Parallelized Ray Tracing In One Weekend

Yao-Te Ying*
leaf.ying.cs11@nycu.edu.tw
Institute of Multimedia Engineering in National Yang Ming Chiao Tung University
Hsinchu, R.O.C.

Ying-Ting Lai*
ytlai.cs12@nycu.edu.tw
Institute of Computer Science and Engineering in National Yang Ming Chiao Tung University
Hsinchu, R.O.C.

Chun-Hung Wu*
wu891123.cs09@nycu.edu.tw
Department of Computer Science in National Yang Ming Chiao Tung University
Hsinchu, R.O.C.

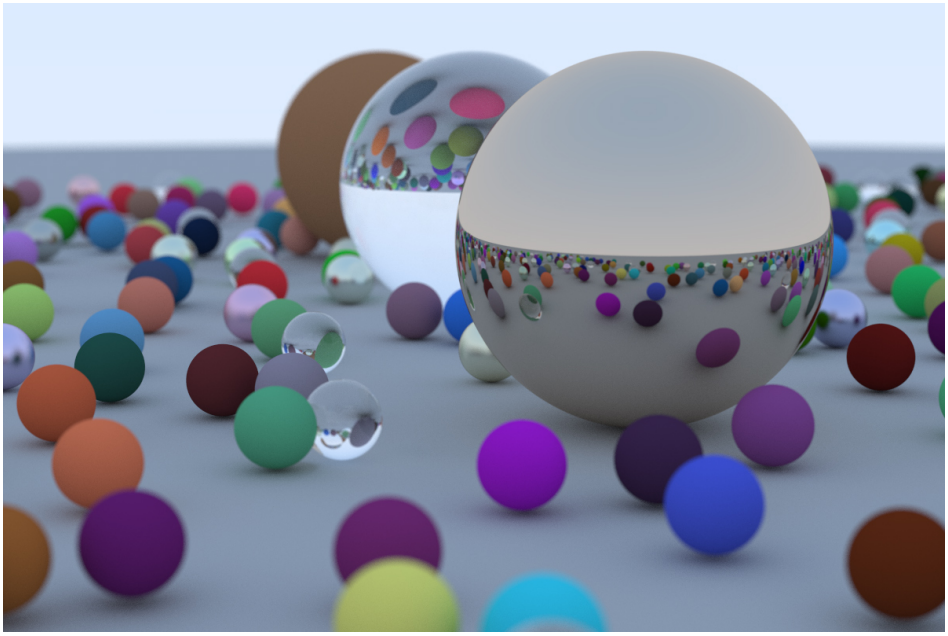

Figure 1: Final scene

CCS CONCEPTS

• **Computing methodologies** → **Parallel computing methodologies**.

KEYWORDS

OpenCL, CUDA, OpenMP, Ray Tracing, Computer Graphics, C++

1 INTRODUCTION / MOTIVATION

Ray tracing is a technique used to simulate the transport of light in digital images, and it involves computationally intensive processes. This technique is capable of modeling various optical effects, such as reflection, refraction, and scattering. In the past, its applications were limited by the significant processing time required. However, in recent years, the introduction of hardware acceleration has enabled ray tracing to be used in real-time applications, including video games. In these contexts, the optimization of ray tracing speed becomes critical, thereby opening the doors to harnessing parallelism more effectively.

In this research, we investigate the performance of ray tracing at various levels of hardware utilization. We base our analysis on the code from Ray Tracing in One Weekend [5]. We demonstrate how slow ray tracing can be when not supported by dedicated hardware, namely the Graphics Processing Unit (GPU). We will instrument all the cores of the computer to showcase the scalability of multi-threading. Additionally, we will maximize its potential by running the ray tracing processes on the GPU.

2 STATEMENT OF THE PROBLEM

To begin, we target the performance bottleneck present in the original serial code. We compiled the code using specific flags: `-O3 -DNDEBUG -pg`. Among these, `-O3`, combined with `-DNDEBUG`, represents one of the most commonly employed configurations in released executables. It triggers a substantial array of optimizations while eliminating non-production assertion overhead. This combination is particularly useful in identifying the actual hotspots within the code. The `-pg` flag is instrumental in generating additional code tailored to produce profile information compatible with the analysis tool, *gprof* [2]. In this project, we rely on gprof for

*All authors contributed equally to this research.

Table 1: Time portion spent in each function (Top 5)

Function name	Time (%)	# Calls	Self / Call (μs)	Total / Call (μs)
`sphere::hit`	91.37	1924751405	0.04	0.04
`hittable_list::hit`	6.41	558455	9.73	148.35
`lambertian::scatter`	0.55	327140	1.42	1.42
`metal::scatter`	0.09	80146	0.94	0.94
`dielectric::scatter`	0.05	45906	0.87	0.87

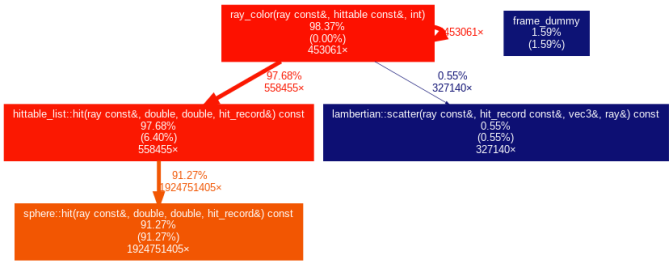

Figure 2: Call graph with gprof

profiling the ray tracing process and gaining insights into how it allocates its computational resources. Table 1 displays the profiling result obtained from an AMD Ryzen^{TM} 7 5800H CPU. We find it interesting in the following aspects:

(1) A great portion of time is spent on a single function, namely `sphere::hit`. Its cost per call is quite cheap but the number of calls is enormous.
(2) The total cost of a single call to `hittable_list::hit`, inclusive of its descendants, is approximately 15 times higher than the cost of the function itself.

These observations can be explained by examining the call graph illustrated in Figure 2 generated by gprof. In this visualization, functions with a small time proportion are omitted for clarity. A closer examination of the source code reveals the following:

- `hittable::hit` is invoked within the `hittable::hit` function.
- Each `hittable::hit` operation iterates over a list of spheres, invoking `sphere::hit` on each one.

In ray tracing, each pixel is individually calculated and rendered, resulting in a nested-loop structure. Additionally, `ray_color` utilizes recursive calls to simulate the bouncing of rays. Within each call, `hittable::hit` is responsible for finding the closest sphere intersected by the scattered ray, enabling the modeling of lighting and shading effects. In essence, `sphere::hit` is called within nested loops and recursively, making a significant contribution to the overall time consumption. Most of the work within `hittable::hit` is delegated to `sphere::hit`, its descendant, resulting in a significantly higher total cost. The flow chart for this section of serial code is depicted in Figure 3.

These two observations highlight a fundamental problem. The approach of serially looping through each pixel is inefficient, and the process of identifying the closest sphere does not necessarily require a one-by-one comparison.

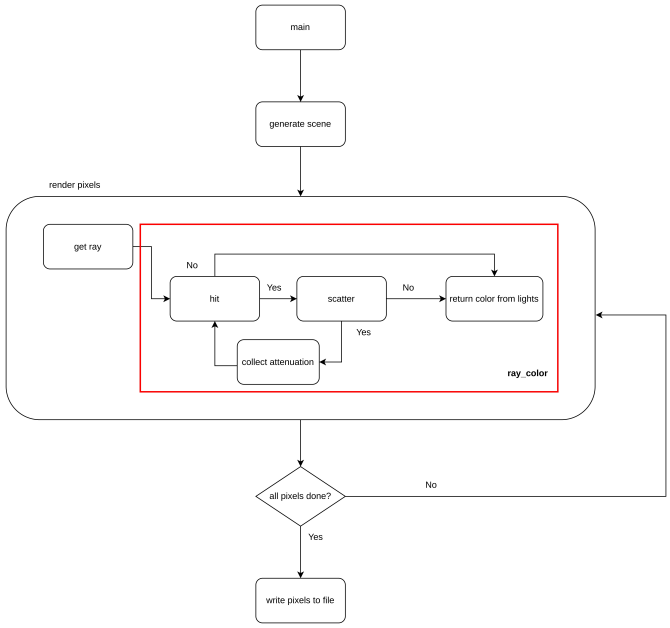

Figure 3: Serial ray tracing approach (flow chart)

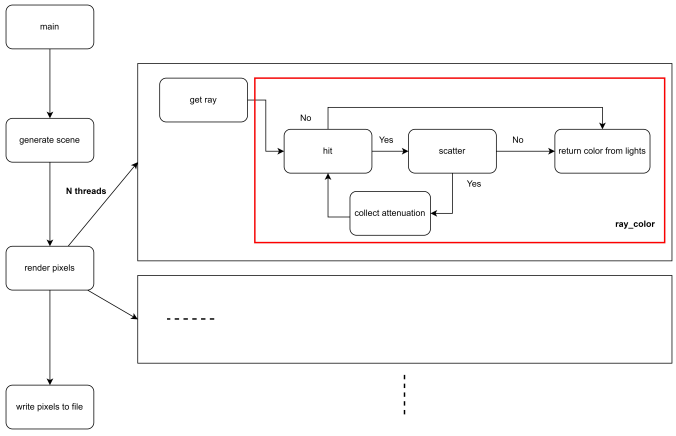

Figure 4: Proposed approach (flow chart)

3 PROPOSED APPROACHES

The Figure 4 shows the general flow of parallelized version of ray tracer.

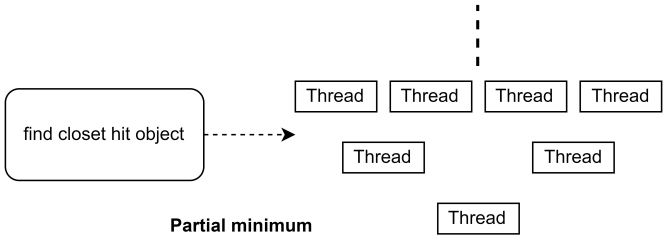

Figure 5: Hit function

3.1 Pixel rendering

Due rendering pixels is independent, we can execute it by multiple threads to achieve better performance.

3.2 Hit function

The Figure 5 shows the general flow of parallelized version of the hit function. The hit function is to find the closet hit object. So, we can use divide-and-conquer approach to split the object array into multiple chunks to find the closet hit object in parallel.

4 LANGUAGE SELECTION

4.1 CPU

We will use OpenMP[1] to parallelize the serial version of the ray tracing code. OpenMP is a high-level and easy-to-use library. It uses directives, clauses to control the parallelism.

4.2 GPU

We will use CUDA[3] and OpenCL[6] to parallelize the serial version of the ray tracing code. CUDA library controls NVIDIA's GPU to do data computation. OpenCL library is an open-source and cross-platform library supporting CPU, GPU, DSP, FPGA, etc. They are widely used in Machine Learning, Data Science, etc.

5 RELATED WORK

Several works have been done in OpenMP, CUDA, OpenCL libraries. The RTFoundation[7] project following C++ standard implements in OpenMP and CUDA. The raytracing[4] project implements in OpenCL.

6 STATEMENT OF EXPECTED RESULTS

We expect that the parallelized code will result in a performance improvement over the original code. Besides, we will calculate the image checksum to make rendering image consistency.

7 TIMETABLE

- `11/01 ~ 11/15`: Implement multi-threading with OpenMP.
- `11/01 ~ 11/22`: Utilize GPU with CUDA .
- `11/23 ~ 12/15`: Explore further parallelization opportunities with OpenCL (if time available).
- `11/16 ~ 12/31`: Analyze performance differences between serial and parallelized code.

REFERENCES

[1] L. Dagum and R. Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering* 5, 1 (1998), 46–55. https://doi.org/10.1109/99.660313

[2] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. 1982. Gprof: A Call Graph Execution Profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction* (Boston, Massachusetts, USA) *(SIGPLAN '82)*. Association for Computing Machinery, New York, NY, USA, 120–126. https://doi.org/10.1145/800230.806987

[3] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. 2008. Scalable Parallel Programming with CUDA: Is CUDA the Parallel Programming Model That Application Developers Have Been Waiting For? *Queue* 6, 2 (mar 2008), 40–53. https://doi.org/10.1145/1365490.1365500

[4] Carter Roeser. 2019. raytracing. https://github.com/cdgco/raytracing.

[5] Peter Shirley. 2020. Ray Tracing in One Weekend. https://raytracing.github.io/books/RayTracingInOneWeekend.html

[6] John E. Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science & Engineering* 12, 3 (2010), 66–73. https://doi.org/10.1109/MCSE.2010.69

[7] SeungWoo Yoo. 2021. RTFoundation. https://github.com/DveloperY0115/RTFoundation.