Empirical Evaluation of Multicore Scaling in Ray Tracing: Time, Speedup, Efficiency, and CPU Utilization

1st Pranshu Saraswat
Department of Computer Science and Engineering
M.S. Ramaiah Institute of Technology
Bangalore, India
ORCID: 0009-0009-4908-6793

2nd Sharanya Sandeep
Department of Computer Science and Engineering
M.S. Ramaiah Institute of Technology
Bangalore, India
ORCID: 0009-0006-9718-1654

***Abstract*—This paper presents a simple study on how ray tracing performance changes when using more CPU cores. A Python ray tracer was run using the multiprocessing module on an AMD Ryzen 9 5900HX processor. Tests were done at different image resolutions and with 1, 2, 4, 8, and 16 worker processes. The results show clear improvements when increasing workers up to 8, and smaller gains or slight slowdowns when using all 16 logical threads. We also examine time, speedup, efficiency, and CPU usage to understand how the workload scales on a modern multicore CPU.**

***Index Terms*—ray tracing, multiprocessing, multicore, parallel performance, Amdahl's law, speedup, efficiency**

I. Introduction

Ray tracing is a rendering technique known for producing highly realistic images by simulating the way light interacts with surfaces. However, this realism comes at the cost of significant computational demand. Each pixel may require calculating multiple light rays, reflections, and refractions, which makes the process both CPU and memory intensive.

With the growing availability of multicore processors in consumer and professional grade systems, parallel computing has become a practical solution for handling such heavy workloads. Unlike older single threaded programs, modern applications can now divide tasks across many cores, taking advantage of hardware that supports concurrency natively.

In this study, we explore how a Python based ray tracing application performs when run with multiple processes on an AMD Ryzen 9 5900HX processor a high performance CPU with 8 physical cores and 16 logical threads. By varying thread count, image resolution, and workload distribution, we aim to understand how well the application scales in real world conditions. Instead of focusing purely on theoretical models, our goal is to capture practical outcomes, such as CPU usage patterns, performance bottlenecks, and trade offs associated with parallel execution.

II. Background

We briefly review performance metrics and multicore considerations.

A. Amdahl's Law

Amdahl's Law bounds the theoretical speedup $S(p)$ as:

$$S(p) = \frac{1}{(1-f) + \frac{f}{p}} \tag{1}$$

where f is the fraction of the program that is parallelizable and p the number of processors. Empirical results deviate from ideal due to overheads such as scheduling, synchronization, and cache effects.

B. Metrics

We use:

- **Execution time** $T(p)$: wall clock time for a run with p workers.
- **Speedup** $S(p) = T(1)/T(p)$.
- **Parallel efficiency** $E(p) = S(p)/p$.

III. Methodology

The ray tracer is written in Python and uses the multiprocessing module to create multiple worker processes. Each process receives tiles of the final image to render. This approach avoids the Python GIL and allows real parallel execution on a multicore CPU. All workers run the same rendering function and report their results back to the main process.The scene used for all tests consisted of three spheres of different colors (blue, green, and yellow) positioned in front of the camera. Two light sources were placed diagonally above the scene to create realistic shading and highlights. The camera was configured to look down the negative z-axis with a 60 degree field of view. The scene layout and rendering configuration were defined using a structured JSON format, which specified object positions, colors, material reflectivity, and lighting parameters.

IV. Experimental Setup

We measured render times for resolutions: 640×360, 1280×720, 1920×1080, 3840×2160. For each resolution we executed runs with worker counts $p = 1, 2, 4, 8, 16$ and repeated each experiment three times to observe variance. CPU

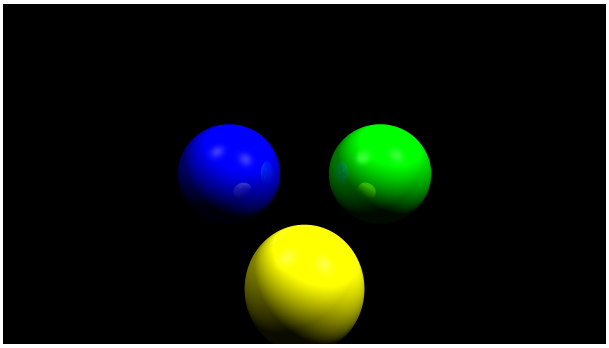

Fig. 1. Sample Output Rendered Image at 3840×2160 with 16 Threads

utilization (average and max) was recorded alongside wall clock time.

V. Results

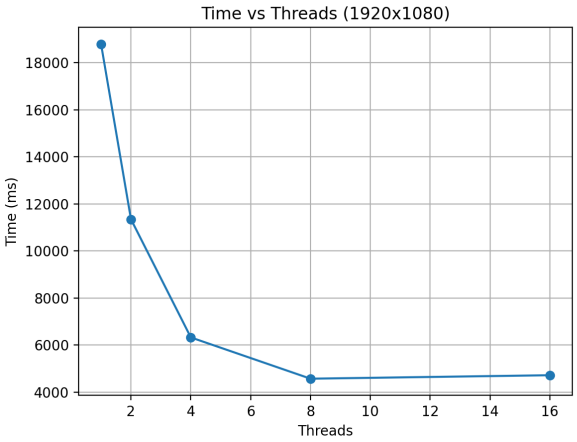

Fig. 2. Render Time vs Threads for 1920x1080 Resolution

The tests were performed at four resolutions: 640×360, 1280×720, 1920×1080, and 3840×2160. Each configuration was run three times, and the median time is reported to reduce short lived system noise. Figures 2–5 show the main trends for 1920×1080, and Fig. 16 shows how time grows with resolution when using 16 workers.

Key observations: The render time consistently decreased as the number of threads increased from 1 to 8, as shown in Fig. 2. This is because the workload dividing the image into tiles benefits from having multiple workers operate on independent regions in parallel. For 1920×1080 images, using 4 to 8 threads achieved nearly 3× to 4× speedup compared to single threaded runs. However, increasing to 16 threads often led to marginal or even negative performance impact, especially at lower resolutions like 640×360 and 1280×720. This occurs due to higher context switching overhead, cache contention, and idle threads when the number of tasks becomes smaller than available workers.

In Fig. 3, we observe near linear speedup up to 4 threads, which then flattens. This behavior aligns with Amdahl's Law

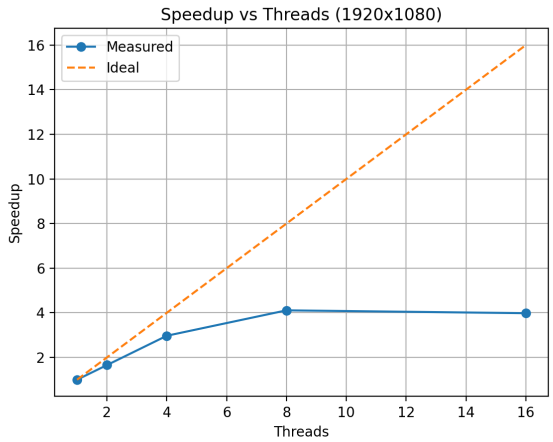

Fig. 3. Speedup vs Threads for 1920x1080 Resolution

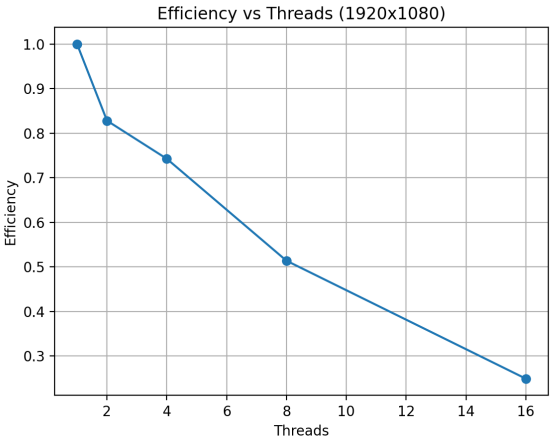

Fig. 4. Parallel Efficiency vs Threads for 1920x1080 Resolution

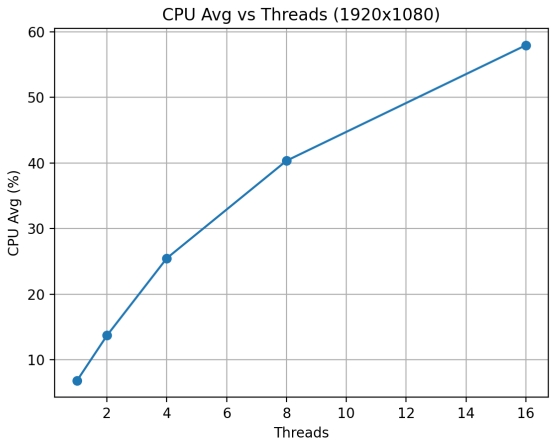

Fig. 5. CPU Average Utilization vs Threads for 1920x1080 Resolution

and highlights the presence of sequential parts in the rendering process or thread synchronization delays. For example, coordination and result collection at the end of each batch introduce a bottleneck when more workers are involved, especially for lighter workloads.

Fig. 4 illustrates this clearly: parallel efficiency drops below 50% when using 16 threads. This means that although all threads are active, much of their potential performance is wasted on overhead, which could include interprocess communication, queue management, or waiting for available tasks.

CPU utilization metrics across all thread counts and resolutions are summarized in Fig. 15. This figure aggregates both average and maximum CPU usage, revealing that although utilization increases with more threads, it does not always correspond to performance gains. For small resolutions, 16 threads often cause oversubscription and reduced efficiency despite high CPU usage. For larger resolutions, utilization is better balanced and aligned with performance scaling.

Additional plots in the Appendix (Figs. 6–17) further support these findings. High resolutions such as 3840×2160 provide enough tile count and data to keep 16 threads busy, showing improved scaling. In contrast, at 640×360 resolution, scaling flattens quickly because the number of tiles is too low to benefit from many workers, resulting in low parallel efficiency and underutilization.

VI. Discussion

The results match what is expected on a multicore CPU. Performance improves when adding more workers, but only up to a point. After 8 workers, the benefits become smaller because of scheduling overhead, memory access delays, and the fixed amount of work. This agrees with Amdahl's Law, which states that the gains from parallelism decrease as the number of processors increases.

Load balance also plays a role. At low resolutions, there are fewer tiles, so some workers finish early and stay idle. For high resolutions, more tiles help distribute work evenly. CPU usage graphs show that high usage does not always mean faster performance, since overhead can also increase.

VII. Limitations and Future Work

This study uses a single CPU model and Windows 11; results may vary on other architectures or OSes. Future work includes:

- Implementing a C++ renderer to compare process vs thread overhead.
- Profiling hotspots and memory bandwidth to identify bottlenecks.
- Testing adaptive tile sizes and work-stealing schedulers for improved load balance.

VIII. Conclusion

This paper reveals the results of a Python ray tracer's performance with different thread counts on AMD Ryzen 9 5900HX CPU. It was found that the time for rendering was greatly reduced when the parallel processing was used and this was most evident when the scaling was done from 1 to 8 worker processes. The performance gain is mainly due to the efficient division of independent rendering tasks over the available cores. On the other hand, going beyond 8 threads up to the complete 16 logical threads very often the performance gains were minor or even a little bit less than before. This is primarily due to the overhead factors which include thread synchronization, context switching, and increased contention for sharing resources like cache and memory bandwidth.

The implications of these findings are that parallel workloads need to be adjusted in a manner that considers both software structure and hardware constraints. One cannot simply increase thread count to automatically get better performance; the optimal configuration is dependent on workload size, resolution and system architecture.

There are several reasons why future iterations of this ray tracer will have to be written in a compiled, low level language such as C++ or Rust so that there will be less interpreter overhead and the programmer will have more control over memory and concurrency. Besides that, trying out with adaptive tile sizes, task granularity and dynamic scheduling techniques like work stealing could result in better load balancing and thus more efficient CPU utilization especially for complex scenes or heterogeneous systems.

Further improvements could include using a lower level language, testing different tile sizes, or exploring alternative scheduling methods.

Appendix

This appendix includes all supplementary graphs for different resolutions and CPU metrics.

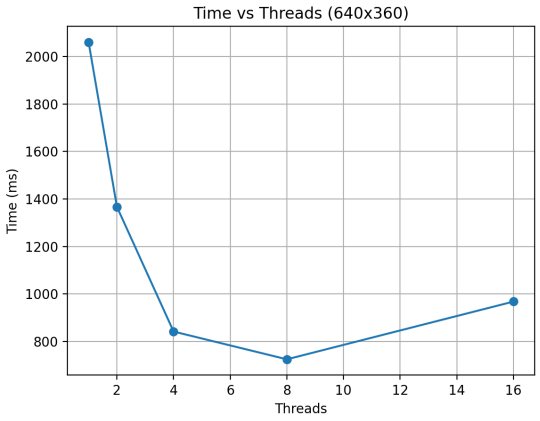

Fig. 6. Render Time vs Threads for 640x360 Resolution

Acknowledgment

We thank our teacher for providing the opportunity to work on this project and for the consistent guidance, feedback, and encouragement offered throughout its development. Their support helped shape the direction of the study and ensured steady progress at each stage.

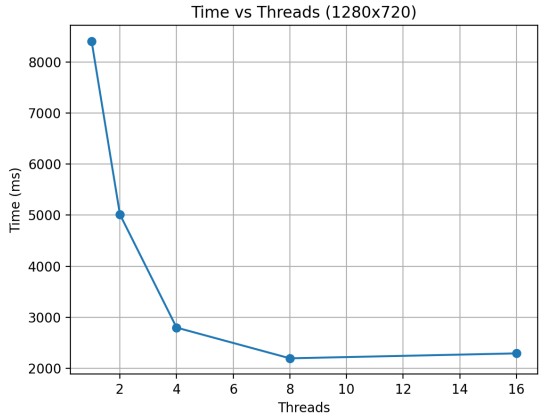

Fig. 7. Render Time vs Threads for 1280x720 Resolution

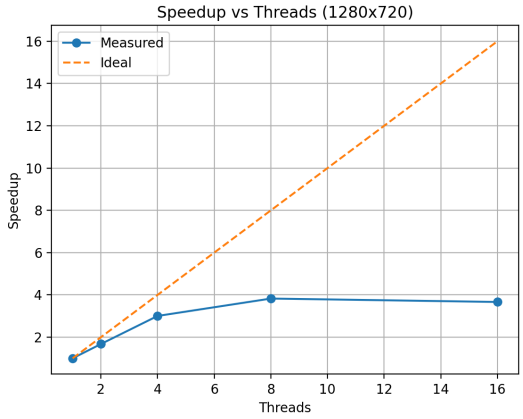

Fig. 10. Speedup vs Threads for 1280x720 Resolution

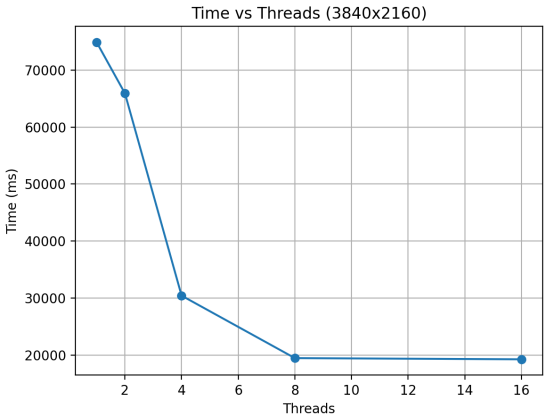

Fig. 8. Render Time vs Threads for 3840x2160 Resolution

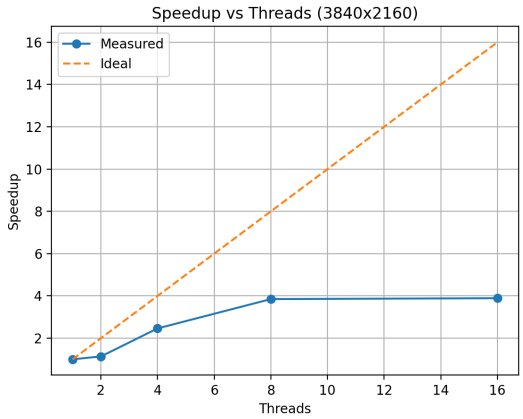

Fig. 11. Speedup vs Threads for 3840x2160 Resolution

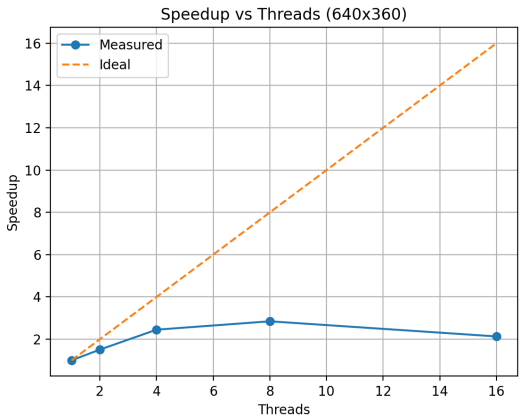

Fig. 9. Speedup vs Threads for 640x360 Resolution

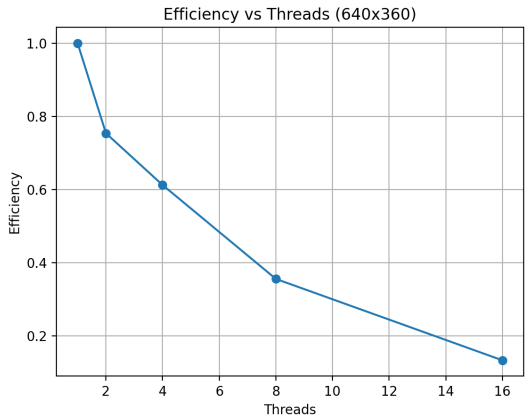

Fig. 12. Parallel Efficiency vs Threads for 640x360 Resolution

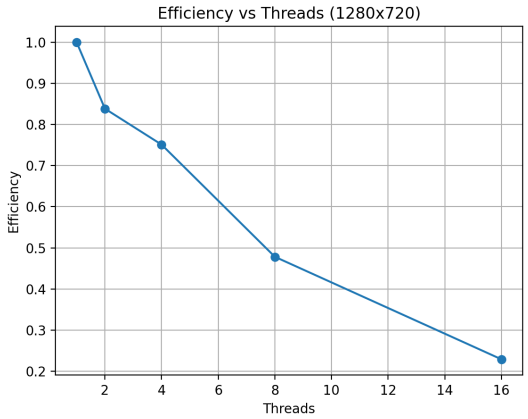

Fig. 13. Parallel Efficiency vs Threads for 1280x720 Resolution

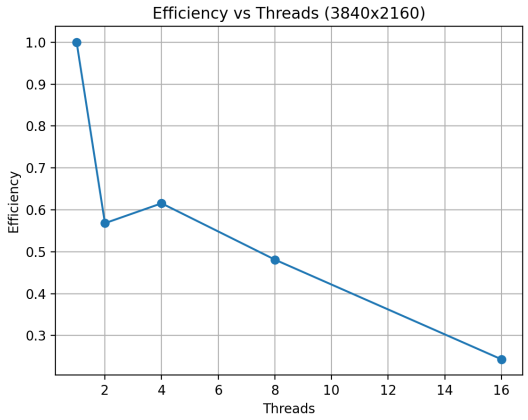

Fig. 14. Parallel Efficiency vs Threads for 3840x2160 Resolution

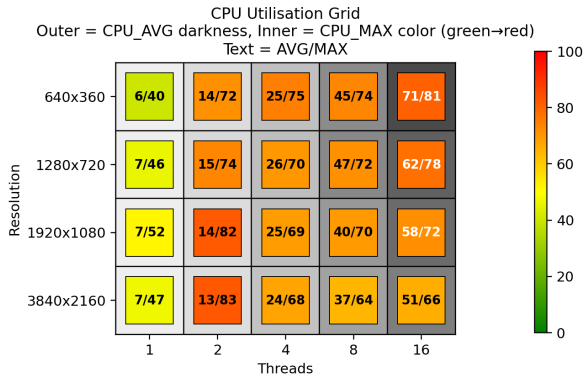

Fig. 15. Combined CPU Avg and Max Utilization vs Threads across all Resolutions

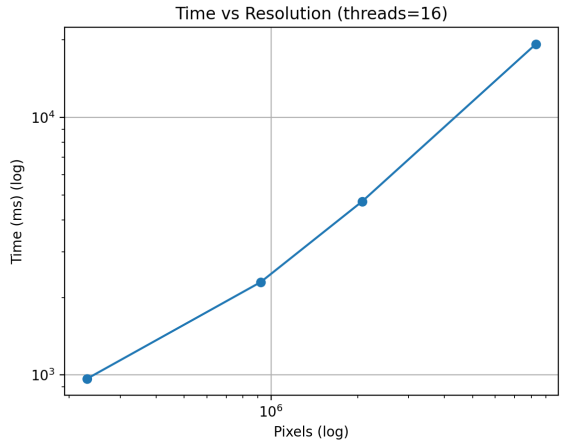

Fig. 16. Render Time vs Resolution at 16 Threads

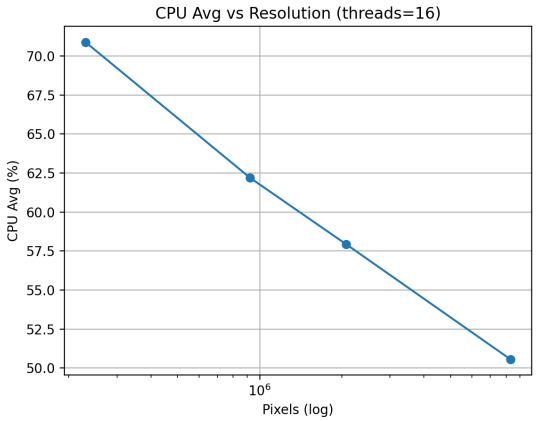

Fig. 17. CPU Average Utilization vs Resolution at 16 Threads

REFERENCES

[1] AMD, "AMD Ryzen 9 5900HX Processor Technical Documentation", 2021.

[2] M. Smith, "Zen 3 Architecture Overview", Journal of Microarchitecture, 2021.

[3] Python Software Foundation, "multiprocessing — Process based parallelism", Python 3 documentation.

[4] Python Software Foundation, "concurrent.futures — Launching parallel tasks," Python 3.14.0 Documentation, 2025.

[5] I. Turner-Trauring, "Python's multiprocessing performance problem," Python Speed blog, Feb. 2023.

[6] Cornell University, "Measuring Efficiency," Parallel Programming Concepts and HPC (Cornell Virtual Workshop), 2025.

[7] I. Wald et al., "Embree: A Kernel Framework for Efficient CPU Ray Tracing," ACM Trans. Graphics, vol. 33, no. 4, 2014.

[8] M. Cui and M. Pericás, "Characterizing and Mitigating Performance Variability in Parallel Applications on Modern HPC multicore Systems," in Proc. 22nd ACM Intl. Conf. Computing Frontiers (CF'25), 2025.

[9] A. Zholdybay and A. Askar, "Python concurrency for high-load multicore processing," Universum: , vol. 10, no. 5 (134), pp. 33–35, 2025.

[10] M. D. Hill and M. R. Marty, "Amdahl's Law in the Multicore Era," IEEE Computer, vol. 41, no. 7, pp. 33–38, July 2008.

[11] J. Spjut, A. Kensler, D. Kopta, and E. Brunvand, "TRaX: A Multicore Hardware Architecture for Real-Time Ray Tracing," IEEE Trans. CAD of Integrated Circuits Syst., vol. 29, no. 11, pp. 1810–1823, 2010.

[12] I. Wald et al., "OSPRay - A CPU Ray Tracing Framework for Scientific Visualization," IEEE Trans. Vis. Comput. Graphics, vol. 23, no. 1, pp. 931–940, Jan. 2017, doi: 10.1109/TVCG.2016.2599041.
[13] V. Fuetterling, C. Lojewski, F. J. Pfreundt, and A. Ebert, "Efficient ray tracing kernels for modern CPU architectures," J. Comput. Graphics Techniques (JCGT), vol. 4, no. 4, 2015.
[14] H. Ono, S. Matsumura, and Z. Ushiyama, "Exploring ghost ray tracing in a massively parallel fast ray tracing within a cloud computing environment," Optical Review, vol. 32, no. 3, pp. 402–411, 2025.
[15] C. Yuksel, "A detailed study of ray tracing performance: render time and energy," The Visual Computer, vol. 34, no. 6, pp. 803–825, 2018.
[16] S. Woop, L. Feng, I. Wald, and C. Benthin, "Embree ray tracing kernels for CPUs and the Xeon Phi architecture," in ACM SIGGRAPH 2013 Technical Briefs, 2013, p. 15.
[17] W. Usher, I. Wald, A. Knoll, M. Papka, and V. Pascucci, "In-situ exploration of particle simulations with CPU ray tracing," Supercomputing Frontiers Innovations, vol. 3, no. 4, pp. 45–58, 2016.
[18] G. Marmitt, H. Friedrich, and P. Slusallek, "Efficient CPU-based volume ray tracing techniques," Comput. Graphics Forum, vol. 27, no. 6, pp. 1687–1709, Sep. 2008.
[19] R. Nordmark and T. Olsén, "A ray tracing implementation performance comparison between the CPU and the GPU," 2022.