MPCS 51087 Project 2, Milestone 2
GPU Ray Tracing with CUDA

Sonia Sharapova

1 Introduction

PLOT COMPARISONS ON LAST PAGE

The following report outlines a GPU Ray Tracing algorithm with CUDA and compares its execution time with OpenMP, another parallelization strategy, and a serial implementation. The goal of the programs is to render a 3D reflective sphere illuminated by a single light source.

2 Implementation

Initialization

For each implementation, the functions were run on a 1000 x 1000 grid, for 1 billion light rays. The 1000 x 1000 grid was initialized with zeros and was iteratively updated at every calculation.

2.1 Serial

The serial implementation for this was used as a baseline benchmark to measure the performance of parallelization techniques. For the serial implementation, I utilized helper functions and the rand() method as they would not be impacted by threading.

Without any parallelization, 1 billion rays took 5.5 minutes to run on the midway server.

Usage:

```
gcc -fopenmp -O3 -g -o serial ray_tracing_serial.c -lm
```

```
./serial
```

2.2 Parallel

The serial implementation was further enhanced through implementing OpenMP and parallelizing the ray tracing function for a multi-core CPU. To ensure thread safety, I performed all the vector calculations locally rather than through the use of helper functions. For the computation of the randomized vector, I utilized the rand_r() function for thread safety.

On 3 threads, the program had an execution time of 2.26 minutes and a speedup of 2.56 when compared to the serial implementation, and on 16 threads the time was 26.5 seconds with a speedup of 13.16 compared to the serial implementation.

Usage:

```
gcc -fopenmp -O3 -g -o parallel ray_tracing_parallel.c -lm
```

```
./parallel 3
```

```
./parallel 16
```

2.3 CUDA: GPU Multi threading

The CUDA implementation for this problem consisted of mapping the ray-object intersection calculations to GPU threads, which allowed for further parallel processing of the multiple rays.

The scene was decomposed into elements processed in parallel by the GPU. This device specific code included computationally intensive tasks in the ray tracing function such as ray generation, vector computations, and shading/lighting calculations. The host-side code is responsible for initialization of the final grid, setup tasks, and memory allocation on the GPU. It also ensures that data is safely transferred between the host and the device. The results are then gathered back on the host for final image composition and output on the grid.

For the computation of the randomized vector, I used CUDA's cuRAND function to ensure thread-safety. With cuRAND, each thread can independently sample the directions for the vector which will not lead to conflicts during parallelization.

The CUDA solution was incredibly efficient with a completion time of 0.459s and a speedup of 758.32 times compared to the serial implementation.

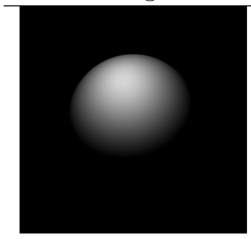
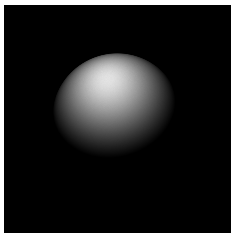
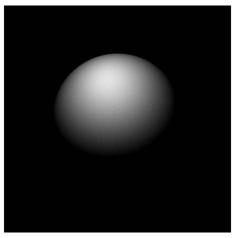
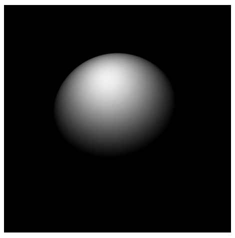
Usage:

```
nvcc -O3 -use_fast_math -o ray_tracings ray_tracing_cuda.cu -arch=sm_75
```

```
./ray_tracings
```

3 Data Visualization and Performances

The computed results of the n x n grid were stored in a file and visualized with python code. They simulared the visualization of the sphere when all the rays have been computed. The states were plotted using matplotlib.

Image	Execution Time (s)	Approach
	348.80, (5.48 min)	Serial Implementation
	136.093, (2.26 min)	Parallel on 3 Threads
	26.50	Parallel on 16 Threads
	0.459	CUDA Solution

4 Conclusion

The time comparisons between a serial implementation, parallelization with OpemMP, and parallelization over GPUs with CUDA. In every implementation, the sphere was correctly constructed, but with various parallelization techniques, the execution time differed greatly. The CUDA implementation on GPUs had the best and fastest performance. For future, use this performance could be further optimized to perform even better.