Introduction

This CUDA-based application harnesses the power of GPUs to perform a parallel ray tracing simulation, aimed at generating a visualization of a three-dimensional scene defined by geometric objects and light sources. By porting the serial implementation of the ray tracing algorithm to the massively parallel architecture of GPUs, the program efficiently computes the intersections of rays with objects in the scene to determine the color and brightness of each pixel in the output image. The core of the simulation, '**rayTraceKernel**', runs on the GPU and executes in parallel across thousands of threads.

Approach + Optimizations

My primary objective was to simulate ray tracing in a parallelized manner, fully utilizing the GPU's capability to handle thousands of threads simultaneously. The ray trace kernel function operates on the GPU, where each thread simulates a single ray or a batch of rays through the scene. This design allows for massive parallelism, significantly reducing the computation time compared to serial processing. On the host side, I focused on setting up the scene, initializing data structures, and managing memory transfers between the host and the device. The host also initiates the kernel launches and handles the final output, including saving the rendered image to a file.

The following optimizations were made in order to port the code efficiently onto the GPU:

- **Efficient Data Handling**: I transitioned from using multiple scalar variables to employing arrays for the vectors used within the ray trace kernel. This change streamlined memory access patterns and reduced the overhead associated with managing numerous individual variables.
- **Pre-check for Ray Validity**: Before proceeding with computationally intensive dot product calculations, I introduced a preliminary check to ensure that the rays (**W_x** and **W_z**) are within the desired range. This optimization helps to avoid unnecessary calculations for rays that won't contribute to the final image.
- **Compiler Optimizations with Fast Math**: By utilizing the **--use_fast_math** compiler flag, I allowed the compiler to employ faster, less precise math operations. This optimization can lead to significant performance gains, especially in floating-point arithmetic-intensive applications like ray tracing.
- **Strided Access Pattern**: To efficiently distribute the workload across threads, I implemented a stride length approach, enabling each thread to process multiple rays in separate iterations. Specifically, I designed the kernel execution such that it takes 10 iterations for a thread to go through all the rays, thereby optimizing memory access and computational balance across the GPU.
- **Precision Considerations**: I opted for single-precision floats instead of double-precision to reduce memory bandwidth and computation time. Given that ray tracing can tolerate some level of imprecision without significantly degrading the visual quality, this trade-off was deemed acceptable for performance gains.
- **Separate Kernel for Random Number Initialization**: Instead of initializing **curand** states within the main loop, I employed a dedicated kernel for this purpose. By pre-initializing these states once and reusing them in the ray trace kernel, I minimized the overhead and ensured consistent random number generation across threads.

Results

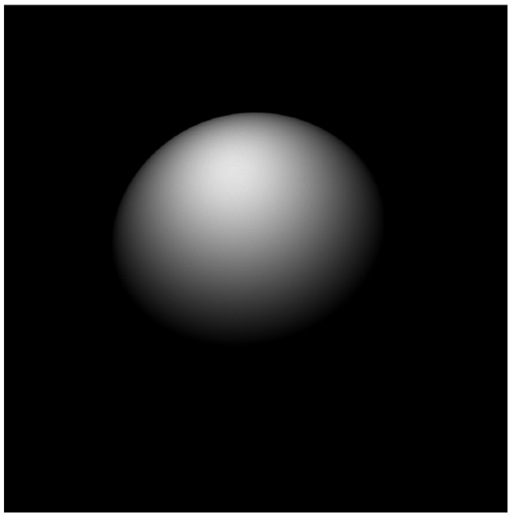
Figure 1: Single Thread Ray Trace

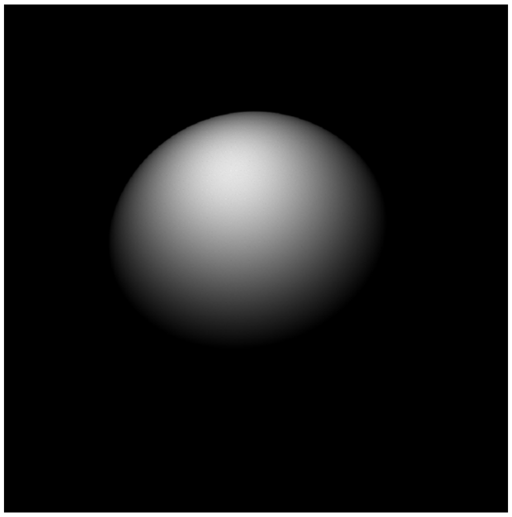
Figure 2: Multicore (nt = 16) Ray Trace

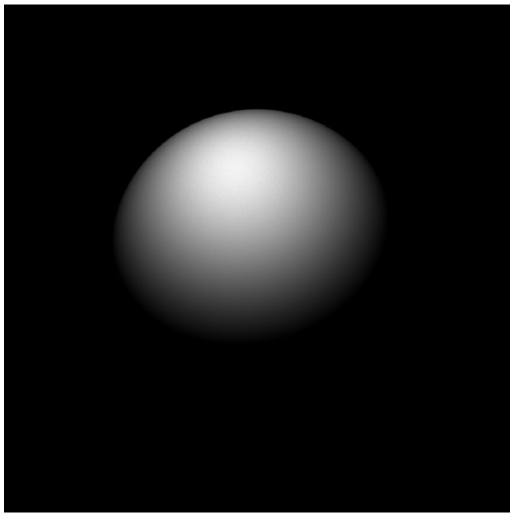
Figure 3: CUDA Ray Trace

Solution Type	**Fastest Time (seconds)**
Serial	362.00
Multicore (nt = 16)	30.00
Multicore (nt = 48)	10.00
GPU (Kernel Call Time)	0.152