EE/CSCI 451 Fall 2023
The Term Project Final Report

Design and Implementation of a Parallel Graphics Rendering Engine

Ke Wang (ID: 1179479122)
Kyuhyeon Nam (ID: 3301972723)

December. 2023

Table of Contents

Abstract

This report presents the outcomes of the project "Design and Implementation of a Parallel Graphics Rendering Engine," which aimed to enhance the efficiency and speed of 3D graphics rendering using parallel computing techniques. Led by Kyuhyeon Nam and Ke Wang, the project focused on integrating the 'Path Tracing' algorithm with CUDA and OpenMP for efficient parallel processing. The primary goal was to develop a rendering engine capable of handling complex 3D scenes more effectively than traditional rendering methods.

The implementation involved optimizing the 'Path Tracing' algorithm for parallel execution, utilizing CUDA for GPU acceleration, and employing OpenMP for multi-threaded CPU processing. This approach addressed several technical challenges, including memory management optimization and synchronization overhead minimization.

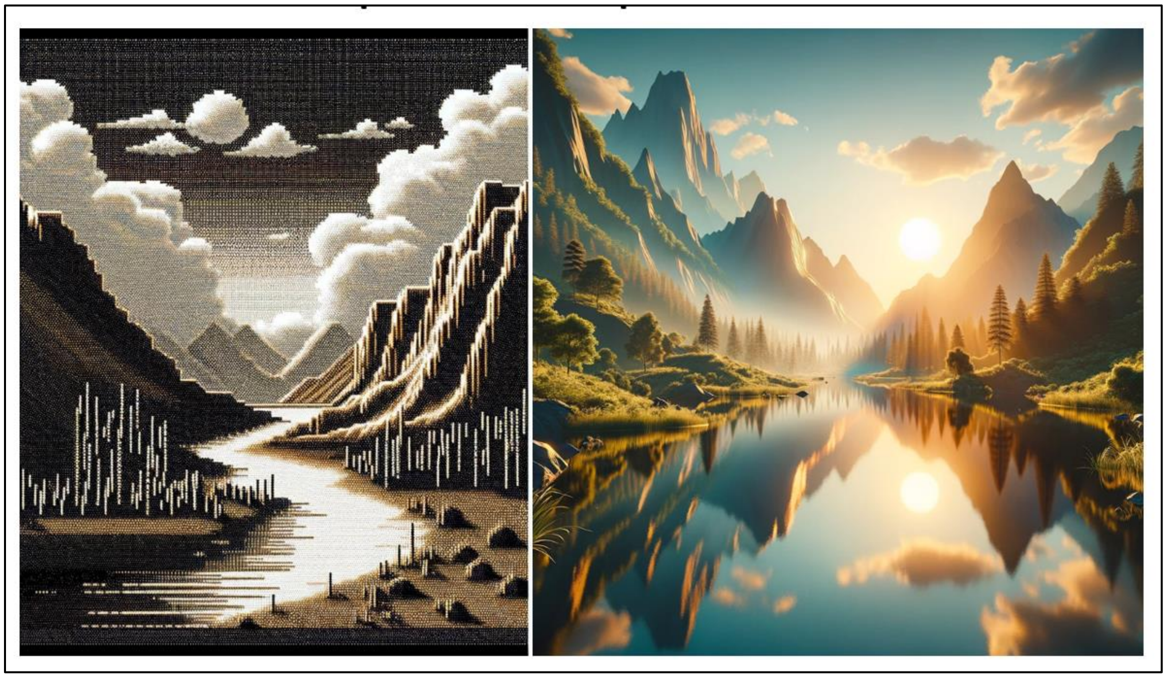
Comprehensive experiments were conducted to evaluate the engine's performance, focusing on variables like resolution, sample count, and thread count. These experiments demonstrated significant improvements in rendering speed, particularly with increased thread counts, while maintaining high image quality.

The project's success illustrates the potential of parallel computing in revolutionizing 3D graphics rendering. It offers valuable insights into workload distribution, computational efficiency, and the balance between speed and image quality in parallel processing environments. This work lays a foundation for future advancements in graphics rendering and sets a benchmark for similar endeavors in the field of computer graphics. The insights and methodologies developed through this project have implications for various applications, from gaming and animation to virtual reality and scientific visualization, underscoring the project's broader impact on technology.

1. Introduction

1.1 Background and necessity of the project

In the constantly evolving field of computer graphics, there is an increasing demand for rendering complex 3D scenes with higher levels of fidelity and efficiency. The development of "Design and Implementation of a Parallel Graphics Rendering Engine" comes as a response to the challenges presented by modern 3D graphics. While traditional rendering techniques are effective, they may not be sufficient to handle the intricacies and dynamics of contemporary graphics. Parallel computing provides a promising solution, allowing for the distribution of rendering tasks across multiple processing units, resulting in significant performance improvements and reduced processing times.


<Figure 1. Early graphics vs Modern Graphics>

1.2 Objectives of the Project

The primary objective of this project, undertaken as a part of graduate-level research, was to explore the capabilities of parallel computing in enhancing the efficiency of 3D graphics rendering. The focus was on integrating the 'Path Tracing' algorithm with parallel computing technologies, specifically CUDA and OpenMP, to observe and measure improvements in rendering performance. This objective was driven by a need to contribute to academic knowledge in the fields of computer graphics and parallel processing, offering practical insights into the application of these technologies in rendering tasks.

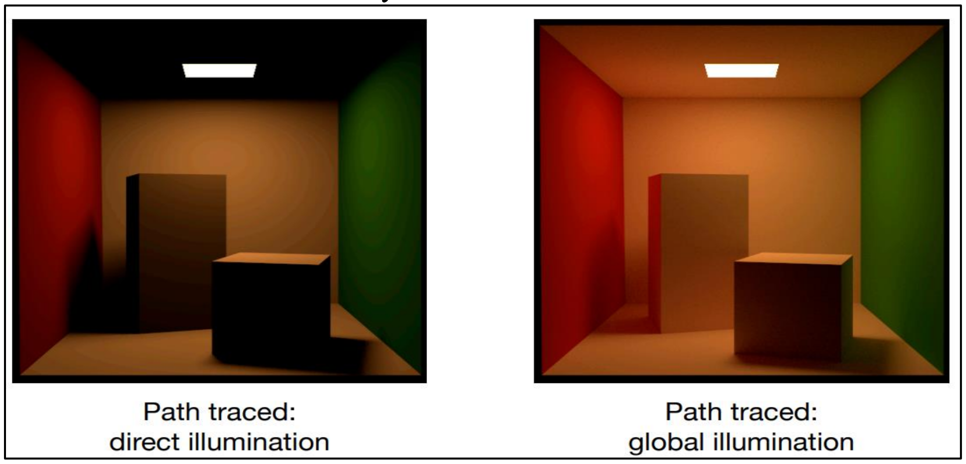
1.3 Significance of the Project

From an academic perspective, this project holds significant value in contributing to the understanding of parallel processing's role in graphics rendering. It provides a practical case study for the application of CUDA and OpenMP in a real-world problem, offering valuable data and insights for future research. Furthermore, by documenting the challenges and solutions encountered during the project, it serves as an educational resource for other students and researchers interested in similar topics. The project’s findings contribute to the ongoing discourse in the academic community about optimizing rendering techniques, potentially inspiring future innovations and experiments in this area.

2. Project Overview

2.1 Objective and Scope

The objective of this project was to create a graphics rendering engine that not only harnesses the latest advancements in parallel computing but also focuses on elevating the realism in rendering complex 3D scenes. The key to achieving this was the strategic integration of the 'Path Tracing' algorithm, known for its ability to produce highly realistic and lifelike images. Our approach involved utilizing parallel computing technologies like CUDA and OpenMP to efficiently divide rendering tasks across multiple processors. The scope of the project was not limited to developing a high-performance engine; it also included a comprehensive evaluation of the engine's performance in diverse scenarios, emphasizing both technological innovation and enhancement of visual authenticity.



<Figure 2. Path Tracing for realistic rendering>

2.2 Hypothesis

“Each thread(each pixel) can calculate independently”

At the outset of this project, we formulated a hypothesis that guided our research and development process. Our hypothesis was: "By integrating the 'Path Tracing' algorithm with parallel computing technologies like CUDA and OpenMP, we can significantly improve the efficiency and realism of 3D graphics rendering, compared to traditional methods." This hypothesis was rooted in the belief that parallel computing can effectively manage the computationally intensive tasks of realistic rendering, thereby enhancing both the speed and quality of the output. The project aimed to test this hypothesis by developing a rendering engine that combines the detailed light simulation of 'Path Tracing' with the high-speed processing capabilities of modern parallel computing frameworks.

2.3 Methodology and Implementation

Our methodology began with designing an architecture capable of integrating 'Path Tracing' with parallel computing. The implementation involved optimizing the 'Path Tracing' algorithm for parallel execution, using CUDA for GPU acceleration, and employing OpenMP for multi-threading in CPUs. This dual approach was aimed at improving rendering speeds while maintaining the quality of the output. The project also focused on addressing technical challenges such as memory management and minimizing synchronization overheads, which are critical in parallel processing environments.

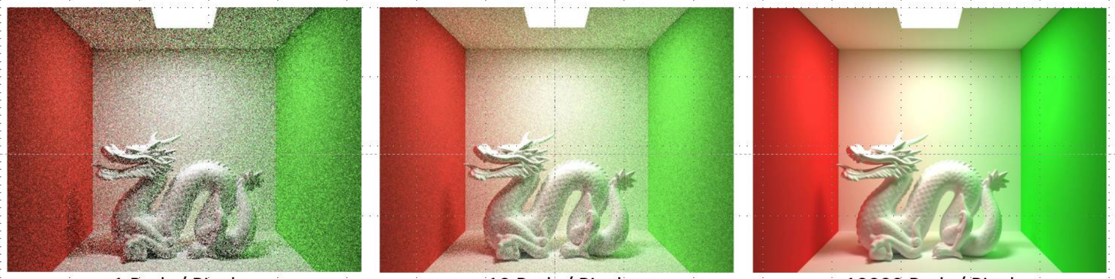
2.4 Significance and Potential Impact

The development of this rendering engine has significant implications for the field of computer graphics, particularly in applications requiring high levels of realism, such as virtual reality, gaming, and animation. By improving the efficiency and realism of rendered images, the project contributes to both the technological and artistic aspects of computer graphics. The insights gained from this project regarding the application of parallel computing in realistic rendering are expected to inspire further research and advancements in the field, showcasing the project's potential as a benchmark in the industry.

Overview of 'Path Tracing' Algorithm

Understanding 'Path Tracing' Algorithm

As a cornerstone of this project, the 'Path Tracing' algorithm plays a crucial role in achieving realistic rendering of 3D scenes. Originating from the field of computer graphics, Path Tracing is a Monte Carlo method of rendering images of 3D scenes with a high degree of realism. Unlike traditional rasterization-based methods, Path Tracing simulates the way light travels in the real world, calculating the color of pixels by tracing the possible paths of light rays as they interact with surfaces in a scene. This process involves random sampling and averaging over many possible light paths, which allows for the creation of highly realistic images featuring accurate lighting, shadows, and reflections.



Relevance to the Project

In our project, optimizing the 'Path Tracing' algorithm for parallel execution was a critical aspect. We utilized CUDA for GPU acceleration and OpenMP for multi-threaded CPU processing, which enabled us to handle complex 3D scenes more effectively than traditional rendering methods. This approach not only addressed technical challenges such as memory management optimization and synchronization overhead minimization but also provided a platform to demonstrate significant improvements in rendering speed, particularly with increased thread counts, while maintaining high image quality.

3. Implementation Details

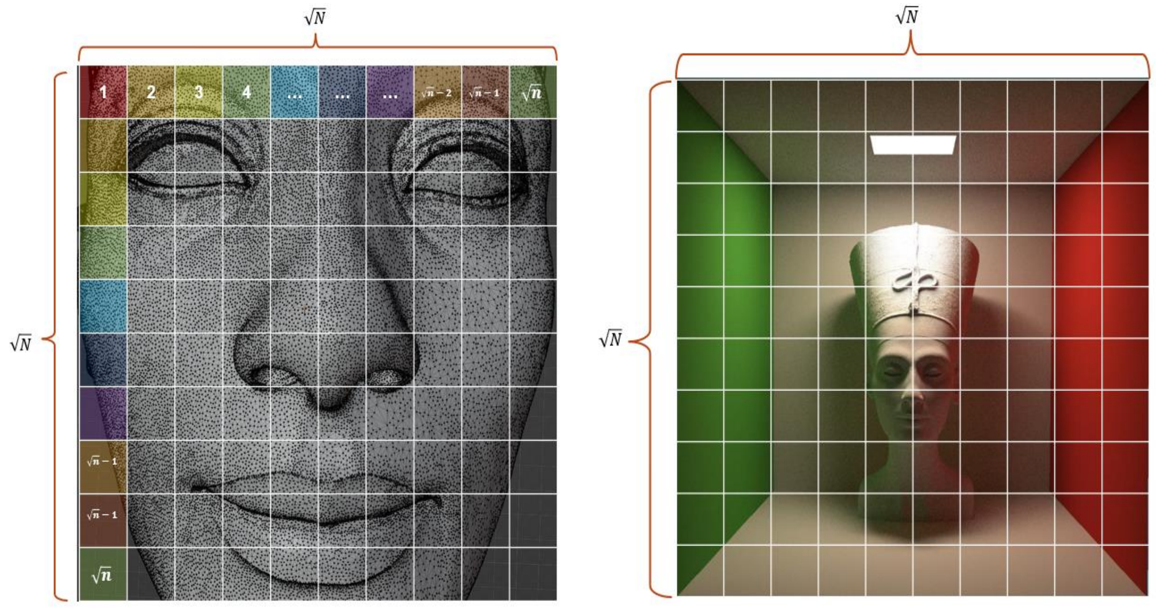
3.1 Key Idea and Implementation Strategy

The cornerstone of our project's implementation was a novel approach to 3D graphics rendering. Central to this was the concept of dividing the rendering workload into smaller, more manageable units. Specifically, we adopted a strategy where objects or rendering images were segmented into $\sqrt{n} \times \sqrt{n}$ grids. Each grid segment was then assigned to individual threads for computation. This approach was fundamental for two reasons:

1. **Efficiency in Parallel Processing**: By breaking down the rendering task into smaller segments, we could more effectively utilize the parallel processing capabilities of both CPUs and GPUs. This ensured that each processor worked on a manageable portion of the rendering task, leading to more efficient computation and reduced processing time.

2. **Enhanced Image Quality**: This method allowed for more detailed and accurate calculations of light interactions within each segment, contributing to the overall realism and quality of the rendered image. The granular approach facilitated by dividing the image into $\sqrt{n} \times \sqrt{n}$ grids enabled a more precise simulation of lighting and shading effects.

By implementing this key idea, we were able to leverage the strengths of the 'Path Tracing' algorithm in a parallel computing environment, significantly enhancing both the efficiency and quality of our 3D rendering engine.



<Figure 3. Key Idea>

3.2 Utilization of 'Path Tracing' Algorithm

In implementing the 'Path Tracing' algorithm, our focus was on capturing its inherent capability to produce highly realistic images. The implementation process was multifaceted, involving several key steps:

1. **Ray Tracing Mechanics:** We started by developing the basic ray tracing mechanics, where rays were cast from the camera into the scene to calculate color information based on light interactions.

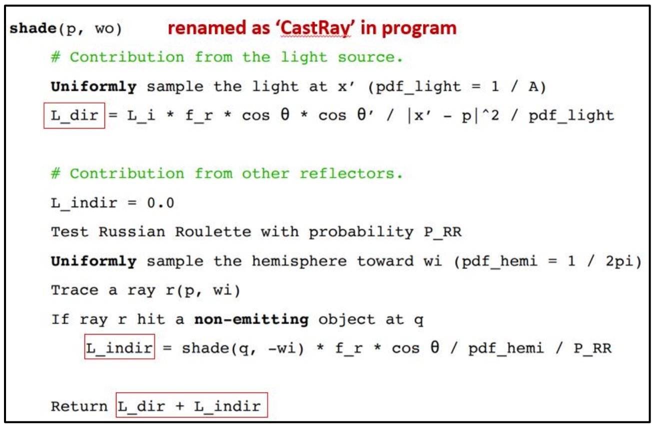
2. **Light Path Simulation**: The core of 'Path Tracing' lies in simulating the paths of light. This involved complex calculations to trace the paths of light as they bounce off surfaces, enabling the creation of realistic lighting effects, including shadows and reflections.

3. **Optimizing for Parallel Processing**: Given the computationally intensive nature of 'Path Tracing', we optimized the algorithm for parallel processing. This optimization was critical for harnessing the full power of CUDA and OpenMP, allowing us to process multiple light paths simultaneously, thereby significantly reducing rendering times.

4. **Grid-Based Division for Rendering:** As detailed on the 12th slide of our presentation, we employed a strategy of dividing the rendering task into $\sqrt{n} \times \sqrt{n}$ grids. Each grid segment was assigned to individual threads, enabling efficient parallel processing and finer control over the rendering outcome.

5. **Iterative Refinement:** The algorithm was refined iteratively, with continuous testing and tweaking to balance between rendering quality and computational efficiency. This process helped in fine-tuning the algorithm to achieve optimal performance on various hardware configurations.

By delving into these specific aspects of the 'Path Tracing' algorithm's implementation, this revised section provides a comprehensive overview of how the key technical component of our project was developed and optimized. It ties in the high-level concepts with the practical steps taken to realize them, offering a clear and detailed picture of the project's technical achievements.

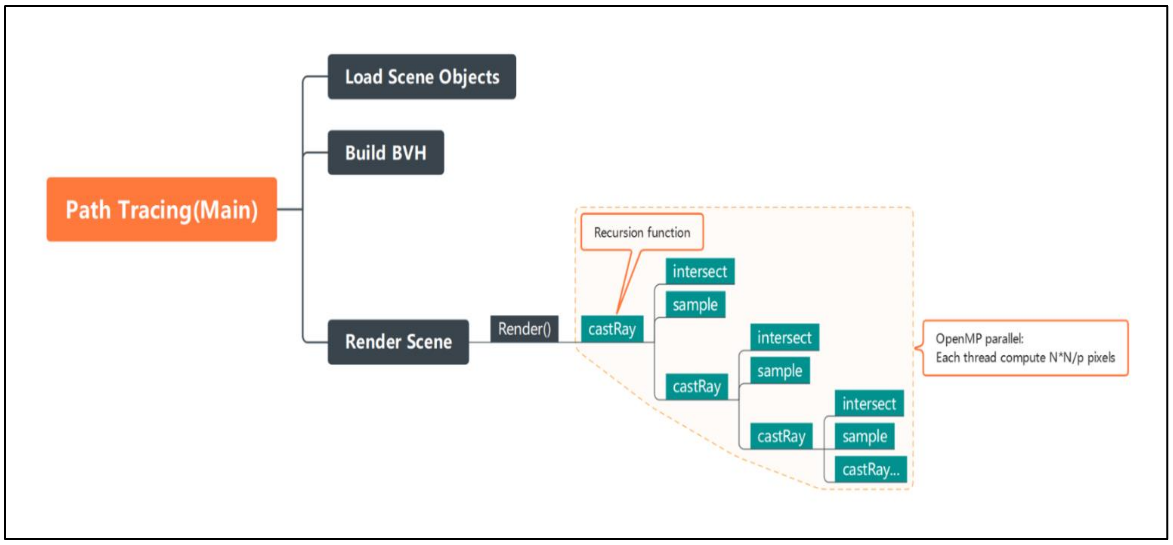


```
shade(p, wo)        renamed as 'CastRay' in program
    # Contribution from the light source.
    Uniformly sample the light at x' (pdf_light = 1 / A)
    L_dir = L_i * f_r * cos θ * cos θ' / |x' - p|^2 / pdf_light

    # Contribution from other reflectors.
    L_indir = 0.0
    Test Russian Roulette with probability P_RR
    Uniformly sample the hemisphere toward wi (pdf_hemi = 1 / 2pi)
    Trace a ray r(p, wi)
    If ray r hit a non-emitting object at q
        L_indir = shade(q, -wi) * f_r * cos θ / pdf_hemi / P_RR

    Return L_dir + L_indir
```

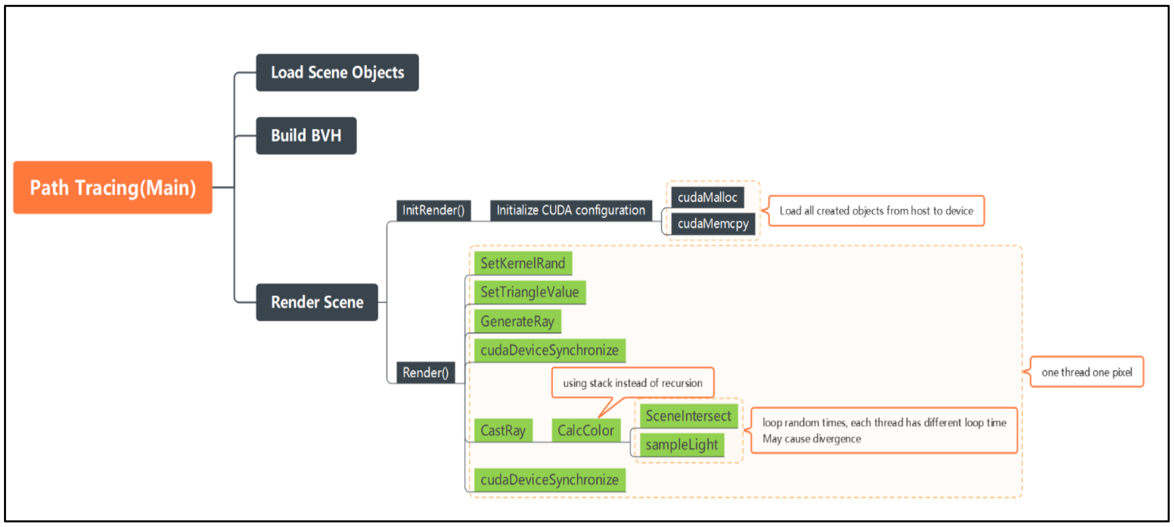
Figure 4. Implementation Algorithm (for each pixel and ray)

3.3 Parallel Processing with CUDA and OpenMP

A significant part of the project involved harnessing the power of parallel computing to improve rendering performance. This was achieved through the use of CUDA and OpenMP, two advanced parallel computing technologies. CUDA was utilized to leverage the parallel processing capabilities of NVIDIA GPUs, enabling the rendering engine to perform intensive computations for graphics rendering in a fraction of the time taken by traditional methods. OpenMP was employed to facilitate efficient multi-threading in CPUs, further enhancing the engine's capability to handle complex rendering tasks simultaneously.



<Figure 5. OpenMP>



<Figure 6. CUDA>

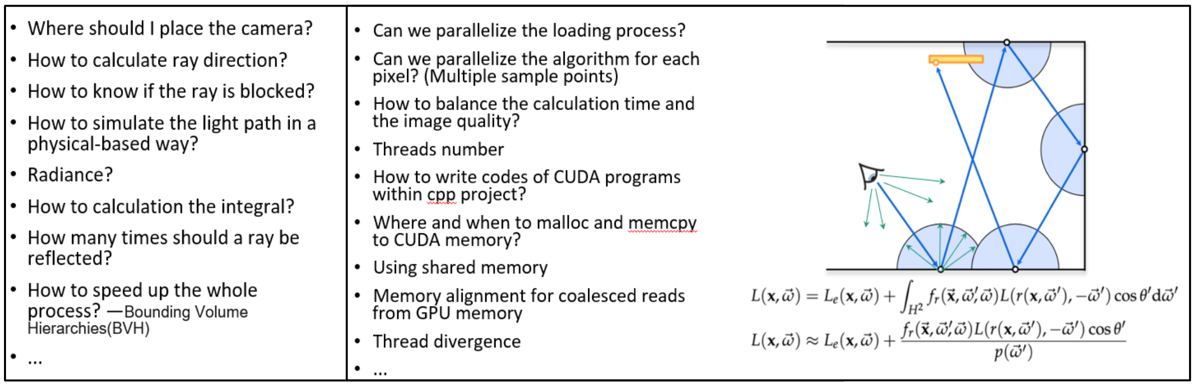
3.4 Technical Challenges and Solutions

The project encountered several technical challenges, particularly in adapting the 'Path Tracing' algorithm for parallel execution and optimizing memory management across different processing units. To address these challenges, the team implemented a series of optimizations:

1. **Optimization of Memory Access Patterns**: Adjustments were made to the way memory was accessed and managed, especially in the context of GPU computing, to reduce latency and increase throughput.

2. **Dynamic Load Balancing**: The engine was designed to dynamically allocate rendering tasks to different processors based on their current workload, ensuring efficient utilization of all available computing resources.

3. **Minimizing Synchronization Overheads**: Efforts were made to minimize the overheads associated with synchronization between different processing units, a common challenge in parallel computing environments.



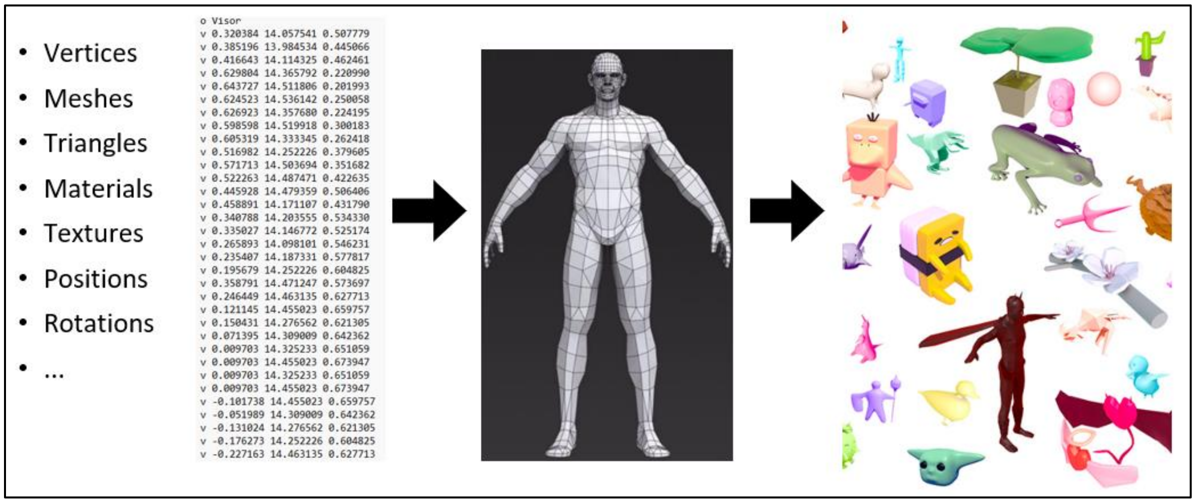
- Where should I place the camera?
- How to calculate ray direction?
- How to know if the ray is blocked?
- How to simulate the light path in a physical-based way?
- Radiance?
- How to calculation the integral?
- How many times should a ray be reflected?
- How to speed up the whole process? —Bounding Volume Hierarchies(BVH)
- ...

- Can we parallelize the loading process?
- Can we parallelize the algorithm for each pixel? (Multiple sample points)
- How to balance the calculation time and the image quality?
- Threads number
- How to write codes of CUDA programs within cpp project?
- Where and when to malloc and memcpy to CUDA memory?
- Using shared memory
- Memory alignment for coalesced reads from GPU memory
- Thread divergence
- ...

$$L(\mathbf{x},\vec{\omega}) = L_e(\mathbf{x},\vec{\omega}) + \int_{H^2} f_r(\vec{\mathbf{x}},\vec{\omega}',\vec{\omega}) L(r(\mathbf{x},\vec{\omega}'),-\vec{\omega}') \cos\theta' \mathrm{d}\vec{\omega}'$$

$$L(\mathbf{x},\vec{\omega}) \approx L_e(\mathbf{x},\vec{\omega}) + \frac{f_r(\vec{\mathbf{x}},\vec{\omega}',\vec{\omega}) L(r(\mathbf{x},\vec{\omega}'),-\vec{\omega}') \cos\theta'}{p(\vec{\omega}')}$$

<Figure 7. Challenge: Algorithm and Parallelization>



- Vertices
- Meshes
- Triangles
- Materials
- Textures
- Positions
- Rotations
- ...

```
o Visor
v 0.320384 14.057541 0.507779
v 0.385196 13.984534 0.445066
v 0.416643 14.114325 0.462461
v 0.629804 14.365792 0.220990
v 0.643727 14.511806 0.201993
v 0.624523 14.536142 0.250058
v 0.626923 14.357680 0.224195
v 0.598598 14.519918 0.300183
v 0.605319 14.333345 0.262418
v 0.516982 14.252226 0.379605
v 0.571713 14.503694 0.351682
v 0.522263 14.487471 0.422635
v 0.445928 14.479359 0.506406
v 0.458891 14.171107 0.431790
v 0.340788 14.203555 0.534330
v 0.335027 14.146772 0.525174
v 0.265893 14.098101 0.546231
v 0.235407 14.187331 0.577817
v 0.195679 14.252226 0.604825
v 0.358791 14.471247 0.573697
v 0.246449 14.463135 0.627713
v 0.121145 14.455023 0.659757
v 0.150431 14.276562 0.621305
v 0.071395 14.309009 0.642362
v 0.009703 14.325233 0.651059
v 0.009703 14.455023 0.673947
v 0.009703 14.325233 0.651059
v 0.009703 14.455023 0.673947
v -0.101738 14.455023 0.659757
v -0.051989 14.309009 0.642362
v -0.131024 14.276562 0.621305
v -0.176273 14.252226 0.604825
v -0.227163 14.463135 0.627713
```

<Figure 8. Challenge: Loading the Scene>

3.5 Performance Optimization Strategies

Alongside these technical solutions, the project also focused on various performance optimization strategies. These included fine-tuning the balance between workload distribution and communication overhead, optimizing algorithm parameters for different hardware configurations, and implementing advanced caching techniques to reduce the computational load.

3.6 Integration and Testing

The integration of these various components into a cohesive rendering engine was a meticulous process, involving iterative testing and refinement. The project team conducted extensive testing under various conditions to ensure that the engine not only functioned as intended but also met the performance benchmarks set at the outset of the project.

3.7 Implementation platform

- Programming Language: C++
- IDE: Visual Studio 2022
- Development OS: Windows
- Development Machine: Laptop
  (Intel i7-7820HQ @2.9GHz, 16GB, Radeon Pro 560 /
  Intel i9-13900K @3.0GHz, 16GB, RTX 4060)

In our project, we strategically chose to develop the parallel graphics rendering engine on a personal laptop using C++ in Visual Studio 2022, because this combination provided an ideal balance of accessibility, performance efficiency, and advanced features for parallel computing, ensuring a flexible yet powerful development environment tailored to our project's technical demands.

The successful implementation of these details was critical in achieving the project's goal of developing a high-performance parallel graphics rendering engine. This section of the report highlights the technical proficiency and innovative problem-solving approaches employed by the project team.

4. Performance Analysis and Experiments

4.1 Experimental Design

To evaluate the performance of the parallel graphics rendering engine, a comprehensive set of experiments was designed and executed. These experiments were aimed at assessing the rendering engine's efficiency and effectiveness under various conditions. Key variables in these experiments included:

1. **Resolution Variations**: Testing was conducted at multiple resolutions to understand how the rendering engine scaled with increased pixel counts.

2. **Sample Count**: The number of samples per pixel was varied to determine the impact on image quality and rendering time.

3. **Thread Count**: The performance impact of using different numbers of threads in both CPU (via OpenMP) and GPU (via CUDA) was analyzed.

4.2 Performance Metrics

The primary metrics used to evaluate performance included rendering time, image quality, and computational efficiency. Rendering time was critical for assessing the speed improvements brought about by parallel processing. Image quality was evaluated to ensure that the speedup did not compromise the visual fidelity. Computational efficiency metrics provided insights into how effectively the engine utilized hardware resources.

4.3 Experimental Environment

To ensure accurate and reliable results, the experiments were conducted in a carefully controlled environment. The key components of this experimental setup were:

1. **Consistent Testing Conditions**: All tests were conducted under consistent conditions, including the same operating system version, driver versions, and other system settings. This consistency was crucial to eliminate external variables that could impact the experiment's outcomes.

    - Platform: CARC (High Performance Computing in USC)

2. **Hardware Configuration**: The tests were carried out on systems equipped with specific hardware components, including high-performance NVIDIA GPUs for CUDA-based processing and multi-core CPUs for OpenMP tasks. This hardware selection was critical to evaluating the rendering engine's performance under realistic yet demanding computational conditions.

    - CPU: Intel® Xeon® Sliver 4126 @ 2.1Ghz

- GPU: NVIDIA Tesla P100
- Memory: 32GB
- CPU / task: 8

3. **Software Setup**: We utilized the latest versions of necessary software tools and libraries. This included up-to-date CUDA and OpenMP libraries, aligning with current standards in parallel computing. The rendering engine was optimized for these platforms to ensure maximum efficiency.

   - gcc: 11.3.0 / cmake: 3.23.2
   - cuda: 11.6.2
   - cudnn: 8.4.0.27-11.6
   - nvhpc: 22.11
   - export LIB: gcc-12.3.0/cuda-12.2.1c-12.3.0/cuda-12.2.1

4. **Benchmarking Approach**: In keeping with the established testing conditions, we proceeded to evaluate the rendering speed of the identical large scale files under every test condition.

   - Object: Nefertiti.obj
   - Size: 183 Mb

5. **Test parameters:** In our experimental setup, we employed a comprehensive range of test parameters to thoroughly evaluate the performance of the rendering engine. This included:

   - **Resolution Variations**: Tests were conducted across a range of resolutions, starting from 256x256 pixels and extending up to 1024x1024 pixels. This range was chosen to understand how the engine performs under varying levels of detail and computational demand.

   - **Thread Count Variations**: We varied the number of threads used in the experiments from 1 to 256. This allowed us to analyze the engine's scalability and efficiency in utilizing parallel processing capabilities at different levels.

   - **Samples Per Pixel (SPP)**: The number of samples per pixel was tested at 32, 64, and 128. These values were selected to assess the impact of sample count on image quality and rendering time, providing insights into the trade-offs between visual fidelity and performance.

   - **CUDA Grid Sizes**: For CUDA-based processing, we experimented with grid sizes ranging from 1x1 to 1024x1024. This variation helped in understanding the optimal grid size for efficient GPU utilization and its effect on rendering performance.

By covering such a diverse range of scenarios, we were able to conduct a thorough and comprehensive analysis of the engine's capabilities.

4.4 Porting to CARC Server for Performance Testing

After the initial development and preliminary testing on a personal laptop, we took a crucial step to ensure the accuracy and reliability of our performance measurements by porting the rendering engine to the CARC (Computing and Advanced Research Computing) server. This transition was essential for several reasons:

1. **Enhanced Computational Resources:** The CARC server offered significantly more advanced computational resources than a personal laptop, providing a more suitable environment for rigorous and intensive performance testing.

2. **Realistic Benchmarking Environment**: By using the CARC server, we could conduct experiments in an environment that more closely resembles the conditions under which high-performance rendering engines are typically employed, thus ensuring the relevance and applicability of our results.

3. **Accurate Performance Evaluation:** The advanced hardware and software infrastructure of the CARC server allowed for more precise and reliable measurements of rendering time, image quality, and computational efficiency, providing a clearer picture of the engine's performance in a high-end computing environment.

4. **Verification of Scalability and Efficiency:** Porting to the CARC server enabled us to test the scalability of our engine effectively, verifying its efficiency and performance across different hardware configurations and under various computational loads.

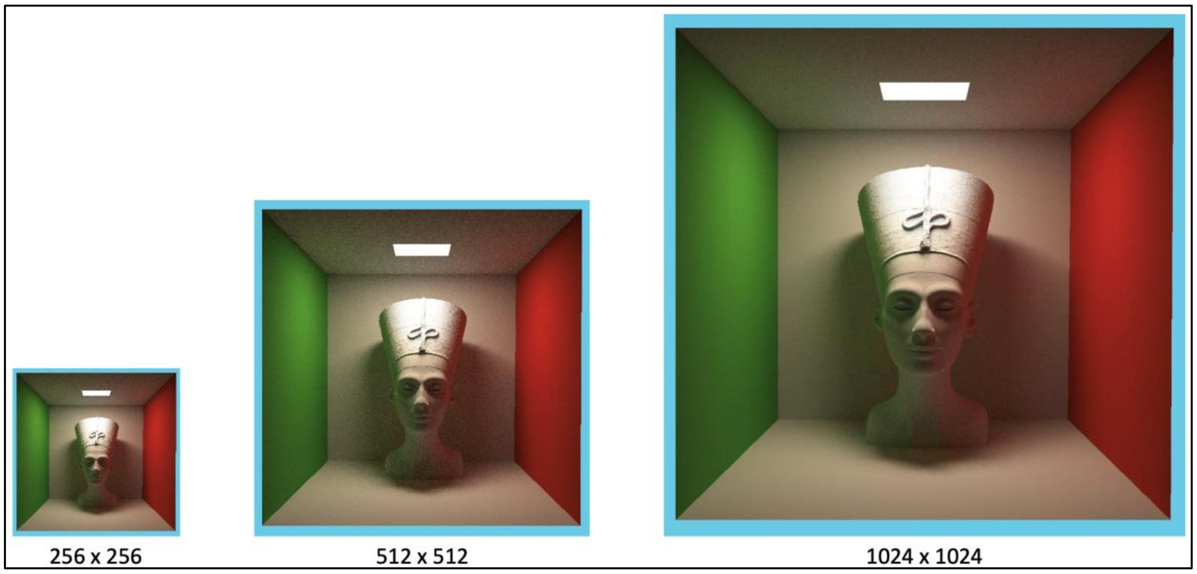
4.5 Results and Analysis

The results of the experiments demonstrated significant improvements in rendering times, particularly when utilizing higher thread counts on GPUs. Notably, the parallel rendering engine achieved substantial speedups compared to non-parallel rendering methods, especially in scenarios with high-resolution and high sample count. However, the experiments also highlighted some trade-offs between rendering speed and image quality, particularly at lower sample counts.

1. **Resolution Impact**: Higher resolutions increased rendering times but were effectively managed by the parallel processing capabilities, showing the scalability of the engine.

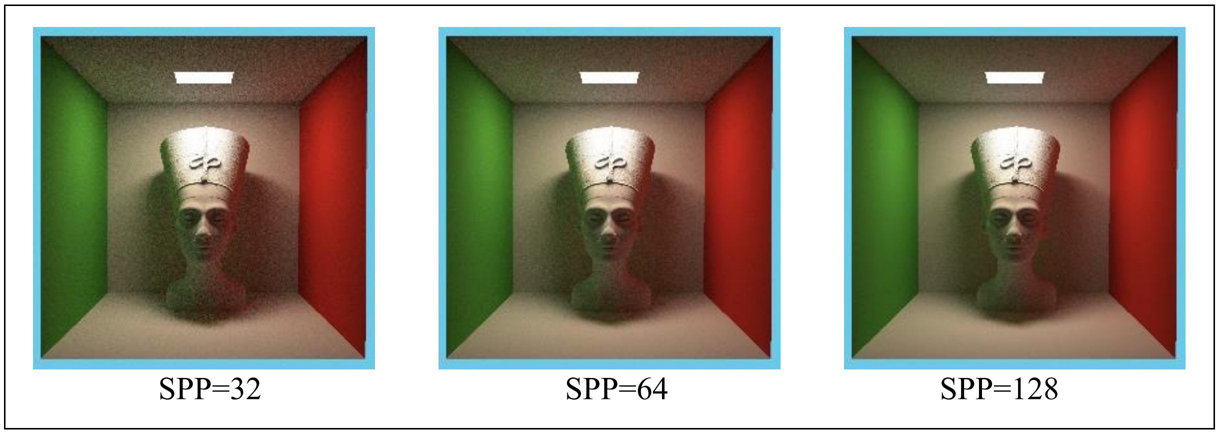
2. **Sample Count Impact**: Increasing the sample count improved image quality but also increased rendering times. The engine's parallel nature helped in mitigating this increase, particularly on GPUs.

3. **Thread Count Optimization**: Finding the optimal thread count for CPUs and GPUs was crucial for maximizing performance. The experiments indicated a point of diminishing returns, beyond which additional threads did not result in significant performance gains.



<Figure 9. Result Image by Resolution>

The image quality improves significantly as the resolution increases. At the lowest resolution (256x256), the details are quite blurry and pixelated. However, at the highest resolution (1024x1024), the image is much clearer, showing a detailed and smooth rendering of the model.



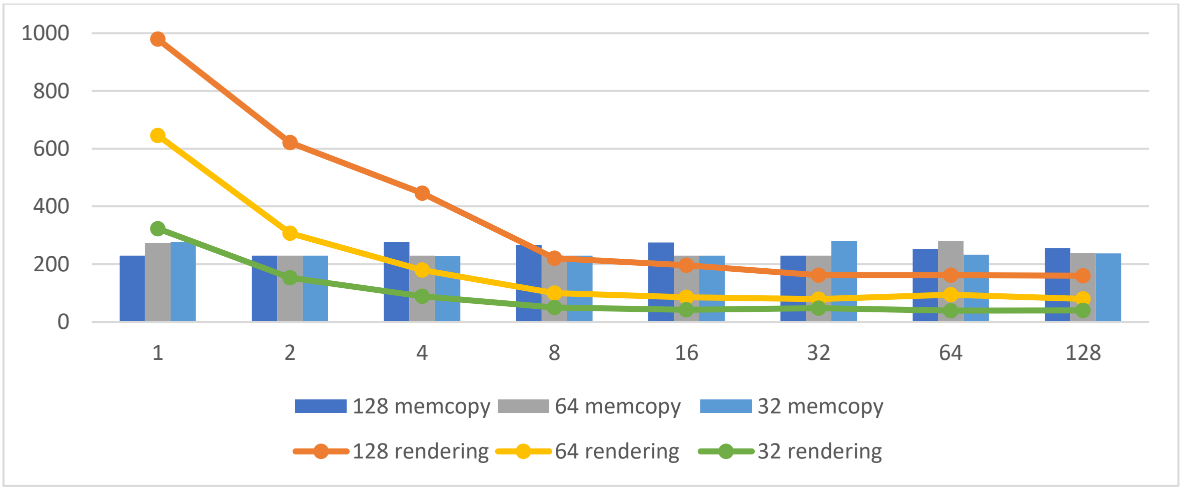
<Figure 10. Result Image by Sample Count>

In our experiments, the impact of varying the Samples Per Pixel (SPP) was evident, reinforcing a well-understood aspect of image rendering. As the SPP increased from 32 to 128, we observed a corresponding improvement in image smoothness and clarity, confirming the direct relationship between SPP and visual quality. This outcome, while anticipated, highlights the essential balance in rendering between achieving detailed imagery and managing computational load.

4.5.1. with OpenMP

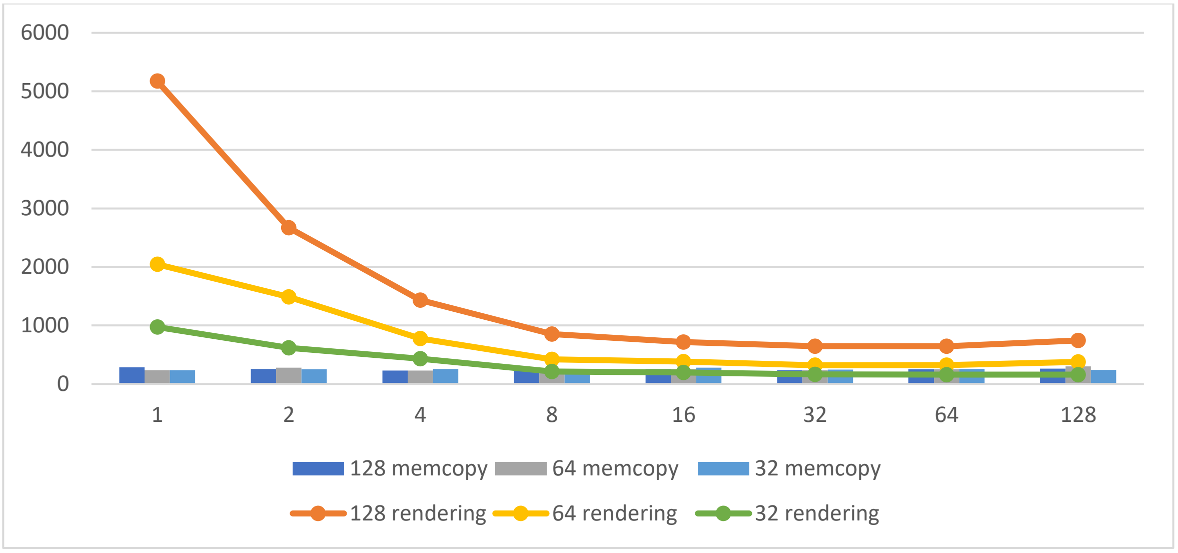
<Table 1. Test Result: Image size(256x256)>

SPP	128		64		32	
Time(s) / No. Thread	memcopy	rendering	memcopy	rendering	memcopy	rendering
1	229	979	274	646	277	323
2	230	621	229	307	229	153
4	277	446	229	180	228	88
8	267	221	229	100	229	50
16	275	196	230	85	230	42
32	230	162	230	79	279	47
64	252	162	280	94	233	39
128	255	160	239	80	237	40
256	out of Memory					



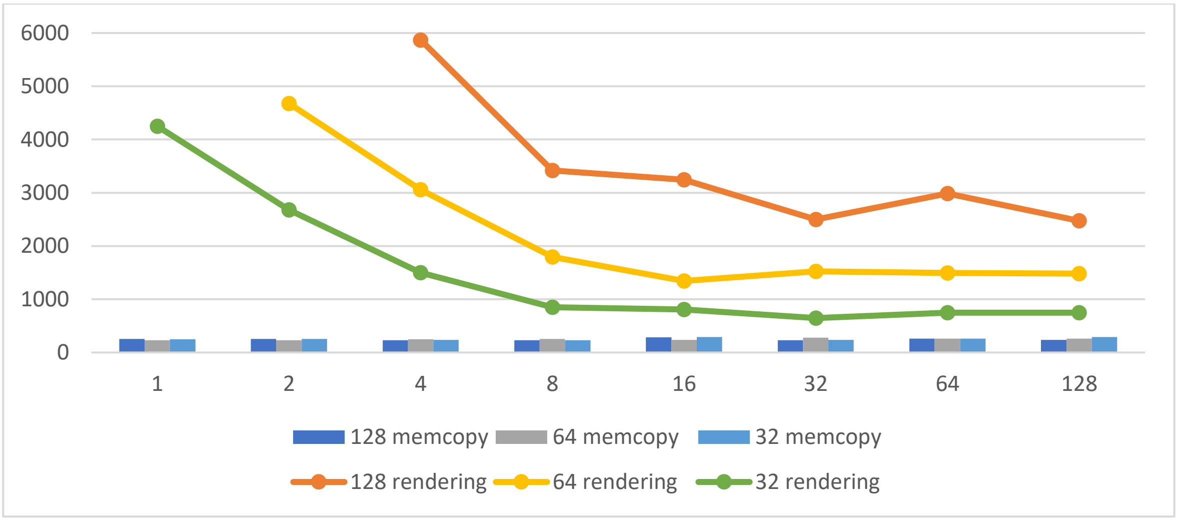
<Table 2. Test Result: Image size(512x512)>

SPP	128		64		32	
Time(s) / No. Thread	memcopy	rendering	memcopy	rendering	memcopy	rendering
1	284	5175	237	2045	236	973
2	255	2672	277	1488	249	617
4	228	1432	230	775	256	431
8	252	851	249	422	229	213
16	255	716	256	382	277	198
32	234	646	235	320	248	163
64	254	644	254	321	255	160
128	262	742	298	376	241	160
256	out of Memory					



<Table 3. Test Result: Image size(1024x1024)>

SPP	128		64		32	
Time(s) / No. Thread	memcopy	rendering	memcopy	rendering	memcopy	rendering
1	257	Time out	233	Time out	251	4250
2	254	Time out	233	4677	254	2678
4	233	5869	249	3059	238	1498
8	229	3417	254	1794	229	851
16	287	3242	234	1344	289	806
32	229	2498	278	1521	234	647
64	259	2984	259	1492	261	749
128	236	2473	262	1484	291	750
256	**out of Memory**					



The comprehensive dataset from our OpenMP experiments illustrates a transition in the rendering engine's performance characteristics depending on the resolution and SPP. At lower resolutions and SPPs, the engine behaves more like a memory-bound program, where performance is primarily limited by the speed of memory operations rather than computational throughput. This is indicated by the relatively small changes in rendering times despite variations in thread counts, suggesting that memory transfer speeds are the limiting factor.

As the resolution increases and with it the SPP, we observe a shift toward compute-bound behavior. In these scenarios, the rendering times extend significantly, and the engine's performance becomes more dependent on the computational capacity of the hardware. This shift is evidenced by the increased rendering times that correspond with higher resolutions and SPPs, where the engine requires more computational resources to process the additional data.

The performance plateau observed at thread counts above 8, when the CPU/task setting is factored in, indicates that the engine's scalability is maximized near the physical core count. However, as the rendering tasks become more computationally intensive with higher resolutions and SPPs, the engine's behavior aligns more closely with that of a compute-bound system, where the potential for performance improvement through parallel processing is greater, albeit still constrained by the system's computational resources.

This nuanced understanding of the engine's performance under different conditions is critical for optimization. It informs developers that while increasing parallelism can improve performance, the type of system bottleneck (memory-bound or compute-bound) will ultimately govern the effectiveness of such optimizations. It also underscores the importance of matching the engine's workload to the system's capabilities, particularly when dealing with high-resolution, high-fidelity rendering tasks that require significant computational power.

4.5.2. with CUDA

<Table 4. Test Result: Image size(256 x 256)>

Rendering time(s)

SPP / grid size	SPP 32	SPP 64	SPP 128	SPP 256	SPP 512
16x16	3	7	15	15	27
32x32	3	5	12	12	24
64x64	4	9	18	18	36
128x128	9	18	37	37	74
256x256	16	32	64	64	129
512x512	16	32	64	64	129
1024x1024	16	33	65	65	129

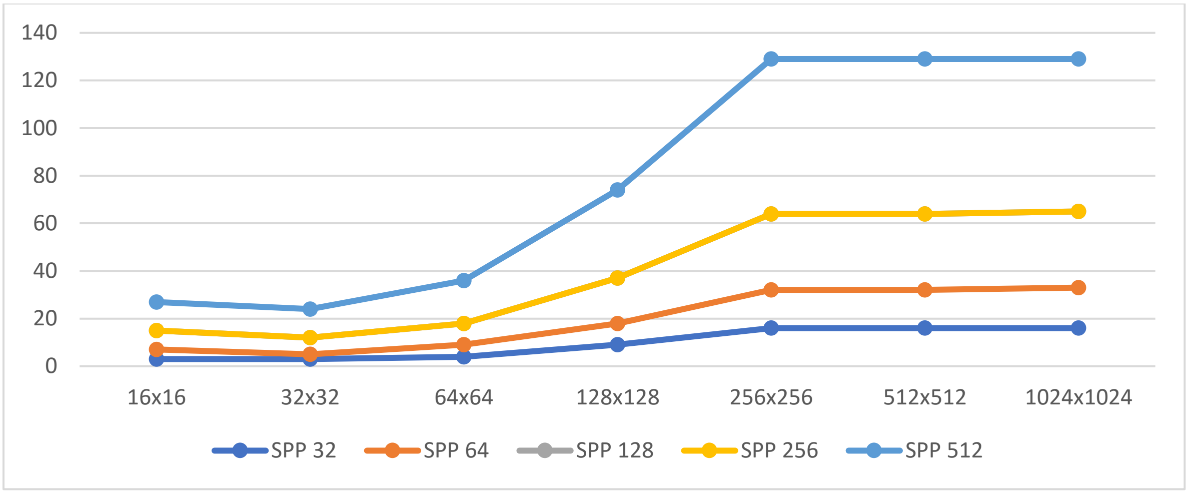


Table 5. Test Result: Image size(512 x 512)

Rendering time(s)

SPP / grid size	SPP 32	SPP 64	SPP 128	SPP 256	SPP 512
16x16					
32x32	6	13	26	52	104
64x64	6	12	24	49	97
128x128	9	19	38	76	152
256x256	20	41	82	165	330
512x512	36	72	143	287	574
1024x1024	36	72	144	288	575

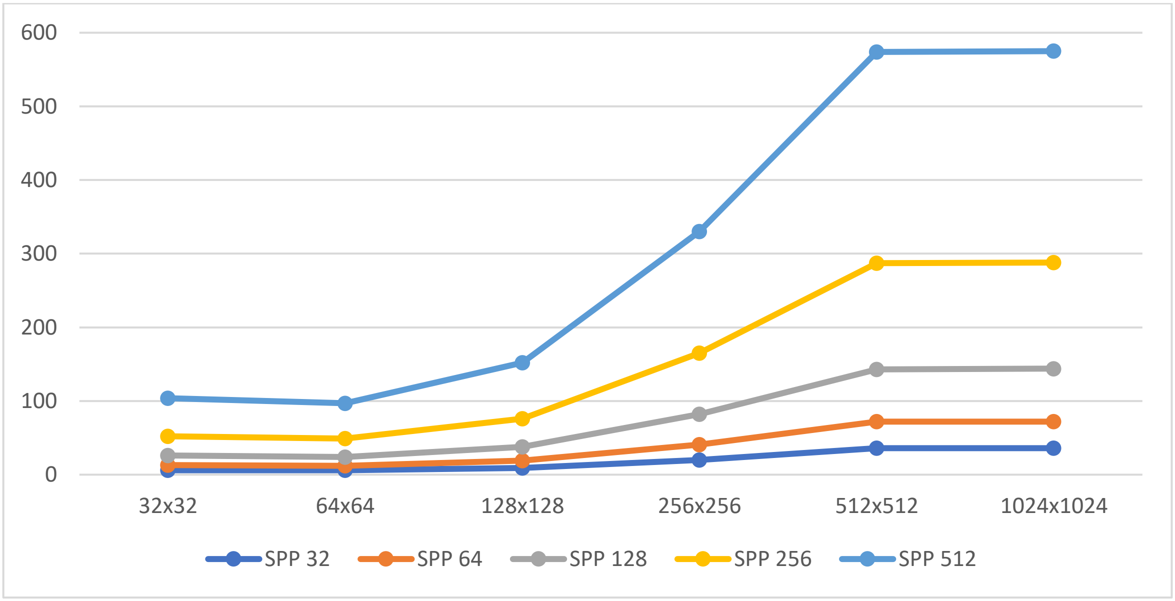
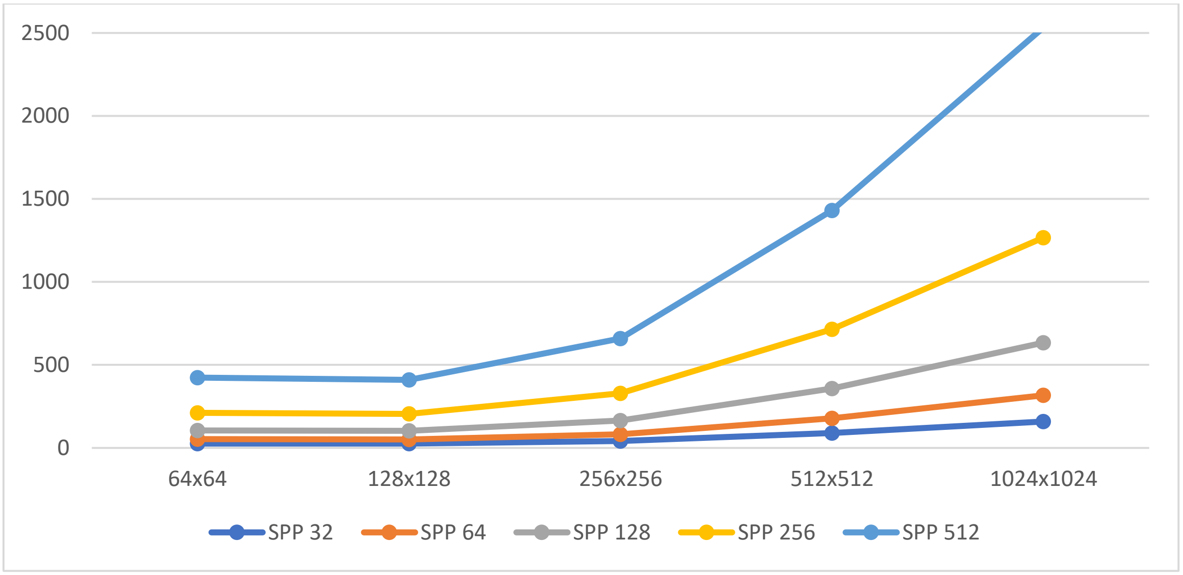


Table 6. Test Result: Image size(1024 x 1024)

Rendering time(s)

grid size \ SPP	SPP 32	SPP 64	SPP 128	SPP 256	SPP 512
16x16					
32x32					
64x64	26	53	105	211	423
128x128	26	51	103	205	410
256x256	41	82	165	329	659
512x512	89	179	358	715	1430
1024x1024	159	317	634	1267	2533



The CUDA-based rendering experiments show a complex relationship between the grid size, the samples per pixel (SPP), and the rendering time for different image resolutions. The data indicates that for each resolution, there is an optimal grid size that minimizes rendering time, and deviating from this optimal size leads to increased rendering times.

For a 256x256 resolution image, a grid size of 32x32 appears to offer the best performance. Similarly, for 512x512 resolution, a 64x64 grid size is optimal, and for 1024x1024 resolution, a 128x128 grid provides the best results. These optimal grid sizes suggest that there is a proportionality factor between the resolution and the grid size that yields the most efficient use of the GPU's parallel processing capabilities.

The observed trend where larger grid sizes result in longer rendering times can be attributed to several factors:

1. **Overhead of Managing Larger Grids:** As the grid size increases, the overhead associated with managing a larger number of blocks and threads may outweigh the benefits of parallel processing. This overhead can include factors such as increased time for thread synchronization and less efficient use of shared memory.

2. **GPU Architecture Constraints:** GPU architecture is designed to handle a certain number of active threads and blocks concurrently. If the grid size is too large, not all threads may be active at once, leading to underutilization of the GPU's computing resources.

3. **Memory Access Patterns:** Larger grid sizes can lead to less optimal memory access patterns, where the memory bandwidth becomes a bottleneck. This is particularly true for global memory accesses, which are slower compared to shared memory accesses. As a result, memory transfer times can increase, contributing to longer overall rendering times.

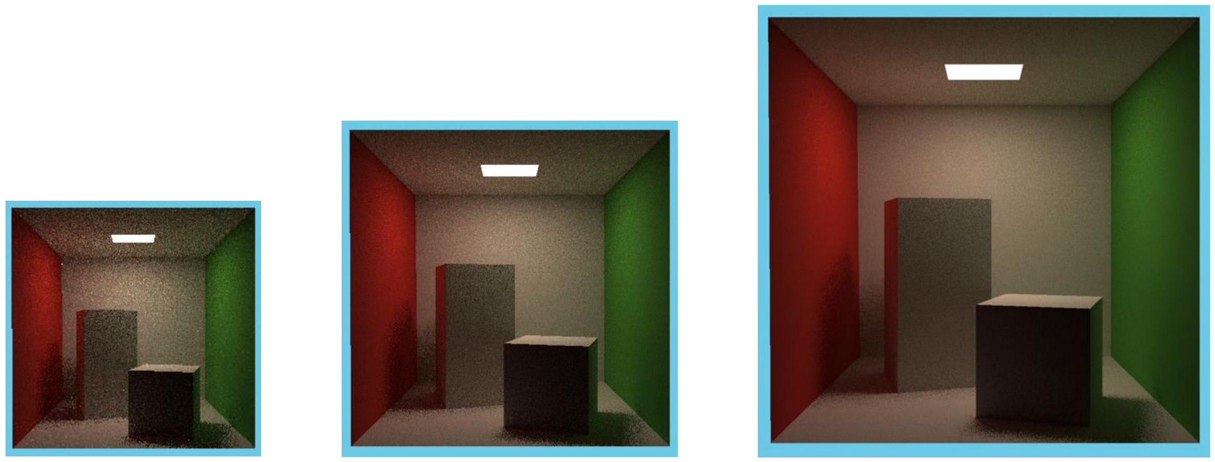
We opted to exclude the object loading times from the performance analysis. This decision was made because the loading times were consistent with those observed in the OpenMP experiments, due to the use of identical loading modules across both sets of tests. Given that the loading times had minimal variability and did not contribute meaningfully to the differences in performance outcomes, we chose to focus our analysis on the rendering performance itself, which provided more significant insights into the efficiency and scalability of our rendering engine in a CUDA environment.

The slight improvements in rendering speed at the optimal grid sizes for each resolution highlight that the efficient use of CUDA cores and memory is highly dependent on matching the problem size to the hardware capabilities. When the grid size exceeds the optimal point, the additional computational resources do not translate into performance gains, potentially due to increased latency and reduced occupancy in GPU processing units.

4.6 Conclusions from Experiments

Our comprehensive experiments with both OpenMP and CUDA have revealed a consistent trend: CUDA markedly outperforms OpenMP in terms of rendering times across all test cases. The disparity in performance becomes increasingly significant at higher resolutions and samples per pixel (SPP), suggesting that CUDA's architecture excels at managing the demands of computationally intensive rendering tasks.

During our experiments, we encountered difficulties with the efficient loading of large-scale objects using the CUDA framework. This prompted us to modify our approach, opting to measure results using a basic box-shaped object as a standardized test model. Despite this simplification, the rendering times achieved by the CUDA version remained impressively superior to those of OpenMP.



<Figure 11. Result Image of CUDA and openMP>

<Table 7. Compare the Best Case>

Time(s)

Resolution	SPP 32		SPP 64		SPP 128	
	openMP	CUDA	openMP	CUDA	openMP	CUDA
256x256	20	3	41	5	82	12
512x512	83	6	147	12	237	24
1024x1024	232	26	465	51	983	103

The lower rendering times of CUDA, even with a simplified object, underscore the efficiency of GPU parallelism in rendering processes, which significantly outperforms CPU-based parallelism facilitated by OpenMP.

It is important to recognize that while CUDA offers exceptional performance, it requires a compatible GPU. Conversely, OpenMP is more widely accessible due to its reliance on CPU processing. For scenarios where rendering speed is paramount and suitable GPU resources are available, CUDA is evidently the optimal choice. However, OpenMP remains a valuable alternative, capable of delivering substantial performance improvements over single-threaded processes in environments where GPU resources are constrained or unavailable.

The inclusion of this comparative table in our report visually underscores the performance advantages of using CUDA for rendering tasks, particularly as the requirements for resolution and realism, indicated by higher SPP, escalate. Despite the challenges posed by object loading complexities, the consistent superiority of CUDA's rendering times reinforces its efficacy as a solution for high-performance rendering applications.

5. Conclusion and Future Work

5.1 Conclusions

Our comparative analysis of OpenMP and CUDA for parallel graphics rendering has yielded significant insights. The experimental data conclusively demonstrates that CUDA offers superior performance over OpenMP across all scenarios, with the gap widening as the resolution and samples per pixel (SPP) increase. This is attributed to the more effective parallelism of GPUs compared to CPUs and CUDA's efficient architecture for handling computationally intensive tasks.

The optimal grid sizes identified for different resolutions in the CUDA environment suggest a clear proportionality that maximizes GPU utilization and minimizes rendering times. However, the increase in rendering times with larger grid sizes beyond the optimal point underscores the importance of tuning parallel execution parameters to the hardware's capabilities.

In practice, this means that to achieve efficient parallel processing, one must properly control the number of threads, optimize memory access patterns, and minimize overhead and synchronization costs. These principles are pivotal for enhancing performance in parallel programming.

5.2 Future Work

As we move ahead, there are several possibilities for future work that we can consider.

1. **Algorithm Optimization:** Further refinement of the rendering algorithms to enhance efficiency, particularly in memory-bound scenarios, can lead to performance improvements.

2. **Hardware Utilization:** Exploring the use of more advanced or different GPU architectures could provide additional insights into the scalability and efficiency of the rendering engine.

3. **Thread Management:** Developing more sophisticated techniques for managing thread creation, scheduling, and synchronization could mitigate the diminishing returns observed with higher thread counts.

4. **Dynamic Grid Sizing:** Implementing a dynamic grid sizing algorithm that adjusts grid sizes in real-time based on workload could optimize rendering times across various resolutions and SPPs.

5. **Hybrid Parallelism:** Investigating the potential of hybrid models that combine both CPU and GPU resources for parallel rendering tasks may offer a balance between accessibility and performance.

6. **Toolchain Improvements:** Enhancing the development toolchain to streamline the porting process from development environments to high-performance servers like CARC could facilitate more rapid iteration and testing.

By addressing these areas, future projects can build on the foundations laid by this research to push the boundaries of parallel rendering performance further. The goal will always remain to balance the computational load with system capabilities while aiming for the highest possible visual fidelity and rendering speed.

6. References

[1] “OpenGL shading language”, Rost, Randi J; Licea-Kane, Bill ; Ginsburg, Dan; Kessenich, John; Lichtenbelt, Barthold; Malan, Hugh; Weiblen, Mike, Pearson Addision Wiseley; 2021

[2] “Computer Graphics Programming in OpenGL with C++”, Gordon, V.scott, Bloomfield, New jersey: Mercury Learning & Information; 2020

[3] OpenGL, https://www.opengl.org, accessed 2023-10-20

[4] KHORONOS group, https://www.khronos.org/opengl, accessed 2023-10-20

[5] “PATH TRACING: A NON-BIASED SOLUTION TO THE RENDERING EQUATION”, ROBERT CARR AND BYRON HULCHER

[6] “Real-Time Bidirectional Path Tracing via Rasterization”, Yusuke Tokuyoshi∗ Square Enix Co., Ltd. Shinji Ogaki† Square Enix Co., Ltd.

[7] “An Improved Illumination Model for Shaded Display”, Turner Whitted, 1980

[8] “Distributed Ray Tracing”, Cook, Porter, Carpenter, 1984

[9] “The Rendering Equation”, James Kajiya, 1986 • “Bidirectional Path Tracing”, Eric Lafortune, Yves Willems, 1993

[10] “Rendering Caustics on Non-Lambertian Surfaces”, Henrik Wann Jensen, 1996

[11] “Metropolis Light Transport”, Eric Veach, Leonidas Guibas, 1997

[12] “A Reflectance Graphics”, ROBERT L. COOK Lucasfilm Ltd. and KENNETH E. TORRANCE Cornell University

[13] “Real-Time Rendering Fourth Edition”, Tomas Akenine-M¨oller Eric Haines Naty Hoffman Angelo Pesce , Michal Iwanicki S´ebastien Hillaire; November 5, 2018