

Project 2 – Final Specification

Introduction

This CUDA and MPI-integrated application extends the capabilities of parallel computing to leverage multiple GPUs for a distributed ray tracing simulation. The enhanced program not only visualizes three-dimensional scenes with intricate geometric and lighting details but also scales the computation across various high-performance GPU models, including A100, V100, and RTX6000. By combining CUDA's parallel processing on individual GPUs with MPI's cross-device communication, the simulation achieves substantial performance gains. Performance metrics have been rigorously obtained from both a single GPU setup and a multi-GPU environment orchestrated via MPI. Additionally, the application's efficiency has been benchmarked on a Midway 3 Cascade Lake node, providing insightful comparisons between serial execution and a parallelized strategy utilizing 16 cores.

Algorithmic Approach

Algorithm Optimizations:

- **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 (\mathbf{W}_x and \mathbf{W}_z) are within the desired range. This optimization helps to avoid unnecessary calculations for rays that won't contribute to the final image.
- **Saving in Binary Form:** The '**save_to_file**' function saves time by writing data in binary mode directly to the file, which is faster - than writing as formatted text because it skips the overhead of converting the binary data to human-readable text form.

GPU Optimizations:

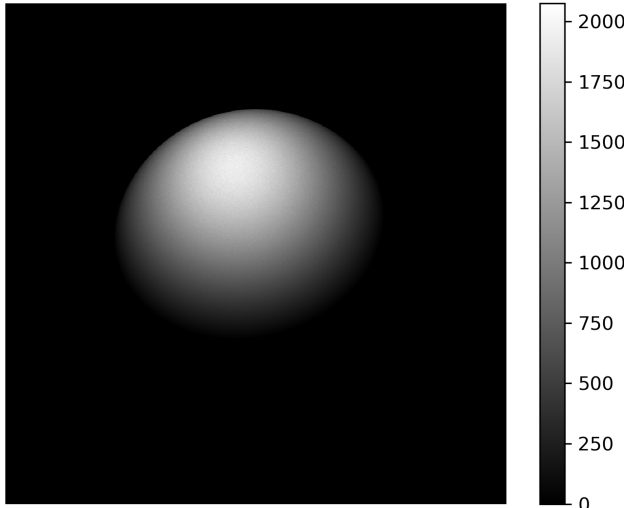
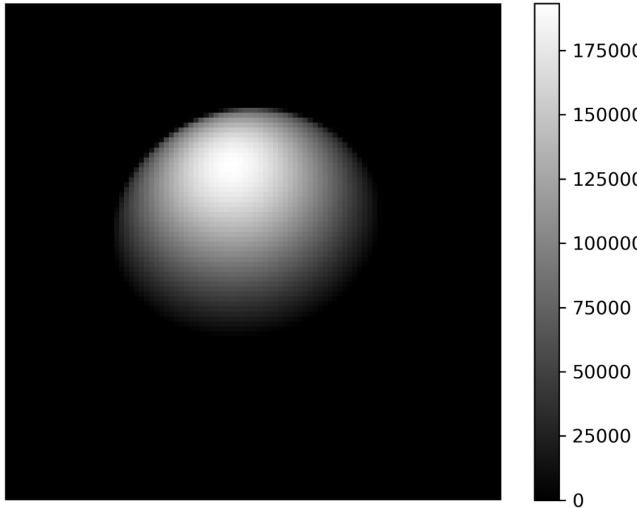
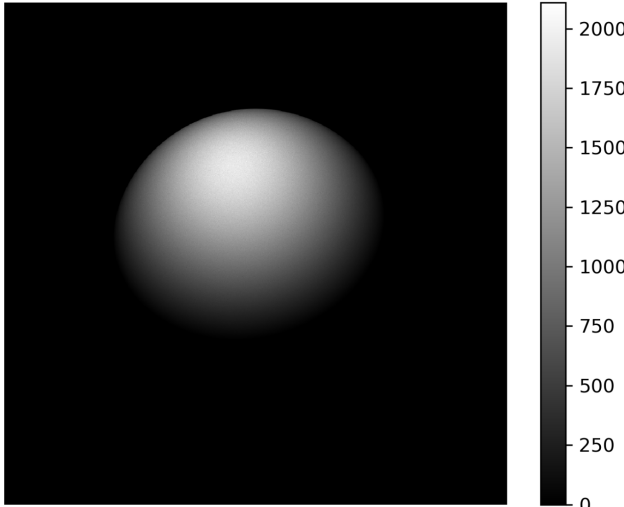
- **Compiler Optimizations with Fast Math:** By utilizing the '**--use_fast_math**' compiler flag (and **-ffast-math** on the OMP parallelization), I allowed the compiler to employ faster, less precise math operations. This optimization can lead to significant performance gains, especially in floating-point arithmetic-intensive applications like ray tracing.
- **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.

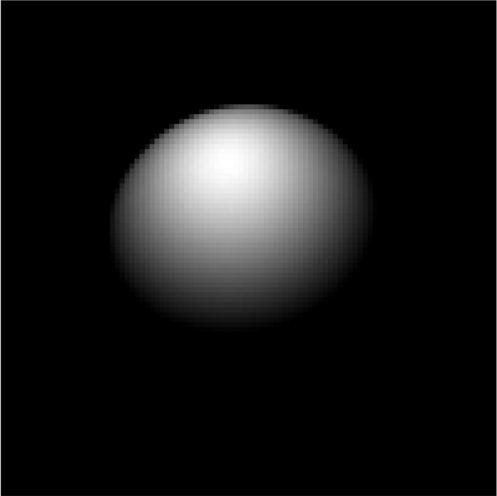
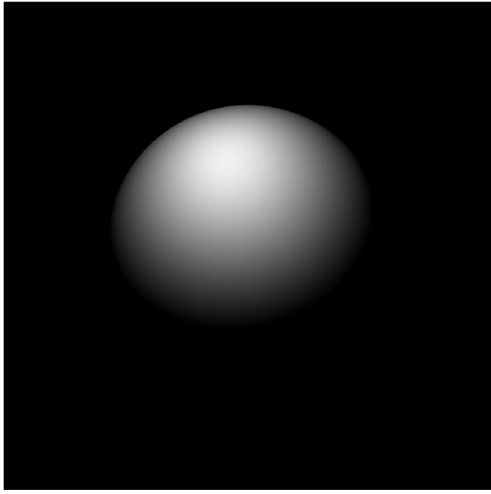
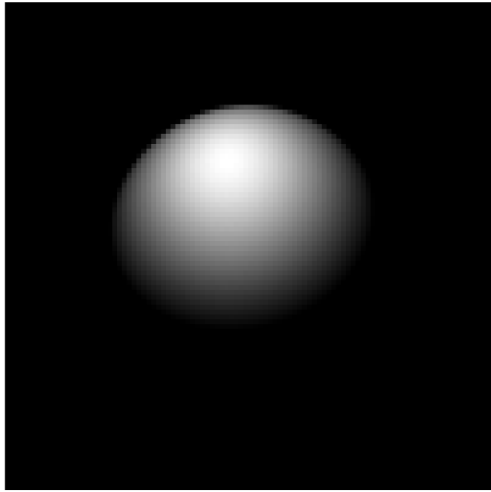
Results:

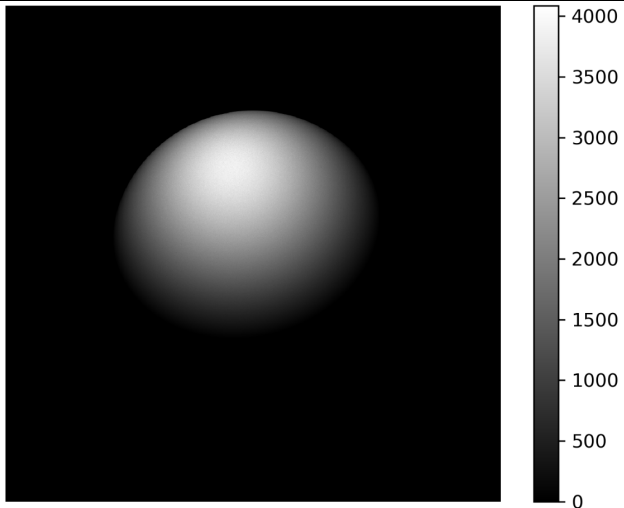
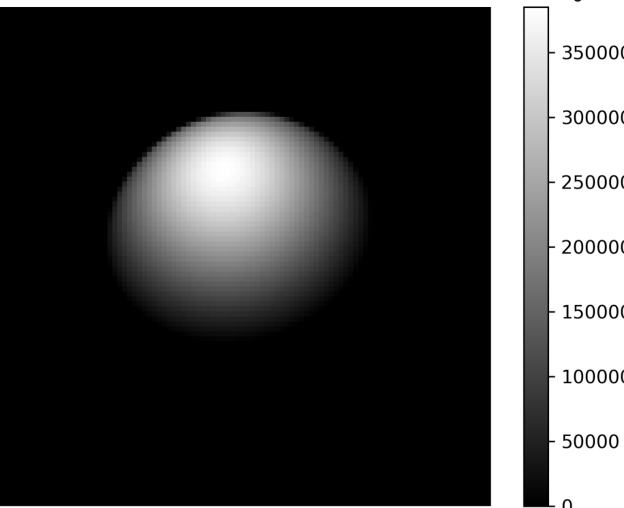
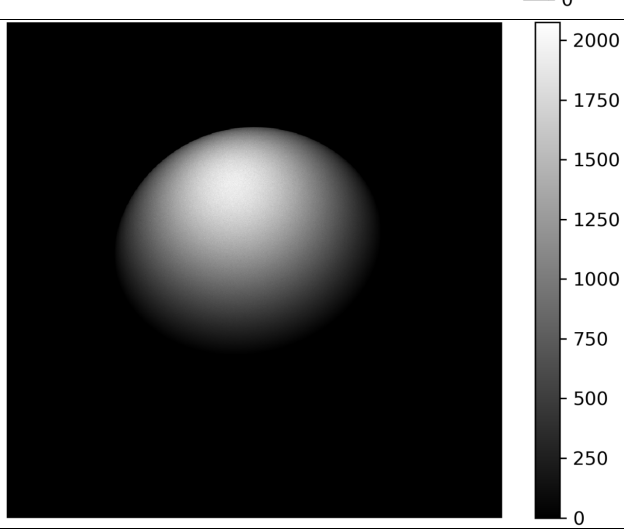
The results below are the performance metrics specified in Table 1 of the final project specification. Further below that are images from each run of the ray tracing, verifying the program results at each specified configuration.

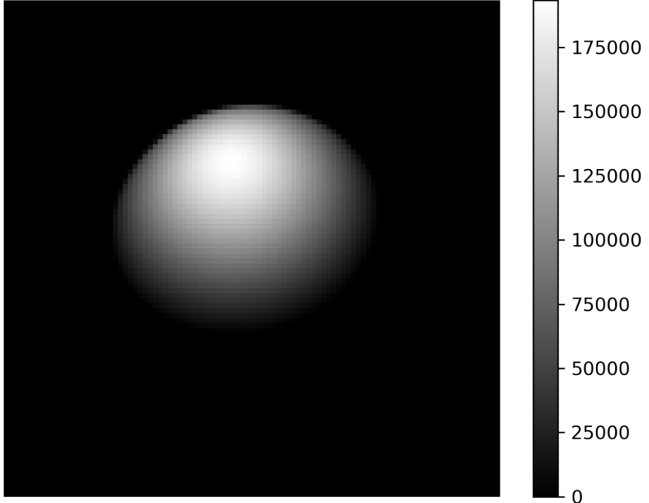
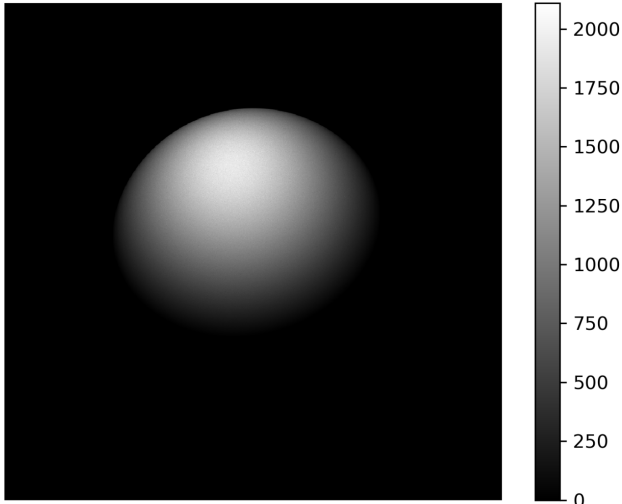
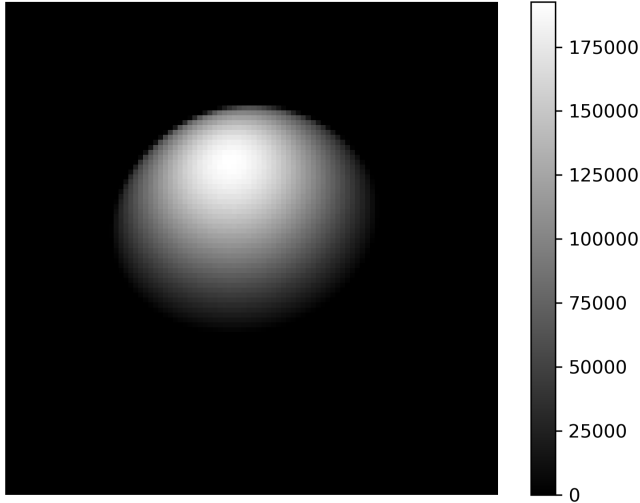
*Note: The A100 results could not be entered at this time due to the A100 being down on the Midway server. These rows will be populated when the GPU is working again. The number of samples for the RTX6000 2 GPU run could not be achieved, as I had implemented the counter in a later edit of the code and could not get access to 2 RTX6000 GPUs on Sunday, February 18th after waiting over 3+ hours. Due to similar issues getting 48 cores on a Cascade Lake node, the multicore simulation was run on 16 cores instead. On the GPU, 512 threads per block were used when I received the CUDA error: '**too many resources requested for launch**'. Otherwise, 1024 threads were used.

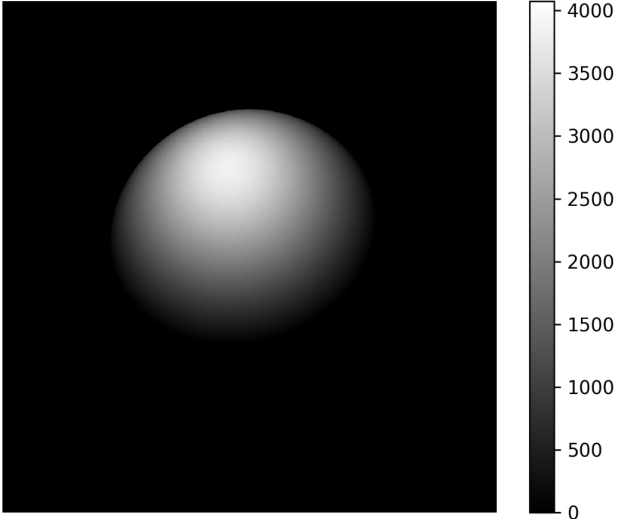
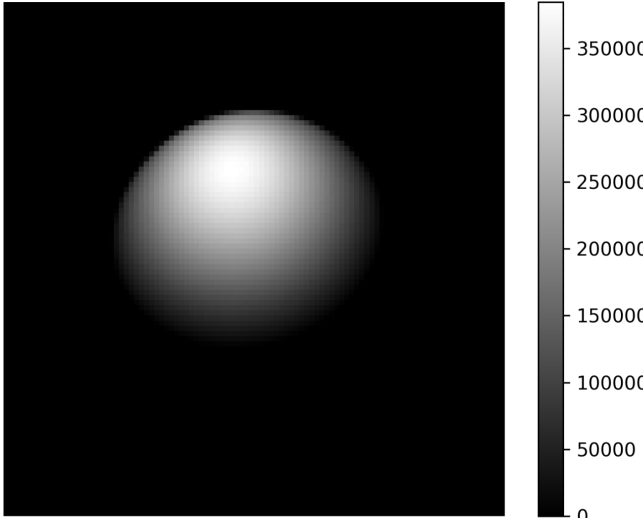
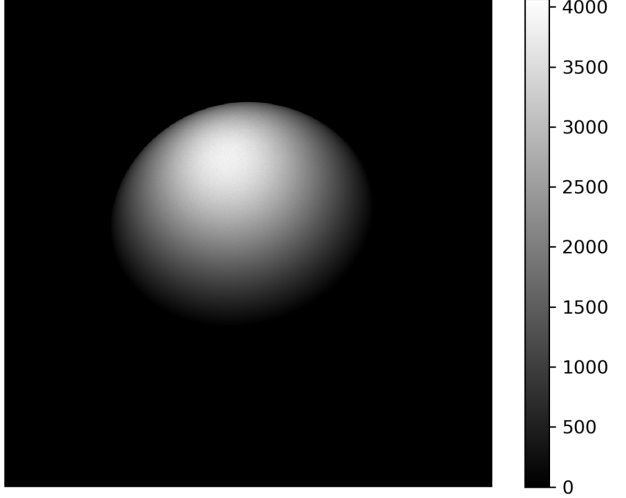
Proc	Grid	Time (SP)(s)	K Time (SP)(s)	Time (DP)(s)	K Time (DP) (s)	Blk/TPB	Cores	Samples	Optimizations
A100	1000 x 1000						N/A		
A100	100 x 100						N/A		
									<ul style="list-style-type: none"> • The use of <code>__constant__</code> memory allows usage of the CUDA constant memory space, which greatly improved speed. • Write the output data in binary mode directly to the file, rather than converting it into formatted text. •initializing random number generation states once per thread, allowing for reuse across multiple kernel executions • Utilized the <code>---use-fast-math</code> compiler flag"
V100	1000 x 1000	0.278	0.164	0.915	0.799	1000 blocks, 512 threads	N/A	14,927,783,642	
V100	100 x 100	0.274	0.164	0.908	0.798	1000 blocks, 512 threads	N/A	14,927,783,642	Same as V100
						1000 blocks, 512 threads (double precision); 1000 blocks, 1024 threads (single precision)	N/A	14,927,783,642	Same as V100
RTX6000	1000 x 1000	0.225	0.142	10.190	10.116				
						1000 blocks, 512 threads (double precision); 1000 blocks, 1024 threads (single precision)	N/A	14,927,783,642	Same as V100
RTX6000	100 x 100	0.224	0.142	10.200	10.131				
									<ul style="list-style-type: none"> • Transitioned from using multiple scalar variables to using arrays for the vectors used within the ray trace kernel. • Before the computationally intensive dot product calculations, I added a preliminary check to ensure that the rays (W_x and W_z) are within the desired range. • Utilized the <code>--fast-math</code> compiler flag
CPU Serial	1000 x 1000	364.000	N/A	433.665	N/A	N/A	1	14,928,181,932	
CPU Serial	100 x 100	362.000	N/A	372.920	N/A	N/A	1	14,928,174,562	Same as CPU Serial
CPU OMP	1000 x 1000	28.000	N/A	27.599	N/A	N/A	16	14,928,119,916	Same as CPU Serial
CPU OMP	100 x 100	28.000	N/A	27.632	N/A	N/A	16	14,928,116,394	Same as CPU Serial
>1 GPU (RTX6000)	1000 x 1000	0.417	Kernel 1: 0.077 Kernel 2: 0.063	5.114	Kernel 1: 2.014 Kernel 2: 4.926	1000 blocks, 512 threads (double precision); 1000 blocks, 1024 threads (single precision)	2 GPUs	N/A (due to GPU unavailability)	Same as V100
>1 GPU (RTX6000)	100 x 100	0.409	Kernel 1: 0.077 Kernel 2: 0.064	5.103	Kernel 1: 1.999 Kernel 2: 4.914	1000 blocks, 512 threads (double precision); 1000 blocks, 1024 threads (single precision)	2 GPUs	N/A (due to GPU unavailability)	Same as V100
>1 GPU (V100)	1000 x 1000	0.428	Kernel 1: 0.076 Kernel 2: 0.034	0.870	Kernel 1: 0.373 Kernel 2: 0.148	1000 blocks, 512 threads (double precision); 1000 blocks, 1024 threads (single precision)	2 GPUs	GPU 1: 7,463,723,242 GPU 2: 3,000,082,804	Same as V100
>1 GPU (V100)	100 x 100	0.489	Kernel 1: 0.076 Kernel 2: 0.034	0.661	Kernel 1: 0.373 Kernel 2: 0.146	1000 blocks, 512 threads (double precision); 1000 blocks, 1024 threads (single precision)	2 GPUs	GPU 1: 7,463,723,242 GPU 2: 3,000,082,804	Same as V100

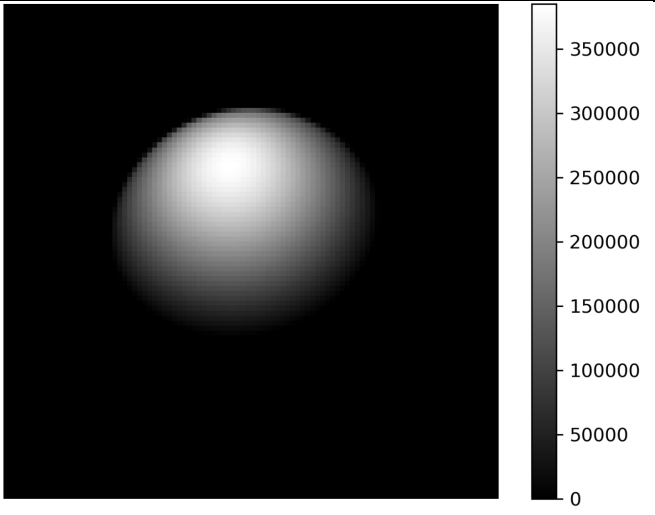
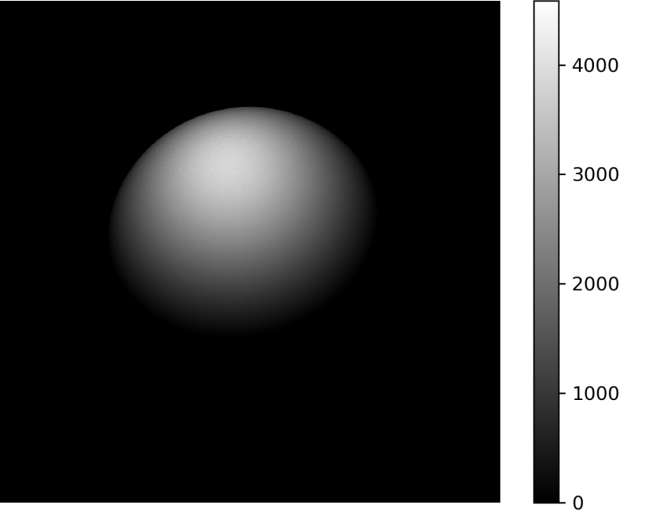
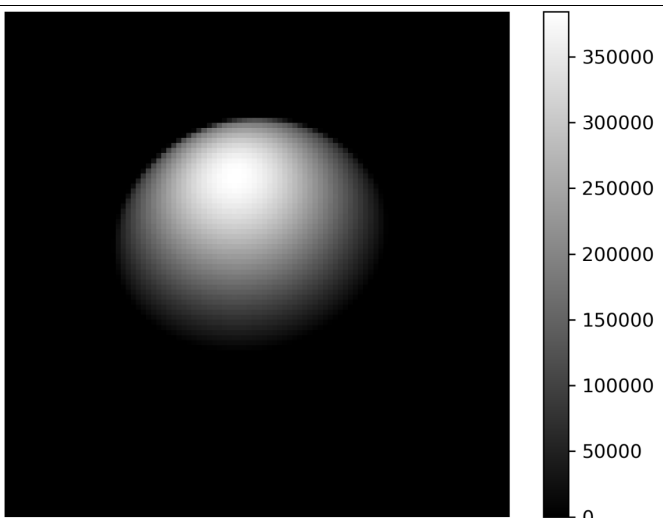
<i>Image</i>	<i>Processor</i>	<i>Grid</i>	<i>Precision</i>
	RTX6000 (2 GPUs)	1000^2	Single
	RTX6000 (2 GPUs)	100^2	Single
	RTX6000 (2 GPUs)	1000^2	Double

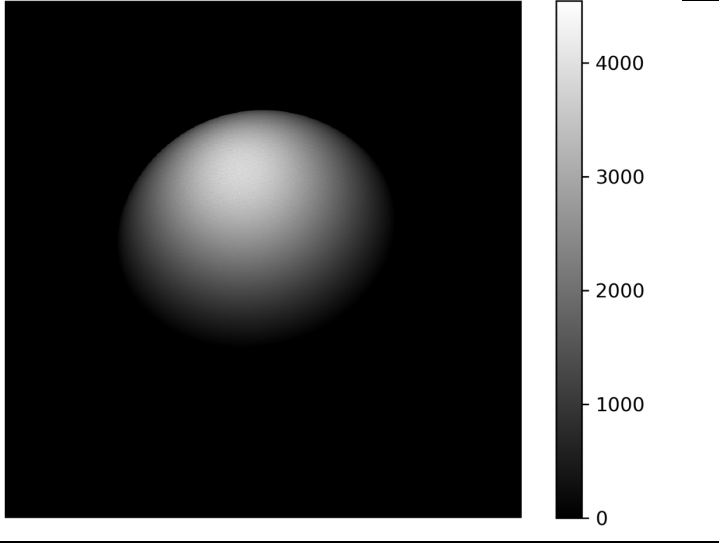
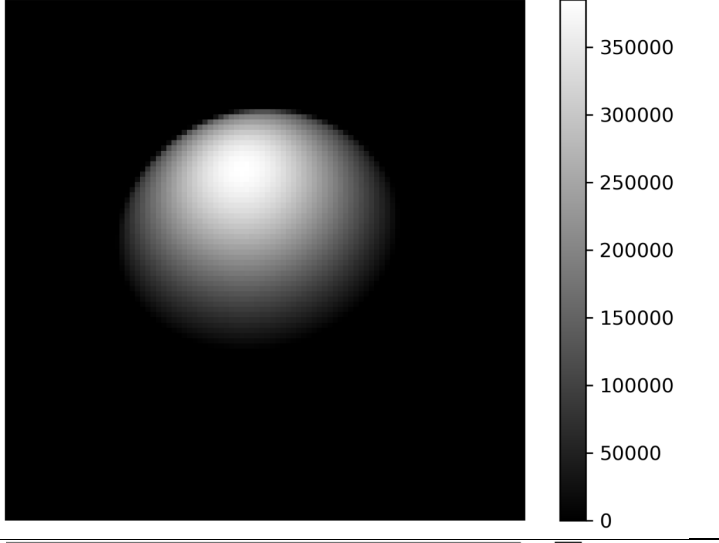
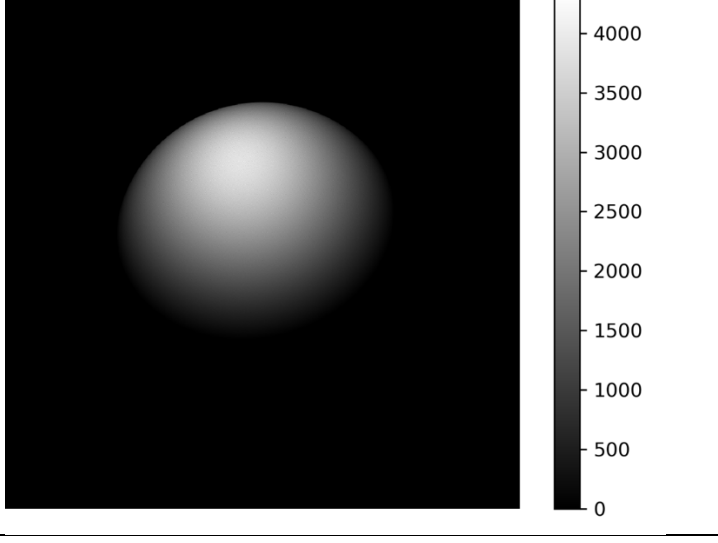
	RTX6000 (2 GPUs)	100^2	Double
	RTX6000	1000^2	Single
	RTX6000	100^2	Single

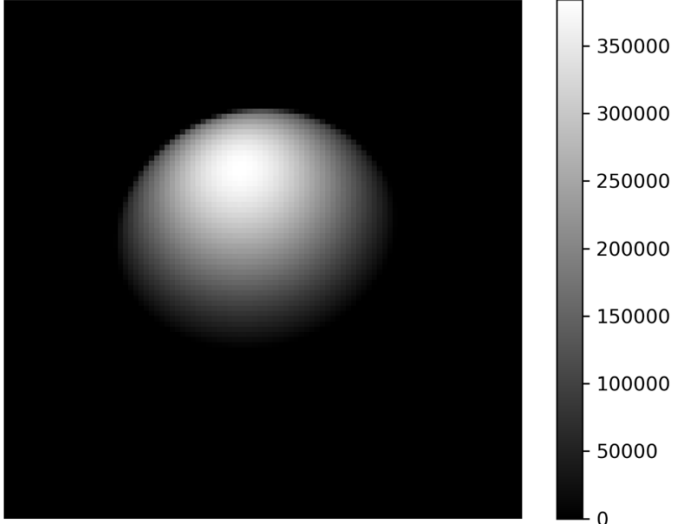
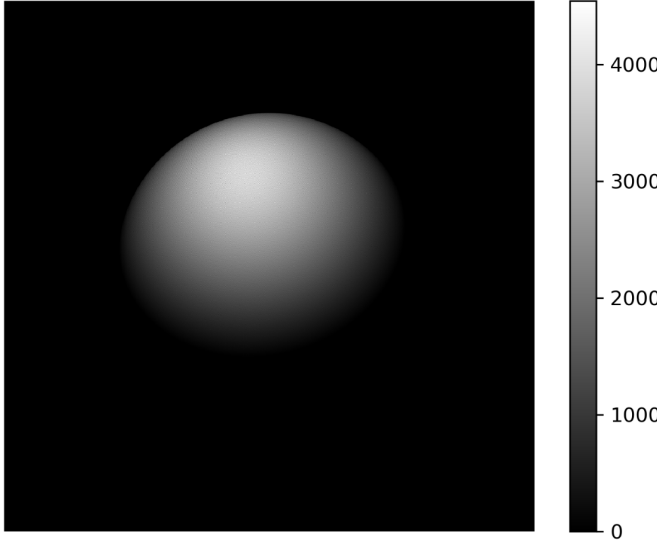
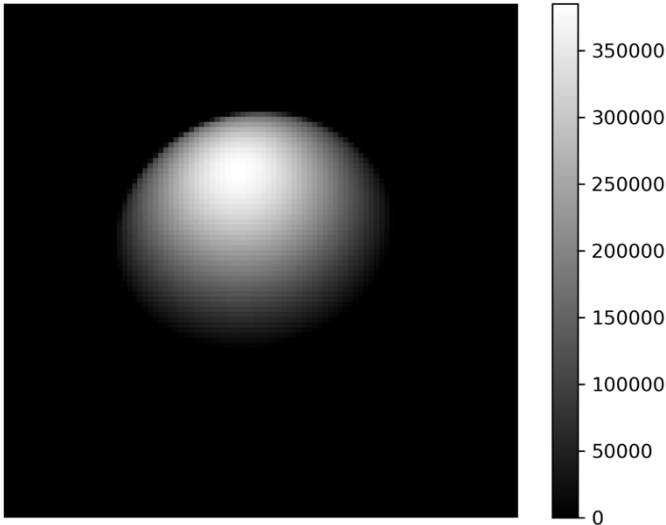
	RTX6000	1000^2	Double
	RTX6000	100^2	Double
	V100 (2 GPUs)	1000^2	Single

	V100 (2 GPUs)	100^2	Single
	V100 (2 GPUs)	1000^2	Double
	V100 (2 GPUs)	1000^2	Double

	V100	1000^2	Single
	V100	100^2	Single
	V100	1000^2	Double

	V100	100^2	Double
	CPU Serial	1000^2	Single
	CPU Serial	100^2	Single

	CPU Serial	1000^2	Double
	CPU Serial	100^2	Double
	CPU OMP	1000^2	Single

	CPU OMP	100^2	Single
	CPU OMP	1000^2	Double
	CPU OMP	100^2	Double