Parallel Ray Tracer

周睦鈞 311553060
曾偉杰 109550156
黃彥傑 109550059

ABSTRACT

This study explores parallelization techniques to enhance the performance of a ray tracing renderer, a computationally intensive task in computer graphics. The renderer, initially serial, is parallelized using SIMD, OpenMP, MPI, and CUDA. The impact of these methods is evaluated through experiments varying image size, samples per pixel, and bounces per ray. Results show that CUDA consistently outperforms CPU methods, with the split kernel approach demonstrating the highest speedup. SIMD and MPI also exhibit significant improvements, emphasizing the effectiveness of parallel computing in optimizing ray tracing workflows. The findings contribute insights into optimizing ray tracing for real-time applications and high-quality rendering.

1 INTRODUCTION

Ray tracing is a rendering technique used in computer graphics to simulate the way light interacts with objects in a virtual scene. Its working principle is to generate several rays of light for each pixel of the image and simulate the effect of its encounter with various objects. This process helps create highly realistic and visually accurate images by accurately depicting light, shadows, reflections and other visual phenomena. By mimicking the behavior of light in the physical world, ray tracing can produce images with unparalleled detail and realism.

One of the key characteristics of ray tracing is its high computational complexity. This is primarily because the technique involves tracing the path of rays as it interacts with objects in the scene, and simulating the complex interactions of light with various surfaces, materials, and atmospheric effects. Therefore, rendering high-quality imagery using ray tracing can be computationally intensive and time-consuming, especially for complex scenes and high-resolution output.

Fortunately, ray tracing demonstrates a remarkable degree of parallelism, often described as an "embarrassingly parallel" problem. To harness this, parallel computing techniques are often employed. These techniques involve breaking down the rendering process into smaller tasks that can be handled simultaneously by multiple processors, computing cores, or arithmetic units. Parallelization can significantly accelerate the rendering process, resulting in producing high-quality images faster and providing real-time ray tracing for applications such as video games, virtual reality, and architectural visualization, making them more suitable for a variety of applications.

In this course we have developed a simple ray tracing renderer and apply various parallelization methods to accelerate the renderer. We will delve into the utilization of various parallelization methodologies, including SIMD intrinsics, OpenMP, MPI, and CUDA, in order to assess the efficacy of these approaches in enhancing rendering performance. Our focus will be on evaluating the extent of speedup and efficiency gained by the integration of these parallel computing techniques in the context of ray tracing.

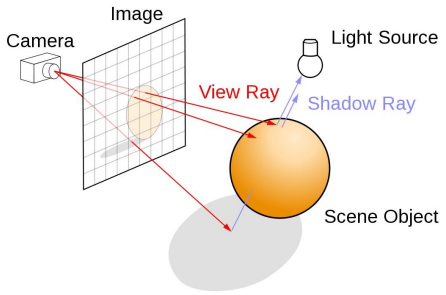

Figure 1: Conceptual Workflow of Ray Tracing

2 STATEMENT OF THE PROBLEM

Figure 1 shows the conceptual workflow of ray tracing. It involves simulating the interaction of light with objects in a virtual scene, with each object comprising geometrical primitives such as triangles and spheres. The workflow of ray tracing typically involves the following steps:

(1) **Ray Generation**: Rays are generated from the virtual camera, propagating through each pixel in the image plane, and cast into the scene.
(2) **Intersection Calculation**: The generated rays are tested for intersections with the geometrical primitives representing the objects in the scene.
(3) **Shading Calculation**: Once an intersection is found, the shading of the intersected point is computed, taking into account the surface properties, light sources, and various lighting models.
(4) **Recursive Tracing**: If the intersected object has reflective or refractive properties, secondary rays are generated and traced recursively.
(5) **Pixel Accumulation**: The results of these calculations for each ray contribute to the final color of the pixel on the image.

The workflow exhibits substantial parallelism throughout its various stages. Notably, in the ray generation phase, multiple rays are generated for each pixel, highlighting the inherent parallel nature of this step. Similarly, in the intersection calculation stage, each ray's intersection point is determined by solving specific equations (e.g., quadratic equations for spheres). Since the computational steps for solving these equations are identical for every ray, they can be effectively parallelized using SIMD/SPMD techniques. Leveraging such parallelization techniques can significantly enhance the overall efficiency and speed of the ray tracing process, significantly reducing the time required for rendering high-quality images. The subsequent subsection formulates the ray tracing process and analyzes the possibilities for parallelization.

Algorithm 1: The Conceptual Workflow of Ray Tracing

```
for p = 1 to P do
    Image[p] = [0 0 0]
    for s = 1 to S do
        Ray = GenerateRay(p)
        for b = 1 to B do
            Initialize IntersectionResult
            for g = 1 to G do
                IntersectionResult = IntersectionTesting(Ray, GeometricObjects[g], IntersectionResult)
            if IntersectionResult.Hit then
                Image[p] = Image[p] + CalculateColor(LightSource, Ray, IntersectionResult)
                Ray = GenerateNextRay(Ray, IntersectionResult)
            else
                break
    Image[p] = Image[p] / S
```

2.1 Parallelism Possibilities

The conceptual workflow of ray tracing, described in Algorithm 1, involves five nested `for` loops, each with dimensions P, S, B, G, and C. These dimensions represent the number of pixels, sampled rays per pixel, bounces per ray, geometric objects, and vector components, respectively.

Although not explicitly present in the algorithm, Loop C pertains to the vector's three components. For example, when adding two vectors, the x, y, z components needed to be added one by one in serial implementation.

The overall complexity of ray tracing can be approximated by the product of these loop dimensions, i.e., $O(P \cdot S \cdot B \cdot G \cdot C)$. The following paragraph outlines the possibilities for parallelizing these factors.

- **Parallizing loop P**: Each thread computes a single pixel independently, requiring no synchronization due to the absence of inter-pixel dependencies.
- **Parallizing loop S**: Each thread computes one sample, necessitating synchronization when multiple threads write results to the image frame buffer for the same pixel.
- **Parallizing loop B**: Parallelizing this loop is deemed infeasible due to dependencies between generated rays. For instance, generating a reflected ray relies on determining the camera ray and performing intersection tests.
- **Parallizing loop G**: Parallelizing this loop is also considered infeasible since synchronization is required for calculating the closest intersection point of a ray with different geometric objects. The potential overhead outweighs performance gains.
- **Parallizing loop C**: Each compute unit independently processes one operation per component of a vector, implementable with SIMD intrinsics without any synchronization overhead. Parallelizing this loop proves beneficial, considering that in ray tracing, many data elements can be conveniently represented as vectors of three components. Instances include the RGB color representation of a pixel, the direction of a ray, and the coordinates of a point.

Given the high synchronization overhead associated with parallelizing Loops B and G, these loops are left unparallelized. Further details on the proposed parallelization strategies for Loops P, S, and C are elaborated in Section 4.

3 THE RAY TRACING RENDERER

For this project, we have developed a ray tracing renderer, initially lacking any parallelization techniques. All subsequent parallelized versions, as detailed in Section 4, are derived from this original renderer. This renderer includes the fundamental functionalities of a ray tracer, supporting common primitives such as spheres and triangles. Additionally, the renderer accommodates at most one point light source, which serves as the illumination for the scene. The rendering process adheres to the rendering equation [1], employing path tracing and Monte Carlo method to approximate the integral.

Written entirely in pure C++, the renderer operates without the need for external libraries. This deliberate choice was made to facilitate the parallelization of the code, eliminating concerns about interfacing with external libraries like OpenGL and modifying their implementations. With a succinct codebase comprising approximately 450 lines, the simplicity of the renderer allows for ease of parallelization.

The rendered image is showcased in Figure 2. Notably, users have the flexibility to alter the scene by directly modifying the camera coordinates, primitive specifications, or light source parameters within the codebase. This design choice enhances the renderer's versatility and makes it adaptable to a range of scenarios.

4 PROPOSED SOLUTION

4.1 SIMD

The SIMD method emphasizes parallel vector computation of floating point values through arithmetic operations, including addition, multiplication, division, inner product, and outer product. The process involves placing the floating-point values into a 128-bit register with an additional padding value.

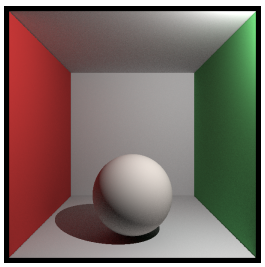

Figure 2: The Rendered Image

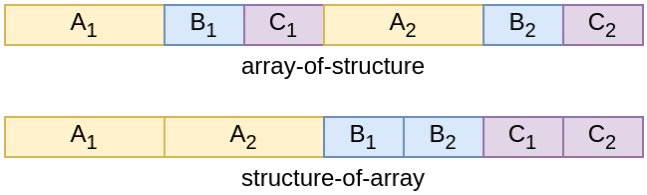

Figure 3: Array-of-Structure vs. Structure-of-Array

4.2 OpenMP

The OpenMP method employs dynamic scheduling to distribute image rows among multiple threads. This approach allows the compiler to automatically allocate threads, assigning each one a specific set of image rows to process.

4.3 MPI

The MPI implementation involves allocating specific segments of an image to individual MPI nodes, enabling parallel processing across different servers. This approach closely resembles the concept of OpenMP, where tasks are distributed among threads, but in the MPI version, the workload is distributed across distinct server nodes.

4.4 CUDA

We have developed two versions of CUDA implementations to accelerate ray tracing. In both versions, each thread is assigned one ray to work with. The distinction lies in their approach to computation. The first version consolidates most computations into a single "mega kernel," while the second version distributes computations across several kernels. The rationale behind splitting the kernel arises from the observation that directly porting a large CPU program into an equally large GPU kernel incurs significant performance penalties, as discussed in [2]. Large kernels may suffer from high register usage, reducing the occupancy and diminishing the latency-hiding capability which is crucial for GPU's high-latency, high-bandwidth memory system.

Additionally, coalescing memory access is essential for overall performance due to the GPU's memory architecture. Therefore, we have optimized memory access patterns for both mega and split kernel implementations, details of which are explained in subsection 4.4.1. Further insights into the "mega kernel" and "split kernel" implementations are provided in subsections 4.4.2 and 4.4.3.

4.4.1 Coalescing Memory Access. The GPU's memory architecture incorporates a coalescing unit, making coalesced memory access significantly faster than uncoalesced access. In our CUDA ray tracer implementation, we address this issue by investigating memory access patterns using the NVIDIA Visual Profiler. The profiling report emphasizes that the ray tracer's performance is primarily constrained by memory operations, underscoring the importance of enhancing memory access patterns.

Our initial implementation suffered from low memory bandwidth utilization due to uncoalesced access patterns when accessing arrays of large structures. We resolved this issue by transforming the array-of-structure layout into a structure-of-array layout, resulting in coalesced access patterns and improved performance. Figure 3 illustrates the memory layouts of array-of-structure and structure-of-array. In our CUDA implementation, after employing the structure-of-array layout, we achieve a 1.09x speedup compared to the array-of-structure layout when rendering Figure 2.

4.4.2 Mega Kernel. This version simply wraps the work inside loop P and loop S into a single "mega" kernel. Each CUDA thread is responsible for one sampled ray, and upon completing the color computation, the result is written back to the image's frame buffer.

4.4.3 Split Kernel. To address the challenges outlined at the beginning of this section, we divide the mega kernel into several smaller kernels. The mega kernel is split into five kernels named *CHECK*, *PROCESS*, *GEN*, *AHIT*, and *CHIT*. Each thread of a kernel retrieve one rays from a large ray pool. The functionalities of these kernels are as follows:

- **CHECK**: Examines the ray's state, terminates rays exceeding their maximum recursion depth, and directs ray IDs to *PROCESS* or *GEN*.
- **PROCESS**: Calculates the color of the ray by the intersection result, and generates the next ray. Ray IDs are directed to *AHIT* or *CHIT*.
- **GEN**: Generates new rays from the camera and directs ray IDs to *CHIT*.
- **AHIT**: Performs intersection tests to determine if the ray hits anything in the scene. If nothing is hit, write the resulting color to the frame buffer.
- **CHIT**: Performs intersection tests and finds the closest hit point.

The flow of rays through each kernel is depicted in Figure 4. Different kernels communicate by sending ray IDs into the queue of the destined kernel. The kernel then uses the ray ID to index into the ray pool and fetch the corresponding ray.

After implementing the split kernel method, we observed a decrease in register usage, as detailed in Table 1. The reduced register usage enables more CUDA warps to execute simultaneously on a Stream Multiprocessor (SM), enhancing latency hiding capabilities on the GPU.

4.5 Comparison of Different Parallelization Methods

This subsection provides a summary of various parallelization methods by analyzing the loops they parallelize in the context of ray

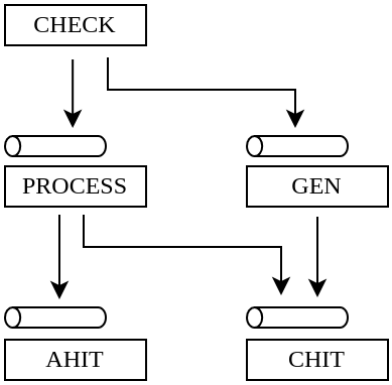

Figure 4: The Flow of The Ray. ***(The kernel is depicted as a rectangular box and the queue is depicted above the kernel.)***

Table 1: Register Usage of Different Kernel

Kernel		Register Usage
Mega		65
Split	CHECK	14
	PROCESS	39
	GEN	28
	AHIT	32
	PHIT	43

tracing. As outlined in 2.1, the ray tracing workflow can be decomposed into five loops denoted by P, S, B, G, and C. Table 2 outlines the parallelized loops for each method, highlighting the distinctive features of each.

Table 2: Parallelized Loops of Different Methods

Method	Parallized Loop
SIMD	C
OpenMP	P
MPI	P
CUDA	P, S

Notably, the SIMD implementation parallelizes the C loop, which is different from the parallelized loops of OpenMP and MPI. Consequently, SIMD can be seamlessly combined with OpenMP and MPI, though it is incompatible with CUDA due to the absence of SIMD intrinsics for computing multiple FP32 operations simultaneously. The performance of combining SIMD with OpenMP and MPI will be further elaborated in Section 6.

5 EXPERIMENTAL METHODOLOGY

Based on the computational complexity outlined in Section 2.1, the ray tracking's execution time is influenced by three factors: **Image size (number of pixels), Samples per pixel, Maximum bounces per Ray**. A larger value for each factor corresponds to a higher-quality rendered result. Therefore, we came up with 3 different input settings. We first define a basic image with the following configuration:

- Image Size: 600 x 600
- Ray samples per pixel: 512
- Maximum bounces per Ray: 4

A sample of this image is depicted in Figure 2. In each experiment, we altered only **one factor** while keeping the other factors consistent with the baseline image. Subsequently, we measured the execution time of each program. The specific settings for each experiment are enumerated below:

(1) Image Size
  (a) 150 x 150
  (b) 300 x 300
  (c) 600 x 600
(2) Samples per Pixel
  (a) 256
  (b) 512
  (c) 1024
(3) Maximum bounces per Ray
  (a) 2
  (b) 4
  (c) 8

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

Table 3: Platform

OS	Ubuntu 20.04.2
CPU	Intel(R) Core(TM) i5-7500 @ 3.4GHz
GPU	NVIDIA Tesla T4
RAM	12GB
GCC	9.4.0
NVCC	10.1.243
Open MPI	4.0.3
CUDA	12.2
OpenMP	4.5

6 EXPERIMENTAL RESULTS

6.1 Basic setting

In the initial experiment conducted under the basic setting, the results are in Table 4, the speedup achieved with SIMD was only marginally greater compared to their original methods, and the difference was not statistically significant. This suggests that the program's primary bottleneck may not be related to vector computation, as the observed benefits were not apparent.

Moving on to the comparison between MPI and OpenMPI, the reason for MPI exhibiting a higher speedup can be attributed to its ability to leverage more processors for parallel computations. The increased number of processors enhances the parallelization capabilities, contributing to the overall speedup compared to OpenMPI.

It is noteworthy that both CUDA methods consistently outperformed any other CPU methods in our experiment. This superior performance can be attributed to the substantial number of computing units available on the GPU, making it better suited for the computational demands of the task at hand.

Table 4: Execution Time and Speedup of Different Methods Based on the Basic Setting *(SIMD: ✓means allowed to use vector computation; ✗means not allowed.)*

Implementation	SIMD	Time (s)	Speedup
Serial	✗	102.46	1.00x
	✓	95.00	1.08x
OpenMP*	✗	39.65	2.58x
	✓	38.90	2.63x
MPI†	✗	15.75	6.51x
	✓	15.65	6.55x
CUDA (Mega)	✗	6.64	15.43x
CUDA (Split)	✗	3.12	32.84x

*4 threads †8 processes

6.2 Varying Image Size

In this experiment, three images of varying sizes were rendered, and the outcomes are presented in Table 5. Notably, as the image size expands, the CUDA version achieves a greater speedup. This observation is rationalized by the fact that with larger computations, it leverages more GPU hardware resources, leading to a more evenly distributed execution time for each pixel and thereby optimizing hardware utilization. In contrast, the speedup of the OpenMP and MPI implementation remains relatively consistent, exhibiting minimal variation with increasing image size.

Table 5: Execution Time and Speedup of Different Methods over Different Image Size

	150 x 150		300 x 300		600 x 600	
	Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup
Serial	5.99	1.00x	25.08	1.00x	95.00	1.00x
OpenMP*	2.98	2.01x	9.70	2.58x	38.90	2.44x
MPI†	3.52	1.70x	4.59	5.46x	15.65	6.55x
CUDA	0.39	15.36x	0.92	27.26x	3.12	30.45x

*4 threads †8 processes

6.3 Varying Number of Samples

In this experiment, we varied the number of sampled rays per pixel for each image. Actually, when the number of samples per pixel is low, the resulting image exhibits considerable noise. Consequently, a higher sample rate is typically more desirable in real-world applications. Table 6 presents the results for different numbers of sample rays per pixel.

Table 6: Execution Time and Speedup of Different Methods over Different Samples Per Pixel

	256		512		1024	
	Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup
Serial	47.43	1.00x	95.00	1.00x	189.46	1.00x
OpenMP*	24.22	1.96x	38.90	2.44x	77.79	2.44x
MPI†	8.32	5.7x	15.65	6.55x	30.29	6.25x
CUDA	1.64	28.92x	3.12	30.25x	6.02	31.47x

*4 threads †8 processes

6.4 Varying Number of Bounces per Ray

In the experiment, we altered the number of bounces per ray per pixel for each image, the result is showed in Table 7. Similar to the observations in Section 6.2 and 6.3, the trend of speedup variation remained consistent across different methods. Notably, the CUDA version consistently exhibited a greater speedup compared to other implementations. Meanwhile, the speedup of the OpenMP and MPI implementations remained relatively stable throughout the experiment.

Table 7: Execution Time and Speedup of Different Methods over Different Bounces Per Ray

	2		4		8	
	Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup
Serial	49.78	1.00x	95.00	1.00x	146.39	1.00x
OpenMP*	21.83	2.28x	38.90	2.44x	60.15	2.43x
MPI†	7.94	6.27x	15.65	6.55x	24.82	5.90x
CUDA	2.21	22.52x	3.12	30.45x	4.02	36.42x

*4 threads †8 processes

7 RELATED WORK

- **Embree [5]**: Performance-oriented ray tracing kernels developed by Intel for efficient rendering of complex scenes. It is also optimized for the latest x86 processors with support for SSE, AVX, AVX2, and AVX-512 instructions. Notably, Embree diverges from our SIMD implementations by choosing to parallelize the entire algorithm rather than just the `vec3` datatype. For instance, during triangle intersection computations, Embree employs SIMD intrinsics to concurrently intersect multiple triangles, showcasing a distinctive approach to achieving parallelism in the context of ray tracing.
- **pbrt-v4 [4]**: An open source physically based renderer, offering a comprehensive framework for ray tracing. Distinguishing itself from our implementation, pbrt-v4 leverages both CUDA and OptiX for accelerated ray tracing, with a notable emphasis on GPU hardware acceleration using the RT Core. This entails a dependency on hardware support for efficient intersection calculations. Notably, pbrt-v4 adopts wavefront rendering to partition the rendering kernel into distinct stages, a methodology akin to our approach. The incorporation of OptiX underscores their commitment to optimizing the intersection process through GPU acceleration, a feature absent in our CUDA-exclusive implementation.

- **OptiX [3]**: A high-performance, programmable ray tracing framework developed by NVIDIA for accelerating the rendering of complex visual scenes. OptiX stands out as a versatile tool that allows developers to harness the power of GPUs for efficient ray tracing. While it serves a similar purpose to our CUDA implementation, it is a broader framework that provides a range of functionalities for real-time ray tracing, making it a valuable resource for applications demanding high-performance rendering.

8 CONCLUSIONS

In conclusion, our study focused on parallelizing a ray tracing renderer using SIMD, OpenMP, MPI, and CUDA. We conducted experiments varying image size, samples per pixel, and bounces per ray to evaluate the impact of these parallelization methods. Results revealed that CUDA consistently outperformed CPU methods, with a split kernel approach demonstrating the highest speedup. SIMD and MPI also exhibited significant improvements. Overall, our findings emphasize the effectiveness of parallel computing in optimizing ray tracing workflows, contributing valuable insights for real-time applications and high-quality rendering.

REFERENCES

[1] James T Kajiya. 1986. The rendering equation. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*. 143–150.

[2] Samuli Laine, Tero Karras, and Timo Aila. 2013. Megakernels considered harmful: Wavefront path tracing on GPUs. In *Proceedings of the 5th High-Performance Graphics Conference*. 137–143.

[3] Steven G Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, et al. 2010. Optix: a general purpose ray tracing engine. *Acm transactions on graphics (tog)* 29, 4 (2010), 1–13.

[4] Matt Pharr, Wenzel Jakob, and Greg Humphreys. 2023. *Physically based rendering: From theory to implementation*. MIT Press.

[5] Ingo Wald, Sven Woop, Carsten Benthin, Gregory S Johnson, and Manfred Ernst. 2014. Embree: a kernel framework for efficient CPU ray tracing. *ACM Transactions on Graphics (TOG)* 33, 4 (2014), 1–8.