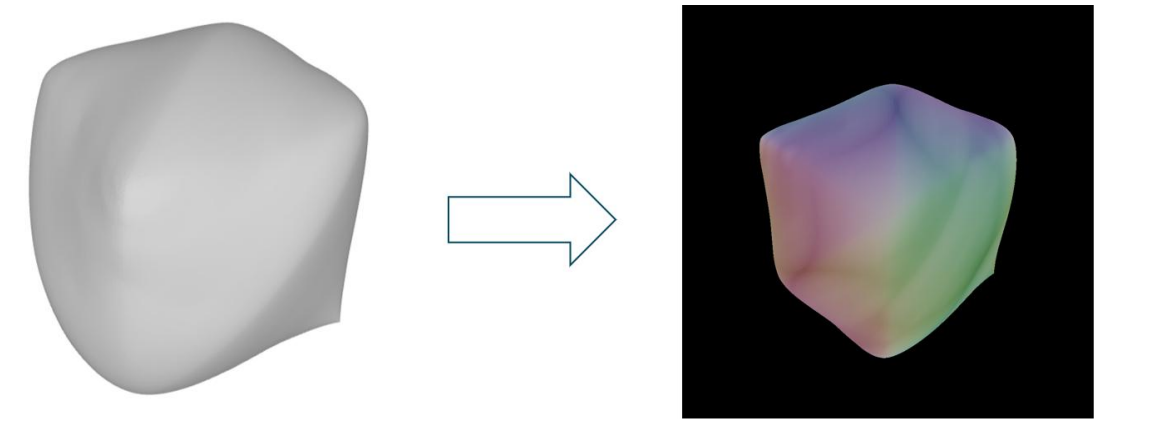
Ray Tracing using C++ and CUDA

Sanket Kumar Nayak

Abstract: Ray tracing is a rendering technique that simulates the way light interacts with objects to create realistic images. This technique computes shadows, reflections, and refractions based on light behavior. During my Master's project at the University of Houston, I got an opportunity to compute ray tracing using GPU-based CUDA (parallel processing). For a particular example, C++ took 38 seconds whereas CUDA could complete it in 0.6 milliseconds (a massive improvement!)

Introduction: Ray tracing is a rendering technique that models how light interacts with objects to generate realistic images. The goal of this project was to generate an image where objects are properly lit and cast accurate shadows based on given light sources.

The program took an OBJ file as input, containing 3D object geometry along with details about light sources. This data defined the objects' structure, material properties, and light interactions. The object described could be of any shape or size, with larger objects increasing the complexity of the calculations. The output was a rendered image that accurately displayed the objects with proper lighting and shadows, producing a far more realistic visual representation.



The initial Object from in an OBJ file

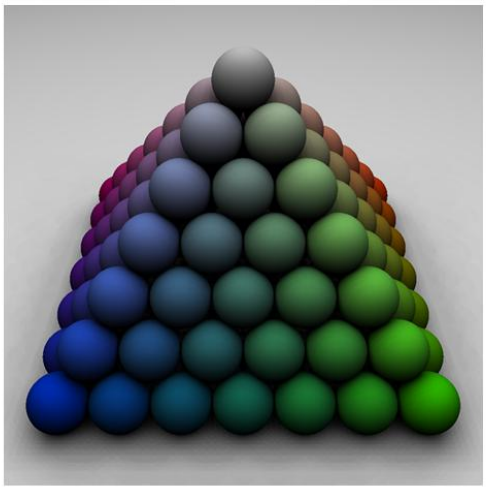
The Object after being projected through Ray Tracing.

Figure 1: A Visual Representation of Ray Tracing

Methodology: The initial implementation was done using C++ on a CPU, where each light ray's interaction with objects was calculated sequentially. However, this approach proved inefficient, as each pixel required multiple ray-object intersection checks, leading to extremely long execution times—often taking minutes, or even hours to compile and run. Debugging and refining the algorithm became impractical due to these delays.

To overcome these limitations, the project was migrated to CUDA, leveraging GPU-based parallel processing. For this project, an Nvidia RTX2060 GPU was used. The transition involved understanding CUDA's architecture, breaking down ray tracing computations into parallel tasks, optimizing memory usage, and refining the implementation to ensure numerical stability. Instead of processing each light ray sequentially, CUDA enabled thousands of GPU threads to execute calculations simultaneously. This shift drastically improved performance, reducing execution time from hours to less than a millisecond. Additionally, the new implementation allowed for more complex scenes and higher resolutions without a major performance hit.

Compared to the original CPU-based implementation, CUDA offered a massive speed improvement, making real-time or near-real-time rendering feasible. In an example, an image was rendered in 38 seconds on a 10th Generation Intel CPU, while it took 0.6 milliseconds on the Nvidia RTX2060 GPU. Faster computation also made debugging and iteration significantly easier, enhancing development efficiency.



	C++ (CPU Based)	CUDA (GPU Based)
Time for the image	38 seconds	0.6 milliseconds
Time per pixel	38 microseconds	0.6 nanoseconds

For these calculations, the CPU used is an Intel Core i7-10750 with 6 cores, and the GPU used is an Nvidia RTX2060.

Figure 2: An image rendered through Ray Tracing, and the time taken to render with each implementation

Challenges Faced: Several challenges arose during the implementation. A large part of the program included calculating shadows based on the light sources, requiring additional ray tracing logic that increased computational load and introduced multiple bugs. Learning CUDA for the first time also posed difficulties, especially in terms of memory management and debugging parallel execution. Additionally, floating-point precision errors led to small but noticeable artifacts in the final rendering, affecting shadow accuracy and light behavior.

To address these issues, shadow calculations were optimized to minimize redundant computations while maintaining accuracy. Over time, CUDA's workflow became easier to manage through practice and experimentation. Floating-point precision errors were mitigated by adjusting mathematical logic to improve numerical stability. Despite these challenges, the project proved to be a valuable learning experience, demonstrating the power of GPU-based computation and opening the door for future work in real-time rendering and complex graphics applications.

CUDA, however, is not universally applicable. While it excels in parallel computing tasks like ray tracing, it is less efficient for sequential processes that require frequent communication between the CPU and GPU. This limitation makes it unsuitable for certain applications where dependencies between calculations are high. GPUs work by running thousands of small threads in parallel, handling multiple calculations simultaneously, which is why they outperform CPUs for tasks that can be split into independent operations.

Conclusion: Understanding the advantages and trade-offs is crucial when deciding whether to use CUDA or stick with a traditional CPU-based approach. Overall, the project was a rewarding experience, enhancing both technical skills and understanding of GPU-based computation, which proved useful in future projects.

CUDA and GPUs are not just useful for graphics - they are also a game-changer in AI applications. GPUs speed up large-scale data processing and make real-time AI applications possible. That's why they are widely used in fields like machine learning, computer vision, and scientific research.