Tuijin Jishu/Journal of Propulsion Technology
ISSN: 1001-4055
Vol. 46 No. 3 (2025)

Performance Evaluation of KD-Tree-Optimized Ray Tracing on CPU and GPU

Veena Sanath Kumar[1*], Dr. Rajeswari[2]
[1]Department of ECE, Acharya Institute of Technology, Bangalore, India
[2]Professor & Head, Department of ECE, Acharya Institute of Technology, Bangalore, India

Abstract: **-** Ray tracing has emerged as a powerful rendering technique for generating photorealistic images in scientific visualization, animation, and interactive graphics. This study evaluates and compares the performance of CPU and GPU implementations of a custom ray tracing algorithm without relying on third-party libraries. The CPU version was developed in MATLAB with sequential execution, while the GPU implementation used CUDA with parallel thread processing. Benchmark tests were conducted across resolutions (11×11 to 1000×1000) and scene complexities (1 to 9 spheres). Results show that CPUs outperform GPUs at low resolutions due to lower memory overhead, while GPUs become significantly faster at higher resolutions and complex scenes. Performance profiling reveals GPU execution benefits from scalable parallelism but exhibits variability at smaller workloads. This paper highlights the trade-offs between CPU and GPU-based ray tracing and offers guidance for selecting appropriate hardware based on rendering requirements. Future work includes CPU parallelization, hybrid processing models, and hardware acceleration through VLSI.

Keywords*: Ray Tracing, GPU vs CPU Performance, CUDA Programming, Computer Graphics Rendering, Parallel Processing*

1 Introduction

Over the past few decades, computer graphics have become a crucial aspect of numerous industries and societal applications. They are essential for visualizing scientific discoveries, simulating real-world phenomena, and creating compelling computer-generated imagery (CGI). This growing field has driven the development of advanced techniques to achieve more realistic renderings, with Ray Tracing standing out as a prominent method.The concept of Ray Tracing dates back to the 16th century, but its practical applications only began to materialize in the 1960s. This technique involves simulating the paths of light rays within a scene, modeling their interactions with surfaces, and calculating the resulting visual effects. Initially, Ray Tracing created realistic light reflections in 3D visualizations. However, its adoption slowed due to the limited computational power of early computers and the availability of less resource-intensive alternatives that produced similar results. With advancements in computing technology, Ray Tracing has made a strong comeback and is now widely used.

Today, Ray Tracing is a cornerstone of 3D visualizations, especially in animated films and increasingly in real-time graphics for video games. The continued improvement in computing hardware has made real-time Ray Tracing more feasible, enabling highly realistic rendering in interactive applications.The rise of general-purpose computing on graphics processing units (GPGPU) has further revolutionized Ray Tracing. Unlike traditional CPU-based programming, GPGPU leverages parallel processing to enable real-time Ray Tracing, which is particularly advantageous for applications like gaming. Meanwhile, animation studios often rely on CPU clusters for Ray-Traced rendering, as these systems excel at handling the diverse computational demands of large-scale animation projects.

2 Background

Computer graphics are fundamental to modern applications, influencing entertainment, education, and engineering through visualization and simulation [1]. With increasing demand for realism, techniques like Ray Tracing have become central due to their ability to simulate light behavior accurately.

2.1 Ray Tracing: Principles and Methods

Ray Tracing is a rendering algorithm that models light interactions to produce lifelike images [2]. Inspired by the pinhole camera model (Figure 1), rays are cast from a virtual camera through pixels to compute reflections, refractions, and shadows based on object properties [3].

Two main approaches exist:

- **Forward Ray Tracing** traces rays from light sources but is inefficient due to many rays missing the viewer [2].
- **Backward Ray Tracing** traces rays from the viewer into the scene, focusing only on those contributing to the image, making it more efficient [3].

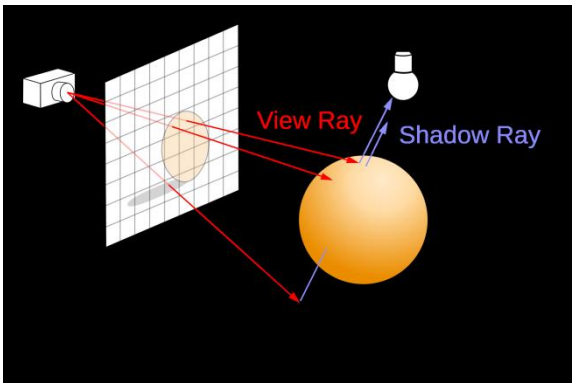


Fig. 2: An established model for describing Ray Tracing. Credit goes to Wikipedia user Henrik.

2.2 Processing Hardware for Ray Tracing

2.2.1 Central Processing Unit (CPU)

CPUs are general-purpose processors with multiple cores optimized for sequential and control-heavy tasks [4]. They effectively orchestrate complex workflows but are less efficient than GPUs for highly parallel workloads like Ray Tracing [5].

2.2.2 Graphics Processing Unit (GPU)

GPUs contain thousands of cores tailored for parallel computation, making them ideal for rendering and data-intensive operations [6]. Their architecture allows simultaneous execution of many threads, enabling faster and more efficient Ray Tracing [7].

2.3 GPU Programming with CUDA

CUDA, developed by NVIDIA, is a platform that enables general-purpose GPU programming [8]. It supports high-level languages and offers memory-sharing features such as Unified Memory, simplifying data exchange between CPU and GPU [9].Tasks are structured into grids and blocks for parallel execution (Figure 3), and synchronization functions ensure coordinated processing.

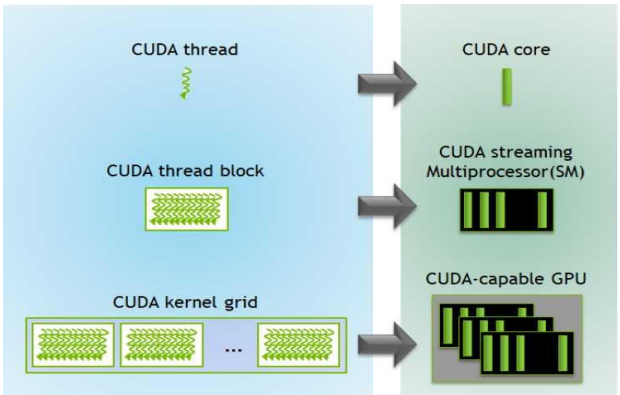


Fig.3: CUDA abstractions mapped to the GPU hardware [10].

2.4 Performance Profiling

Profiling tools assess performance metrics such as memory usage and execution time, helping identify bottlenecks [11]. This is crucial in Ray Tracing and GPU computing, where small optimizations can yield significant performance gains [5].

2.5 Related Work

Norgren [12] showed GPUs outperform CPUs in Ray Tracing due to parallelism, though future CPUs may close the gap. Liljeqvist [13] explored hybrid CPU-GPU setups but found them less effective due to communication overhead.This study builds on these works by benchmarking Ray Tracing performance on raw CPU and GPU hardware without relying on external libraries or engines.

3 Methodology

This section describes the structured approach to implement and evaluate a ray tracing algorithm on CPU and GPU hardware. The methodology covers rendering techniques, performance benchmarking, dataset handling, implementation specifics, profiling, and verification processes.

3.1 Rendering Process

Rendered images were saved in **Portable Pixel Map (PPM)** format for its simplicity and wide compatibility. Each pixel's RGB values (0–255) were calculated based on light interactions in the virtual scene and stored in plain-text format with.ppm extensions. This lightweight approach allowed for straightforward viewing and debugging using tools like GIMP.

3.2 Benchmarking and Scene Setup

Performance testing involved rendering scenes with 1 to 9 spheres arranged in a grid and illuminated by a single light source (Figure 4). Spheres were chosen for their geometric simplicity and ease of intersection testing.

- **Scene Complexity:** Increased by varying the number of spheres.
- **Resolution Range:** From 11×11 to 1000×1000 pixels with a 1:1 aspect ratio.
- **Timing:** Execution times were recorded using MATLAB's Chrono library [14], with each configuration tested 50 times to ensure consistency.

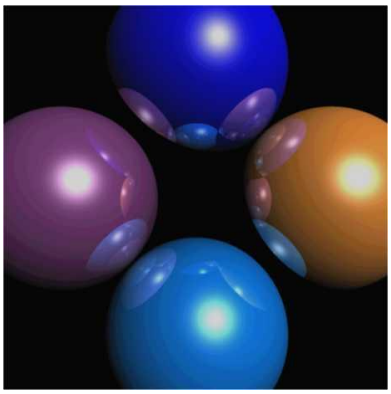


Fig. 4: Example Render Created with the GPU Implementation.

3.3 Dataset Collection

Render times, sphere counts, and resolutions were stored in **CSV format** for easy analysis. Separate datasets were created for CPU and GPU, then merged using a Python script with a hardware identifier for comparison (see Table 1).

Table 1: Example Data Set.

Spheres	Resolution	Time	Type
6	200x200	0.000171	GPU
6	200x200	0.000112	GPU
...	...	...	...
6	200x200	0.084363	CPU
6	200x200	0.078270	CPU

3.4 Implementation Strategies

3.4.1 CPU Implementation (MATLAB)

The CPU version used sequential execution, handling one ray per pixel. It supported recursive reflections (up to 5 bounces) and basic shading. MATLAB's high-level functions facilitated development but limited parallelism.

3.4.2 GPU Implementation (CUDA)

The CUDA version is executed in parallel, assigning rays to individual threads. Core tasks—ray intersection and shading—were handled inside kernels. cudaMallocManaged was used for unified memory allocation, minimizing data transfer between CPU and GPU. This approach significantly improved performance but required careful synchronization and memory handling.

3.5 Profiling and Optimization

The **NVIDIA Visual Profiler** was used to analyze GPU performance:

- **Memory Overhead:** Detected during allocation via cudaMallocManaged, especially for small resolutions.
- **Kernel Insights:** Identified computation bottlenecks and highlighted areas for optimization.
- **Resolution Scaling:** Larger images have better-amortized memory costs, revealing improved GPU scalability.

3.6 Verification

Correctness was validated by comparing rendered outputs:

- **Visual Accuracy:** Images were inspected for lighting, shadows, and reflections.
- **Cross-Validation:** CPU and GPU outputs were compared to ensure consistent results.
- **Debugging Aid:** Render anomalies helped locate logical errors in code.

3.7 Hardware Configuration

Experiments were conducted on:

- **CPU:** Intel Core i7-6700K (4.00 GHz, 4 cores/8 threads).
- **GPU:** NVIDIA GTX 1070 (8GB GDDR5, 1920 CUDA cores).

While sufficient for testing, the older hardware may impact rendering performance at higher resolutions. Newer GPUs could offer better efficiency and scalability.

4 Ray Tracing on the GPU

Advancements in hardware and algorithmic efficiency have driven the evolution of real-time ray tracing. Early implementations relied on CPU-based approaches, with Muuss (1995) demonstrating distributed ray tracing for missile tracking, followed by Parker et al. (1999) and Wald et al. (2001), who optimized ray tracing for CPUs. The shift toward **GPU-based ray tracing** began with Purcell et al. (2002), utilizing programmable GPU pipelines, and was later enhanced by custom accelerators such as SaarCOR (Schmittler et al., 2004) and RPU (Woop& Schmittler, 2005). Recent improvements, including Foley's kD-Tree optimizations (2005) and Popov et al.'s spatial indexing techniques (2006), have significantly boosted GPU efficiency in ray tracing applications.

4.1 Iterative Ray Tracing

Traditional recursive ray tracing is inefficient for GPU architectures due to stack memory constraints and the high cost of deep recursion. An iterative ray tracing algorithm was developed to address this, replacing recursion with an explicit stack. This approach stores secondary rays, such as reflections and refractions, and processes them iteratively until all rays have been traced. By eliminating deep recursive calls, iterative ray tracing significantly reduces memory overhead, ensuring efficient GPU execution.

4.2 Parallelization Strategies

Ray tracing inherently lends itself to parallelism, with each pixel or ray being processed independently. On **CPUs**, parallel execution is achieved through **multi-threading**, **SIMD (Single Instruction, Multiple Data) optimizations**, and **packet traversal**, where grouped rays share computations to enhance cache efficiency. On **GPUs**, workloads are divided among thousands of threads, with each thread handling a ray or pixel, structured in **warps and blocks** to optimize execution.

The **TRACE function** governs ray processing, handling intersections, shading, and reflection/refraction calculations while leveraging depth and refraction stacks for efficient traversal. This function ensures the

computational workload is distributed efficiently, maximizing GPU utilization for real-time applications. he following pseudo-code outlines the **iterative ray tracing approach**:

```
FUNCTION TRACE(primary_ray)
color ← BLACK
   ray ← primary_ray
refractionStack ← NEW STACK
depthStack ← NEW STACK
treeDepth ← 0
continueLoop ← TRUE

   WHILE continueLoop DO
      hit ← INTERSECT(ray, scene)

      IF hit THEN
color += SHADE(ray, hitPoint)
treeDepth += 1

         IF material is REFLECTIVE AND treeDepth ≤ maxDepth THEN
            IF material is TRANSPARENT AND NOT TOTAL_INTERNAL_REFLECTION THEN
refractionStack.PUSH(RAY(hitPoint, refracted(ray.direction)))
depthStack.PUSH(treeDepth)
            END IF
            ray ← RAY(hitPoint, reflected(ray.direction))
         ELSE
continueLoop ← FALSE
         END IF
      ELSE
color += BACKGROUND_COLOR
continueLoop ← FALSE
      END IF

      IF NOT continueLoop AND refractionStack is NOT EMPTY THEN
         ray ← refractionStack.POP()
treeDepth ← depthStack.POP()
continueLoop ← TRUE
      END IF
   END WHILE

   RETURN color
END FUNCTION
```

This **iterative approach** ensures that reflections and refractions are processed efficiently **without deep recursion**, making it well-suited for **GPU execution**. Each thread in a GPU **processes a single ray independently**, storing reflection and refraction rays dynamically in a **stack-based data structure**. By leveraging **explicit stack management**, the algorithm efficiently tracks ray interactions while avoiding costly **function call overheads** associated with recursion.

Compared to recursive implementations, this approach achieves **higher memory efficiency, reduced stack overflow risks, and better parallel scalability**, making it ideal for **real-time GPU ray tracing**.

4.3 kD-Trees for Efficient Ray Tracing

Spatial acceleration structures such as **kD-Trees**are crucial in optimizing ray-object intersection tests. Implementing kD-Trees on GPUs presents challenges related to stack management and traversal efficiency. **Stack-based iterative traversal** methods explicitly track traversal states, allowing for efficient navigation through spatial hierarchies. However, stack-based approaches can be memory-intensive, necessitating alternative techniques.

4.3.1 Stackless Traversal Methods

Two notable **stackless traversal** techniques, **kD-Restart,** and **kD-Backtrack**, have been developed to mitigate memory constraints. The **kD-Restart algorithm** restarts traversal from the root whenever an intersection is missed, reducing stack usage but introducing redundant calculations. An improved variation, **kD-Backtrack**, utilizes parent links to resume traversal at the lowest ancestor node, minimizing unnecessary computations and improving performance.

4.3.2 Hybrid Approaches

Hybrid techniques such as **Short-Stack and Push-Down** balance stack-based and stackless methods. **Short-stack traversal** employs a small, fixed-size stack, defaulting to restart behavior if exceeded, while **Push-Down optimization** allows traversal to resume deeper within the hierarchy, reducing redundant computations. These methods provide a trade-off between memory efficiency and traversal speed, making them suitable for GPU-based ray tracing.

4.4 The Uber-kD-Tree: A Unified Spatial Hierarchy

To further optimize spatial partitioning, the **Uber-kD-Tree** integrates multiple object-based kD-Trees into a **global hierarchy**, reducing traversal redundancy and memory overhead. This structure significantly improves scalability in complex scenes by merging redundant nodes and optimizing traversal paths. The Uber-kD-Tree enhances ray tracing efficiency by structuring spatial data more effectively, enabling real-time rendering in applications requiring high geometric complexity.

4.5 Advancements in GPU Architecture for Ray Tracing

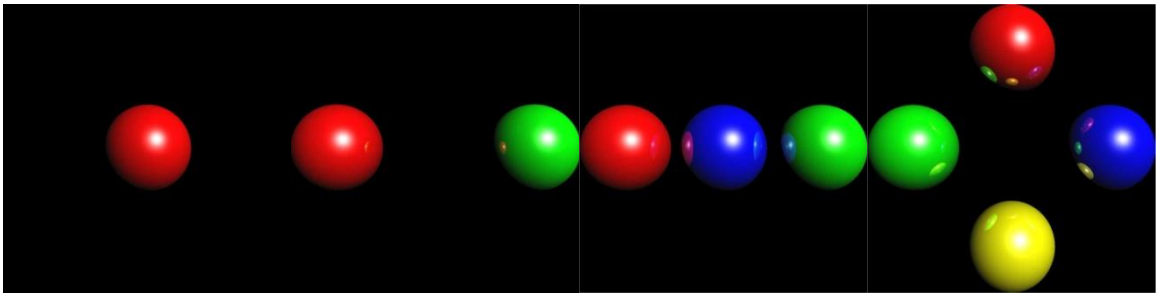
Recent GPU advancements have significantly improved real-time ray-tracing capabilities, including high-bandwidth memory (HBM), cache hierarchies, and dedicated ray-tracing cores. Combined with algorithmic optimizations, these enhancements enable applications in **computer graphics, gaming, virtual reality, and scientific visualization** to achieve unprecedented levels of realism and interactivity. The combination of **iterative ray tracing, parallel processing, and optimized spatial data structures** continues to drive the performance and feasibility of real-time ray tracing on modern GPUs.

5 Results and Analysis

This section presents a comparative analysis of CPU and GPU rendering performance, focusing on two critical factors: **resolution** and **scene complexity**. Graphs and statistical results provide insights into execution time variations across different configurations. The findings help assess the efficiency of hardware in rendering tasks, particularly for real-time applications.

5.1 Impact of Resolution on Rendering Performance

Resolution is key to rendering time, as higher pixel counts require increased computational effort. This subsection explores how **resolution scaling** affects execution time while keeping scene complexity constant. A single sphere was used across all resolution tests to ensure a controlled study, as shown in **Figure 7**. This standardization ensures that variations in rendering time are attributed solely to changes in resolution rather than scene composition.



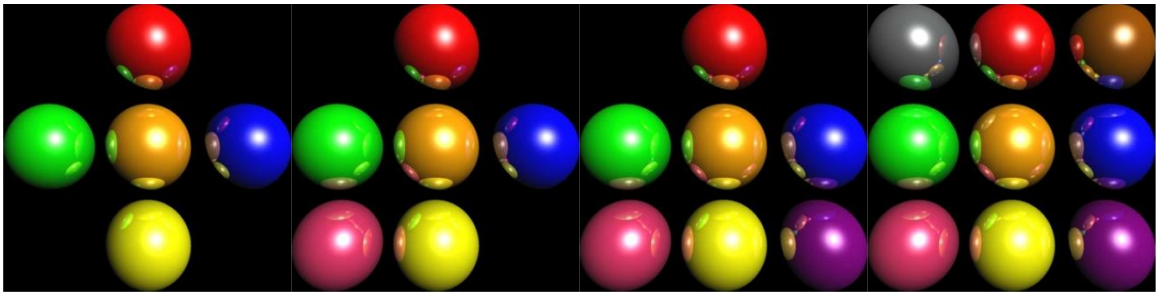


Fig. 7: Sphere 1 to 9 Rendered on The GPU with a Resolution of 1000x1000 Px.

5.1.1 CPU vs. GPU Execution Times

Figure 8 illustrates the relationship between **rendering time and resolution** for CPU and GPU implementations. The resolutions tested range from **1×1 to 20×20 pixels**, where an intersection between CPU and GPU execution times occurs at approximately **14×14 to 15×15 pixels**.

At lower resolutions (**1×1 to 15×15 pixels**), the **CPU outperforms the GPU**, with execution times remaining stable across multiple runs. In contrast, GPU execution times exhibit higher variability, likely due to **thread scheduling inefficiencies** at such low workloads. The **GPU demonstrates its advantage beyond 15×15 pixels**, where parallelism compensates for initial latency, resulting in a steady decrease in execution time relative to the CPU.

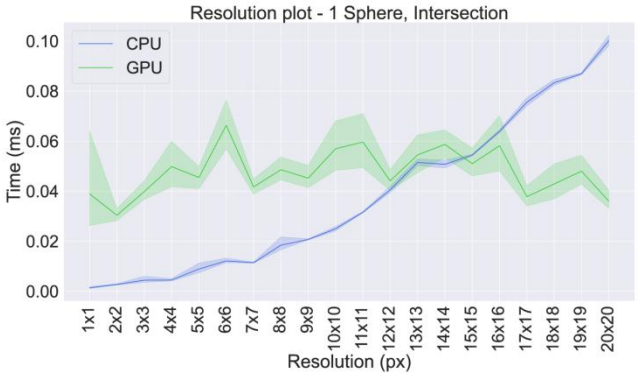


Figure 8: CPU-GPU execution time intersection at 14×14 pixels.

Figure 9 extends the resolution range to 1000×1000 pixels to further analyze this trend, with a logarithmic time axis for clarity. At high resolutions, **GPU execution times stabilize**, reflecting efficient workload distribution. Conversely, CPU execution time increases exponentially, indicating limited parallel scalability. **Beyond 600×600 pixels, the CPU struggles to maintain linear scaling**, confirming that **GPUs are better suited for high-resolution rendering tasks**.

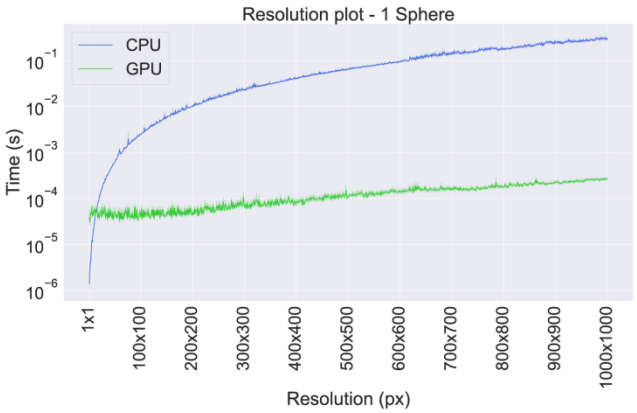


Figure 9: Execution time vs. resolution for CPU and GPU (logarithmic scale).

5.2 Impact of Scene Complexity on Rendering Performance

Beyond resolution, **scene complexity** significantly influences execution time, particularly due to increased ray-object intersection calculations. To analyze this, a series of tests were conducted where the **number of spheres** in the scene varied from **one to nine** while maintaining a **constant resolution** of **1000×1000 pixels**.

Figure 10 presents **bar graphs** showing execution times across different numbers of spheres and resolutions. The results highlight that **scene complexity heavily influences CPU performance**, while the **GPU exhibits variable behavior across different configurations**.

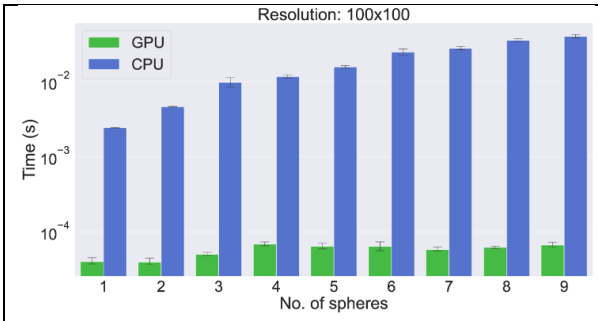
5.2.1 CPU vs. GPU Performance Trends

CPU Performance:

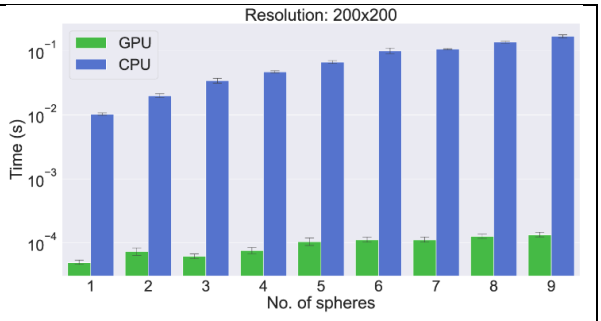
- The CPU shows a **linear increase in execution time** as more spheres are added.
- The CPU is relatively efficient for lower numbers of objects, but performance degrades as **ray-object intersections increase**.

GPU Performance:

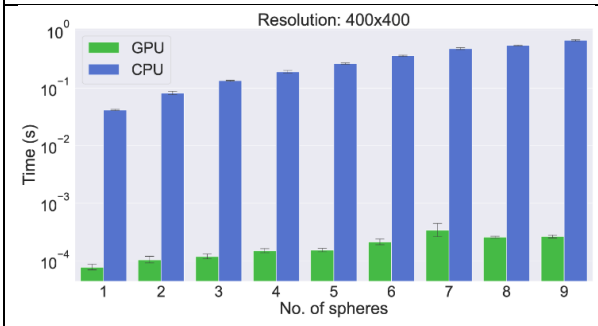
- The **GPU demonstrates inconsistencies** at lower sphere counts, where rendering time fluctuates due to **thread execution overhead**.
- As scene complexity grows, the GPU's **parallelization becomes more effective**, showing **less variation in execution time**.
- **Beyond 600×600 resolution, GPU consistently outperforms CPU**, validating its **scalability in complex rendering tasks**.



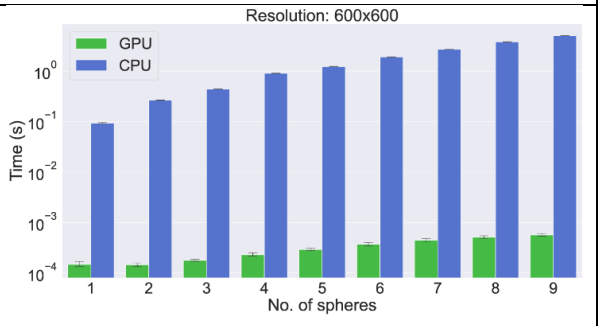
(a) Execution Time for 100x100 Pixel Resolution.



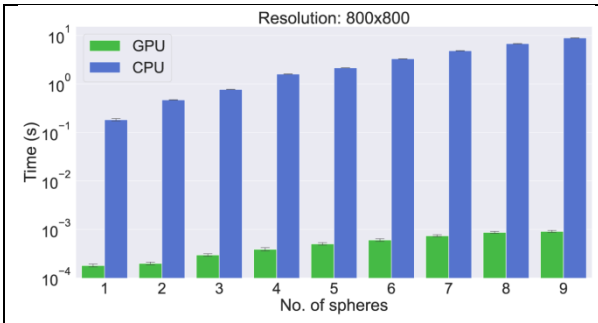
(b) Execution Time for 200x200 Pixel Resolution.



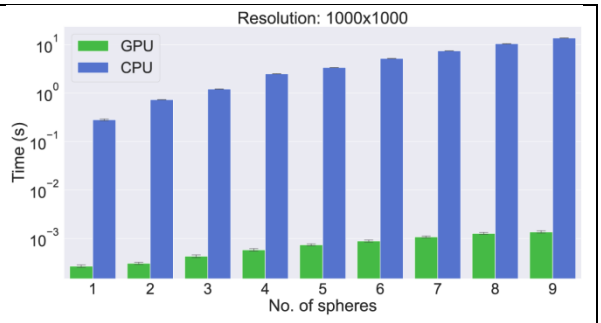
(c) Execution time for 400×400 px resolution.



(d) Execution time for 600×600 px resolution.



(e) Execution time for 800×800 px resolution.



(f) Execution time for 1000×1000 px resolution.

Figure 10: Execution time for different sphere counts at multiple resolutions.

5.3 Key Findings and Comparative Analysis

1. **Resolution Impact:**
   - The CPU is **more efficient at low resolutions** but **fails to scale** as resolution increases.
   - The **GPU benefits from increased resolution**, stabilizing execution times beyond **600×600 pixels**.
2. **Scene Complexity Impact:**
   - **CPUs exhibit predictable scaling**, with execution time rising consistently as scene complexity grows.
   - **GPU performance fluctuates at low object counts** but stabilizes at higher complexities due to effective parallelism.
3. **CPU vs. GPU Trade-offs:**
   - CPUs are **better suited for low-resolution, low-complexity rendering tasks**.
   - GPUs provide **superior performance for high-resolution and high-object-count scenes**, making them ideal for **real-time applications**.

This study confirms that while **CPUs perform efficiently for simple rendering tasks**, their **scalability is limited** as resolution and complexity increase. **Despite initial performance inconsistencies, GPUs significantly outperform CPUs for high-resolution, complex scenes**, reinforcing their role in modern rendering applications. The findings emphasize the **importance of parallelization in graphics computing**, highlighting the **GPU's strengths in high-performance rendering tasks**.

6 Conclusion and Future Work

This study demonstrates that while CPUs perform efficiently in low-resolution, low-complexity scenes, they are limited by their sequential processing model in scaling up to more demanding rendering tasks. In contrast, GPUs—despite initial latency and execution variability—excel in high-resolution and high-complexity environments due to their extensive parallelism. The intersection point where GPU performance overtakes the CPU was observed around 14×14 pixels, and beyond 400×400 pixels, the GPU achieved speedups of several orders of magnitude. Scene complexity further emphasized the GPU's ability to manage parallel workloads effectively, whereas CPU times increased linearly. These findings confirm the GPU's dominant role in modern rendering pipelines and suggest that CPU-based rendering may still be relevant for lightweight, cost-sensitive applications such as indie games. Future work will explore CPU multi-threading, real-time implementations, and hardware acceleration through VLSI and FPGA integration to further enhance ray tracing performance.

References

[1] Peddie, Jon. Ray Tracing: A Tool for All. Springer Nature Switzerland AG, 2019. ISBN: 9783030174897.
[2] Glassner, Andrew S. An Introduction to Ray Tracing. London: Academic, 1989. ISBN: 0122861604.
[3] Phong, Bui Tuong. "Illumination for Computer Generated Pictures." Communications of the ACM 18.6 (June 1975): 311-317. ISSN: 0001-0782. DOI: 10.1145/360825.360839. URL: (https://doi.org/10.1145/360825.360839).

[4] Hennessy, John L., and David A. Patterson. Computer Architecture: A Quantitative Approach. 5th ed. Waltham, MA: Morgan Kaufmann, 2011. ISBN: 9780123838728.
[5] Rauber, Thomas, and Gudula Runger. Parallel Programming for Multicore and Cluster Systems. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. ISBN: 9783642048180.
[6] Intel. "What is a GPU?" [Online] 2022. URL: (https://www.intel.com/content/www/us/en/products/docs/processors/what-is-a-gpu.html) (visited on 01/06/2022).
[7] Brodtkorb, Andre R., Trond R. Hagen, and Martin L. Søvsetra. "Graphics Processing Unit (GPU) Programming Strategies and Trends in GPU Computing." Journal of Parallel and Distributed Computing 73.1 (2013): 4-13. ISSN: 0743-7315. DOI: (https://doi.org/10.1016/j.jpdc.2012.04.003).
[8] Sanders, Jason, and Edward Kandrot. CUDA by Example: An Introduction to General-Purpose GPU Programming. Boston, Mass.: Addison-Wesley, 2010. ISBN: 9780131387683.
[9] Nvidia. CUDA Toolkit Documentation v11.7.0. 2022. URL: (https://docs.nvidia.com/cuda/index.html) (visited on 01/06/2022).
[10] Gupta, Pradeep. "CUDA Refresher: The CUDA Programming Model." [Online] 2020. URL: (https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/) (visited on 01/06/2022).
[11] Chien, Steven, Ivy Peng, and Stefano Markidis. "Performance Evaluation of Advanced Features in CUDA Unified Memory." In: 2019 IEEE/ACM Workshop on Memory-Centric High-Performance Computing (MCHPC). 2019, pp. 50-57. DOI: 10.1109/MCHPC49590.2019.00014.
[12] Norgren, Daniel. "Implementing and Evaluating CPU/GPU Real-Time Ray Tracing Solutions."Mälardalen University, 2016.
[13] Liljeqvist, Erik. "Evaluating a CPU/GPU Implementation for Real-Time Ray Tracing."Mälardalen University, 2017.
[14] Cppreference.com. "Date and Time Utilities." [Online] 2022. URL: (https://en.cppreference.com/w/cpp/chrono) (visited on 01/06/2022).
[15] Rossant, Cyrille. "Straightforward Ray Tracing Engine." [Online] URL: (https://gist.github.com/rossant/6046463). 2017.
[16] Nordmark, Robin, and Tim Olsen. "CPU/GPU Performance Analysis of Ray Tracing." [Online] URL: (https://github.com/skvarre/raytracing). 2022.
[17] Freeman, Guo, et al. "Pro-Amateur-Driven Technological Innovation: Participation and Challenges in Indie Game Development." In: Proc. ACM Hum. -Comput. Interact. 4. GROUP (Jan. 2020). DOI: 10.1145/3375184. URL: (https://doi.org/10.1145/3375184).