Ray Tracing on a Sphere

Performance Analysis of CUDA Performance

Sonia Sharapova

Introduction:

This report outlines a GPU Ray Tracing algorithm with CUDA and compares its execution time with a serial version, parallel implementation, and across different GPUs using CUDA.

Methodology

Serial Implementation

The serial version of the ray tracing algorithm processes each pixel one at a time. For each pixel, it calculates the light's path, including reflections, refractions, and shadows, to determine the final color of that pixel. The algorithm iterates over all pixels in the image, rendering each scene sequentially. This approach, while straightforward, does not leverage modern multi-core processors' capabilities, leading to long rendering times for complex scenes or high-resolution images.

Parallel Implementation

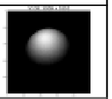
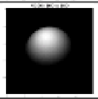
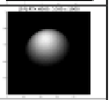
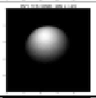
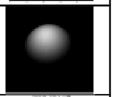
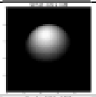
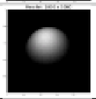
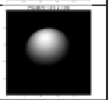
To improve the performance of the serial implementation, OpenMP was implemented to parallelize the code and distribute work for the ray tracing function for a multi-core CPU. To ensure thread safety, the vector computations were performed locally, without the use of helper functions. This modification also increased the runtime further as the functions did not have to be called at every step. The rand_r() function was used to ensure thread safety across computations and updates.

CUDA Implementation

To further optimize the computation time, CUDA was invoked and compared across a variety of available GPUs. The CUDA implementation parallelizes the ray tracing process by assigning the computation of each pixel (or a small group of pixels) to separate threads on the GPU. This allows for the simultaneous calculation of multiple pixels, drastically reducing the total rendering time. The CUDA model divides the image into blocks and threads, where each thread calculates the color for a single pixel, and each block contains a group of threads working together. For consistency, all computations were performed on 1000 blocks. The number of threads per block varied according to GPU availability and optimal performance. This approach takes full advantage of the GPU's architecture, designed for high-throughput parallel computations.

Performance Comparison

To compare the performance of the serial and CUDA implementations, we measure the rendering time of a sphere with varying GPUs, grid dimensions, thread number (per block), and single vs. double precision. The performance metrics are measured in the total time the program took to execute the ray tracing function and the total program computation time. These standardizations ensured a fair comparison between the CPU and GPU processing capabilities.

Proc	Grid	Time (SP)	K Time (SP)	Time (DP)	K Time (DP)	Blk/TPB	Cores	Samples	Image
A100	1000^2	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
A100	100^2	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
V100	1000^2	0.299	0.179603	0.839	0.720831	256 Threads	1	14, 926,918,271	
V100	100^2	0.302	0.177101	0.858	0.730012	512 Threads	1	14, 926,417,231	
RTX6000	1000^2	0.301	0.204290	9.950	9.758760	512 Threads	1	14, 926,918,283	
RTX6000	100^2	0.287	0.188481	12.624	12.553495	512 Threads	1	14, 926,734,271	
CPU Serial	1000^2	348.807	N/A	409.827	N/A	N/A	N/A	Timed-out	
CPU Serial	100^2	336.183	N/A	396.787	N/A	N/A	N/A	Timed-out	
CPU OMP	1000^2	16.385	N/A	15.436	N/A	On 32 threads	32	14, 927,936,827	
CPU OMP	100^2	16.685	N/A	16.730	N/A	On 32 threads	32	14, 927,842,972	
> 1 GPU^{+}	1000^2								
> 1 GPU^{+}	100^2								

Optimization Strategies

Helper/Built-in Functions

Throughout this project, I observed a variety of techniques that reduced the computation time. One of the best optimizations I observed was bringing the helper functions I had into the driver function. This included removing built-in math functions, such as pow(), and calculating everything manually. Having these computations done locally reduced the running time by 0.4s. Surprisingly, I found that the fabs() function performed 0.2s better than the manual computation in single precision, but this had a worse performance in double precision.

Writing to File

Writing data to a binary file rather than a CSV file often had a shorter runtime compared to writing to a text file. This was most likely because binary files are a more efficient data representation and result in a reduced file size. Because we were generating and storing large amounts of data, this reflected in the total runtime and performance of the program.

Double vs. Single Precision

The precision of computation, single or double, significantly affected both performance and memory usage. Single precision using 32-bit floating-point numbers had better execution speeds compared to double precision using 64-bit floating-point numbers. This makes sense because single-precision uses less memory in the program.

Different Flags

Flags such as -use_fast_math increased the speed by 0.1s, which was not too big but still something interesting.

Possible Further Optimizations:

Integrating MPI with CUDA for distributed parallelism would further optimize this program. This would scale CUDA beyond the GPU resources of a single node, facilitating distributed parallelism across multiple GPUs and nodes.

Results

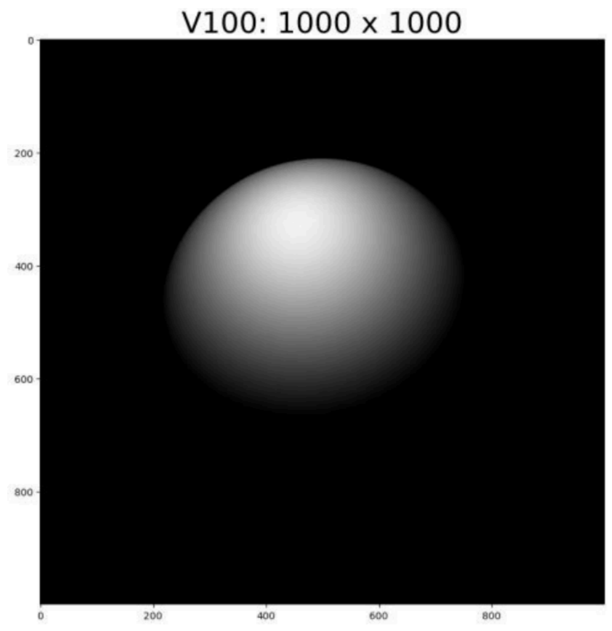
The CUDA implementation demonstrated a significant improvement in rendering times compared to the serial version. For simple scenes, the speedup can be substantial, often in the order of tens to hundreds of times faster, depending on the scene complexity and resolution. As the scene's complexity increases, the advantage of CUDA becomes even more pronounced, showcasing the scalability of parallel processing for graphics rendering.

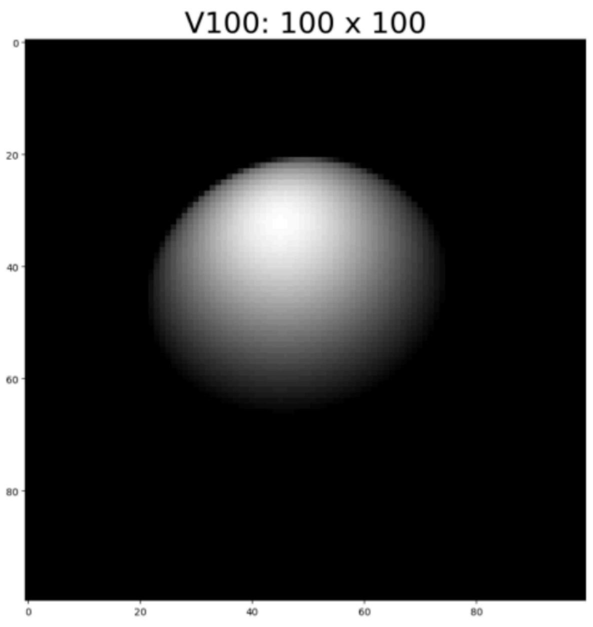
Conclusion

By using the GPU's parallel processing capabilities, the CUDA implementation offers dramatically reduced rendering times, making real-time ray tracing and the rendering of complex scenes more feasible. Furthermore, optimization strategies for both serial and CUDA implementations can yield additional performance gains, emphasizing the importance of both algorithmic efficiency and hardware utilization in high-performance computing applications.

This report underscores the transformative potential of GPU computing in graphics rendering and other computationally intensive tasks, highlighting the shift towards parallel processing in modern computing environments.

Sample Larger Image:


V100: 1000 x 1000
0
200
400
600
800
0
200
400
600
800


V100: 100 x 100
0
20
40
60
80
0
20
40
60
80