1.0	3.17	15.56

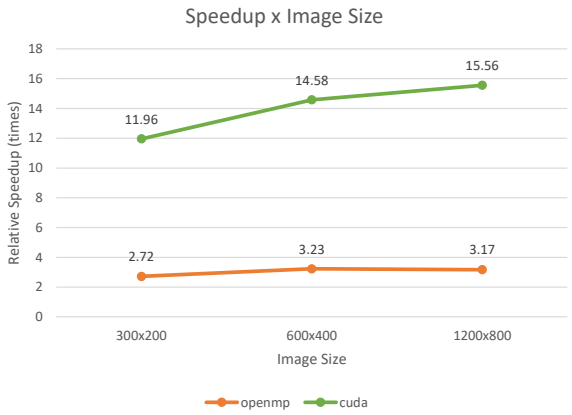

Figure 5: The relative speedup to serial implementation of different image sizes

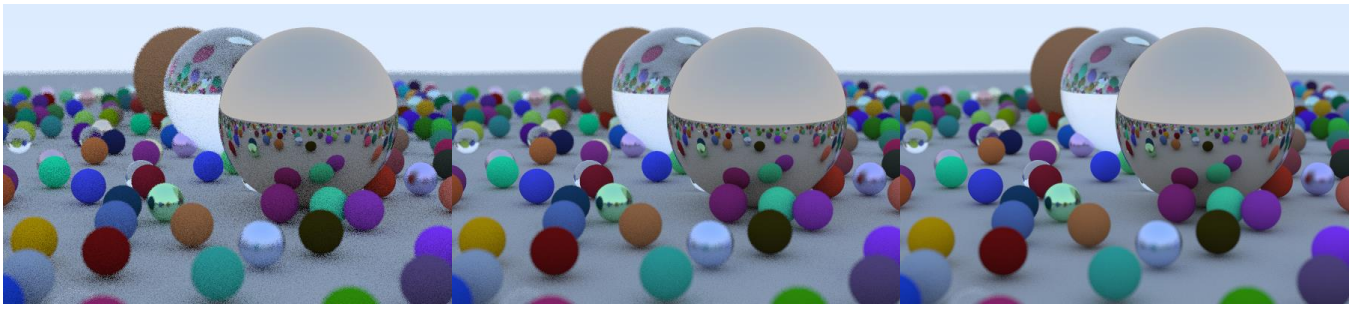

Figure 6: Rendered Image with Different Number of Samples per Pixel
Left: 10 samples per pixel. **Middle:** 50 samples per pixel. **Right:** 200 samples per pixel.

6.2 Sampled Rays per Pixel

For this experiment, we sample different numbers of rays per pixel for each image. If the number of samples per pixel is low, the resulting image will feature a lot of noise as shown in figure 6. Thus, higher sample rate is usually more desirable in real-world application. For the different number of sample rays per pixel, the following table 4 shows the result, we plot the speedup on figure 7. Although the images differ greatly, the results and trends here of the speed-up do not differ much from the experiment in section 6.1, which increases the image size. We believe that this is because adding samples per pixel is actually the same as adding the number of pixels in terms of computation.

Table 4: Performance over different samples per pixel

Samples	-	serial	openmp	cuda
10 samples	Time(s)	14.17	4.58	1.03
	Speedup	1.0	3.09	13.76
50 samples	Time(s)	70.33	21.71	4.85
	Speedup	1.0	3.24	14.5
200 samples	Time(s)	281.75	87.92	19.06
	Speedup	1.0	3.2	14.78

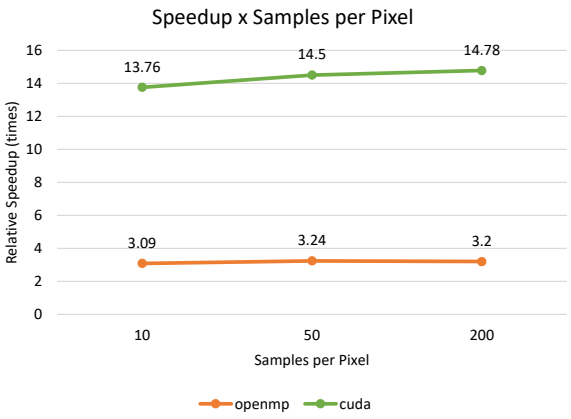

Figure 7: Speedup in different samples per pixel

6.3 Maximum Reflection

Interestingly, when we decrease the maximum reflection depth, there is almost no performance difference on the serial and openmp implementation, however, the cuda implementation gets significant improvement. To explain the phenomenon, we can recall the SIMT model behind the GPU architecture. Suppose there are 32 threads in a warp, each thread is responsible for calculating the full path of a ray bouncing in the scene. If the number of reflection times varies a lot between rays, all the other threads in a same warp must stall until the thread computing the ray having most reflection times complete, which leads to poor SIMT efficiency. Decreasing the number of maximum reflection times eases the situation.

Table 5: Performance over different maximum reflection

max depth	-	serial	openmp	cuda
10	Time(s)	68.97	24.37	2.38
	Speedup	1.0	2.83	28.98
25	Time(s)	71.30	23.69	3.53
	Speedup	1.0	3.01	20.21
50	Time(s)	70.72	22.17	4.84
	Speedup	1.0	3.19	14.61

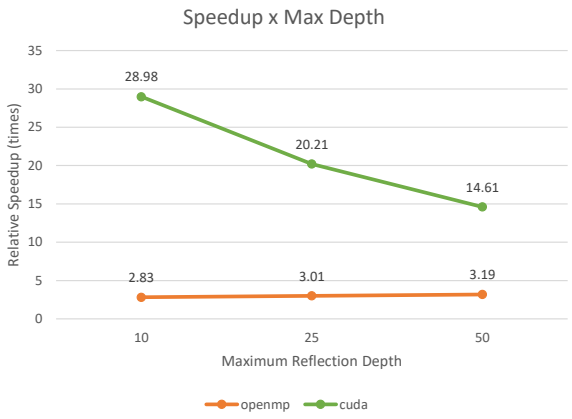

Figure 8: Speedup in different maximum reflection depth

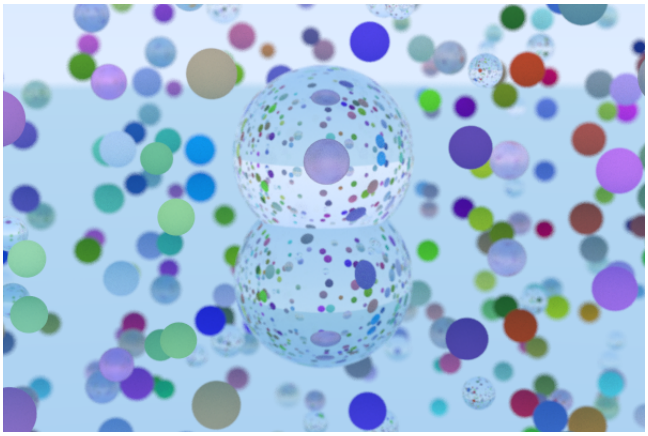

Figure 9: An alternate scene for rendering.

6.4 Different scene

In this experiment, we would like to verify whether the speedups of the two methods are also consistent in other scenes. Thus, we created a more diverse scene that contains roughly the same number of objects, but each object floating in the air, increasing the chance of reflection. This scene can be seen in figure 9. Table 6 shows the experiment result on different scenes, the speedup relative to the serial implementation is plotted on figure 10. We can see that the speedup is consistent across different scenes for either openmp or cuda.

Table 6: Performance over different scene

view	-	serial	openmp	cuda
1	Time	70.72	22.17	4.85
	Speedup	1.0	3.19	14.61
2	Time	71.51	21.77	3.94
	Speedup	1.0	3.28	18.15

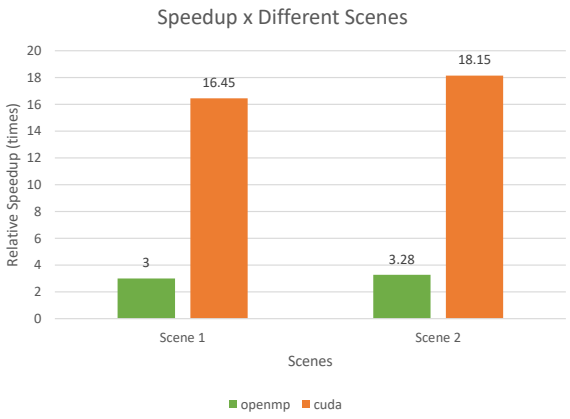

Figure 10: Speedup in different scenes

6.5 Discussion: branch divergence in path tracing

Refer to experiment results above, we can observe that the CUDA implementation only attains a speedup of 10 to 20 times instead of a hundred time speedup. Why is this the case? We suppose the main reason is branch divergence. There are two factors that lead to branch divergence in path tracing. As mentioned above, there could be different numbers of ray reflection times in a same warp.

The other one is that those rays in a same warp might hit objects with different textures. Light interact differently with different textures, which would result in many different branches in the compiled code. For example, when light hits rough metal, the diffuse effect is much more significant. While light hits shiny glass, the perfect reflection and refraction appear. To simulate those behaviors of light, threads have to execute different code, and as the number of textures increase largely, the efficiency of the SIMT model then decreases.

7 RELATED WORK

Ray Tracing in One Weekend [3] introduces the concept to ray tracing (path tracing) and provides tutorials on how to implement serial path tracing from scratch using C++.

ToyRTX [4] converts ray tracing in one weekend [3] into openGL, and uses compute shader program to realize path tracing on GPU. It also implements triangle meshes and bounding volume hierarchy. However, this implementation is an approximation to path tracing and makes it difficult to profile the overall performance of the parallelized section.

GPU Acceleration of Ray Tracing with CUDA [2] perform detail performance analyse with Nvidia performance tools. They tests how different optimization method effect performance. These method are double and float, recursive to iteration and block size optimization. They also indicates that memory bandwidth is not the bottleneck of performance. Compare it to our work, it only test with different samples ray per pixel, but we also do iteration and float optimization.

8 CONCLUSIONS

8.1 Contribution

Our main contribution is figuring out a suitable method to parallelize path tracing algorithm, and then implement two parallelized versions by openMP and CUDA, with different experiment parameter set. As the computation workload grows, the speed up of the parallel version becomes more significance.

Branch divergence has important effect on GPU architecture, we talk about how different material and the numbers of reflection times affect the performance of cuda. We prove the latter through experiment of changing the number of maximum reflection times.

8.2 Future work

Bounding Volume hierarchy (BVH) is a common method to speed up ray-object detection in an algorithmic aspect of the ray tracing algorithm, but it also increase the complexity of the structure of the program flow, leading to more branch divergence. We might

implement BVH in our ray tracer in the future, and profile its performance on the GPU architecture.

A second extension is to implement path tracing of more complex objects. Currently, there are only spheres in our path tracer. To construct more complex and general scenes, we would like to add triangle objects and implement ray-triangle collision algorithms in our future work.

REFERENCES

[1] Henrik. 2008. Ray Tracing Diagram. https://commons.wikimedia.org/wiki/File:Ray_trace_diagram.svg.
[2] navining. 2020. GPU Acceleration of Ray Tracing with CUDA. https://github.com/navining/cuda-raytracing.
[3] Peter Shirley. 2020. Ray Tracing in One Weekend. https://raytracing.github.io/books/RayTracingInOneWeekend.html
[4] Ubpa. 2020. ToyRTX. https://github.com/Ubpa/ToyRTX.