Project 2

February 20, 2025

Introduction

There was no room in the table to describe all of the optimizations so I condensed them into paragraphs discussing CPU, GPU, and trade-offs. At the time of this submission I am still waiting for 2 a100 runs to finish as discussed.

CPU Performance Optimizations

The CPU implementation uses OpenMP for multi-threaded parallelism, distributing the computational workload across multiple cores. The workload is divided into static chunks using `#pragma omp parallel for schedule(static, 1)`, ensuring an even distribution of work across threads to minimize load imbalance. Each thread maintains a local grid to accumulate brightness values before merging them into the global grid, reducing the need for expensive atomic operations and improving memory efficiency.

To enhance memory locality and cache efficiency, computations are structured to ensure contiguous memory access. The 2D grid that stores brightness values is accessed in a **row-major order** to exploit spatial locality and reduce cache misses. Random number generation is also optimized by initializing a separate `XorShift128Plus` random number generator per thread, avoiding contention in shared state updates and ensuring independent random sequences for each execution path.

I further optimize the mathematical computations by employing precomputed factorial values for Taylor series approximations of sine and cosine functions, reducing redundant floating-point operations. Vectorized operations using Highway (HWY) intrinsics (`LoadU`, `Mul`, `Div`) are used to process multiple floating-point values in parallel, accelerating vector arithmetic. Additionally, atomic operations are minimized by accumulating results in thread-local storage before performing a final global aggregation in a separate parallel loop.

The OpenMP strategy is further refined by using `#pragma omp atomic` for incrementing the rejection and acceptance counters, ensuring minimal contention while tracking simulation statistics. The kernel execution is timed using `omp_get_wtime()` to measure performance, providing insight into execution efficiency and identifying potential bottlenecks for further optimization.

GPU Performance Optimizations

The GPU implementation is structured to maximize occupancy, ensuring that as many Streaming Multiprocessors (SMs) as possible are fully utilized. The number of threads per block and blocks per grid is tuned using `cudaOccupancyMaxPotentialBlock` to determine the optimal launch configuration. This ensures that a high number of warps are scheduled per SM, reducing idle GPU execution cycles and maximizing computational throughput.

Random number generation is offloaded to the GPU using `curandState`, with each thread initializing its own state in the `setup_random_states` kernel. This avoids unnecessary kernel launches and allows the persistent kernel to reuse pre-initialized states, reducing overhead. Additionally, the simulation employs `persistent_kernel_loop`, where each thread iterates through multiple ray samples rather than launching a separate kernel for each batch. This eliminates kernel launch overhead and improves GPU efficiency.

Memory access patterns are optimized by preloading frequently accessed variables (such as the sphere center `C` and light source `L`) into shared memory. This reduces global memory fetch latency since shared memory access is significantly faster. Additionally, the simulation kernel employs a precomputed batch of random vectors stored in `d_vectors`, reducing the need for per-thread random generation and improving throughput.

Fast mathematical approximations further improve performance. The implementation replaces standard `sqrt` calls with `fast_sqrt`, a manually implemented Newton-Raphson approximation that reduces computational latency. The GPU version also avoids expensive branching by using fused multiply-add (FMA) operations where possible, reducing instruction latency.

The grid-based brightness accumulation leverages `atomicAdd` to safely update global memory, ensuring that multiple threads can contribute results concurrently without race conditions. To reduce contention, partial results are first computed in registers and accumulated locally before being written back to the global grid. The final aggregation across MPI ranks is done using `MPI_Reduce`, consolidating all GPU contributions efficiently.

The kernel execution time is measured using CUDA events (`cudaEventRecord()`), ensuring precise profiling of the GPU workload. The implementation also includes debugging utilities such as `CUDA_POST_KERNEL_CHECK` to verify successful kernel execution and detect runtime errors.

Performance Tradeoffs

While the implemented optimizations significantly improve performance, they also introduce trade-offs that impact resource utilization and scalability. One such trade-off in the GPU implementation is the duplication of grid memory to allow per-thread-block accumulation before a final reduction, reducing memory contention but increasing overall memory usage. This results in slightly fewer active threads per block, as some registers and shared memory resources are reserved for grid storage rather than computation. Additionally, using persistent kernels improves execution efficiency by reducing kernel launch overhead, but it can introduce workload imbalance if some threads complete their assigned work faster than others. In the CPU implementation, the decision to precompute factorial values and leverage vectorized operations accelerates mathematical computations but increases cache pressure due to additional memory accesses. Furthermore, OpenMP's static scheduling ensures load balancing but does not dynamically adjust for varying per-thread workloads, potentially leading to underutilization in some cases. The use of `atomicAdd` for brightness accumulation in the GPU grid ensures correctness but can introduce memory contention at high thread densities, particularly when multiple warps attempt to update the same memory location simultaneously.

Successful Optimizations

Optimization	**Impact**
OpenMP parallelization	Utilizes multiple CPU cores, reducing execution time.
Thread-local grid accumulation	Reduces contention on shared memory, improving efficiency.
HWY vectorized operations	Leverages SIMD for faster floating-point computations.
Taylor series approximations	Minimizes expensive transcendental function calls.
Persistent CUDA kernels	Reduces kernel launch overhead and increases utilization.
Optimized block/thread configuration	Maximizes GPU occupancy and throughput.
Shared memory usage	Reduces global memory latency for frequently accessed data.
Precomputed random vectors	Lowers the cost of random sampling per thread.
`fast_sqrt` function	Replaces expensive `sqrt()` calls with an approximation.
`atomicAdd` brightness accumulation	Ensures safe concurrent writes to the output grid.
CUDA occupancy tuning	Ensures high warp utilization across Streaming Multiprocessors.
MPI-based reduction	Efficiently aggregates results across multiple GPUs.

Table 1: Summary of Successful Performance Optimizations

Performance

[1]

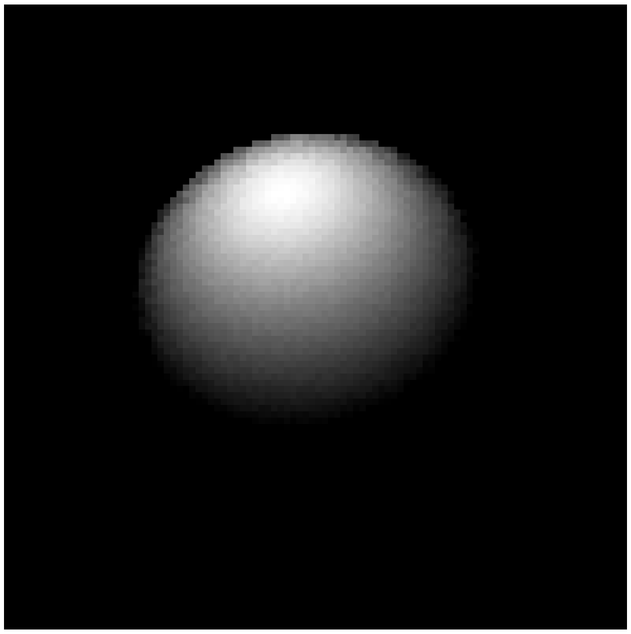
[1] *Use these rows to report on multi-GPU runs. At minimum you should run one 2 GPU configuration. Anything beyond that will depend on availability.

Flag	Description
NVCC (CUDA) Flags	
`-O3`	Enables aggressive compiler optimizations for maximum performance.
`--use_fast_math`	Enables approximate math functions for increased speed.
`-faggressive-loop-optimizations`	Optimizes loops to improve execution efficiency.
`--ptxas-options=-v`	Provides verbose output from the PTX assembler for debugging.
`--extra-device-vectorization`	Instructs the compiler to use additional vectorization optimizations.
`--resource-usage`	Outputs GPU resource usage statistics for performance analysis.
`-Xptxas=-dlcm=ca`	Forces L1 cache usage for local memory access optimization.
`--fmad=true`	Enables fused multiply-add operations to reduce floating-point instruction laten
`-Xcompiler=-fopenmp`	Passes OpenMP flags to the host compiler for multi-threading support.
`-gencode=arch=compute_80,code=sm_80`	Targets NVIDIA Ampere architecture GPUs (A100, RTX 30xx).
`-gencode=arch=compute_80,code=compute_80`	Ensures forward compatibility with future GPUs.
Intel ICX (CPU) Flags	
`-O3`	Enables highest level of compiler optimizations.
`-qopenmp`	Enables OpenMP for multi-threading.
`-ftree-vectorize`	Enables automatic loop vectorization for SIMD execution.
`-ffast-math`	Allows floating-point optimizations that may sacrifice strict IEEE compliance.
`-funroll-loops`	Expands loops to eliminate branching overhead.
`-march=native`	Optimizes for the specific CPU architecture of the host system.
`-fverbose-asm`	Generates assembly output with detailed comments.
Linker and MPI Flags	
`-lomp`	Links against the OpenMP runtime library.
`MPI_LIBRARIES`	Ensures proper linking with MPI for multi-node parallelism.

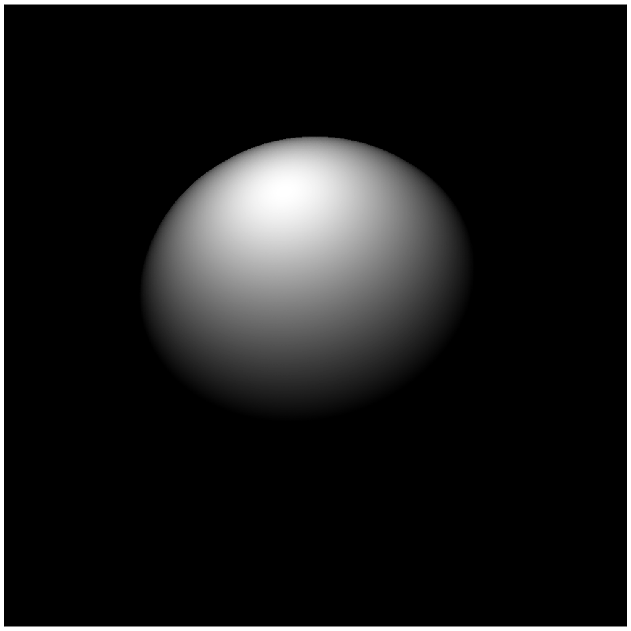
Table 2: Compiler Flags and Their Performance Impact

Proc	Grid	Time(SP)	K Time(SP)	Time(DP)	K Time(DP)	TPB/Blk	Cores	Samples
A100	1000^2	107.55ms	121.548ms	514.609ms	499.988 ms	640 / 96	-	7463024805
A100	100^2	185.867ms	171.402ms	500.563	492.009ms	640 / 96	-	7462986737
V100	1000^2	101.403ms	88.79ms	437.698ms	426.308ms	512 / 120	-	7463746055
V100	100^2	96.95ms	88.76ms	433.7ms	438.6ms	768 / 72	-	7463746055
RTX6000	1000^2	359.03ms	348.78ms	8949ms	8935.15ms	768 / 72	-	7463746055
RTX6000	100^2	364.26ms	353.68ms	8942.2ms	8935.26ms	128 / 480	-	7463746055
CPU Serial	1000^2	91.106s	91.0981s	77.0s	75s	-	1	7463055357
CPU Serial	100^2	91.106s	130.573s	78.3s	75.01s	-	1	7463311285
CPU OMP	1000^2	10.21s	9.99s	6.10s	6.09s	-	16	7462817538
CPU OMP	100^2	33.35s	33.35s	6.45s	6.39s	-	16	7463014922
> 1 GPU*	1000^2	73.27ms	44.83ms	254.588ms	230.418 ms	768 / 80	2	7463291276
> 1 GPU*	100^2	45.61ms	54.50ms	236.664ms	224.402ms	768 / 80	2	7463291276

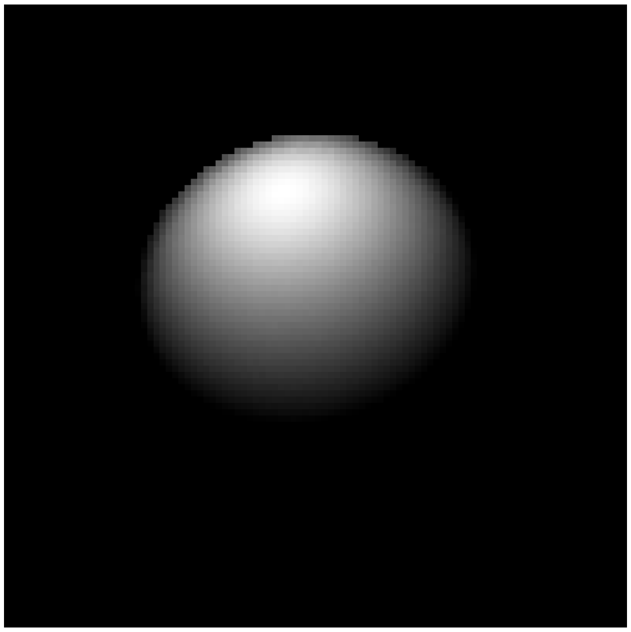
Table 3: Fill out Table 3 with your best performance for the hardware and problem size specified. The Total Time (Time) and Kernel Time (K Time) columns must include both single and double precision performance (SP/DP). Assume one billion rays, xorwow RNG in curand, with problem parameters set as in Milestones 1 and 2. The CPU must be a Midway 3 Cascade Lake node. For the multi-GPU runs, the "cores" column should be used to denote the number of MPI ranks. Samples refers to the total number of random numbers drawn in the simulation. It is included as a sanity check.



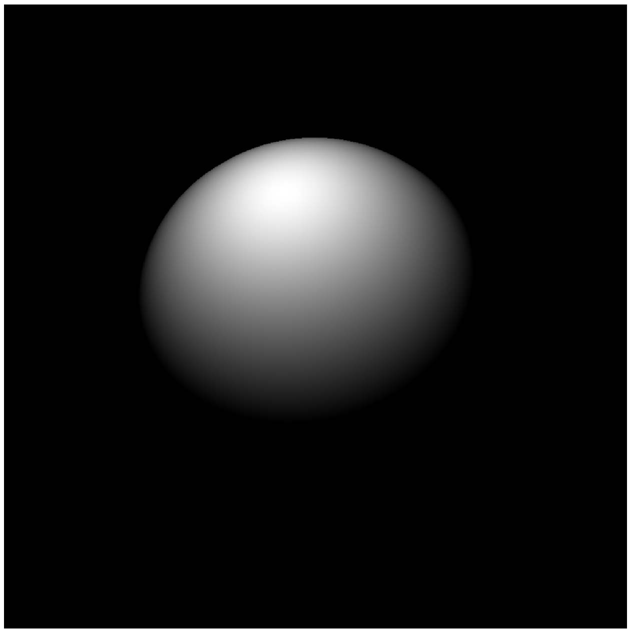
(a) CPU Double Serial 100



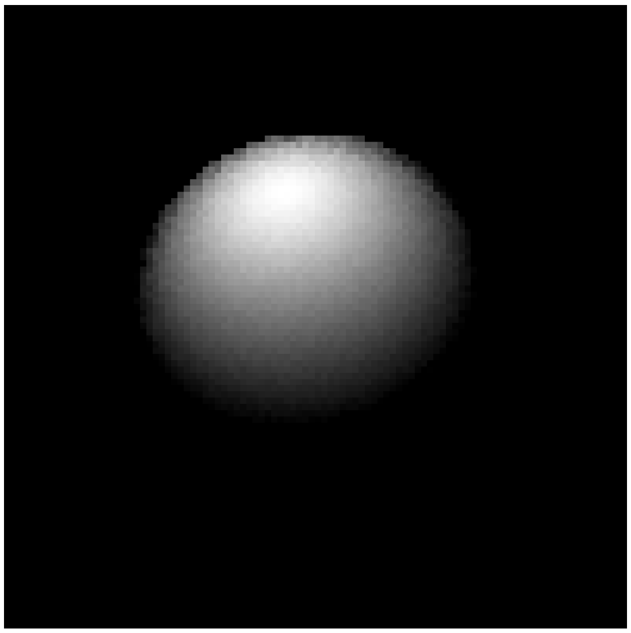
(b) CPU Double Serial 1000



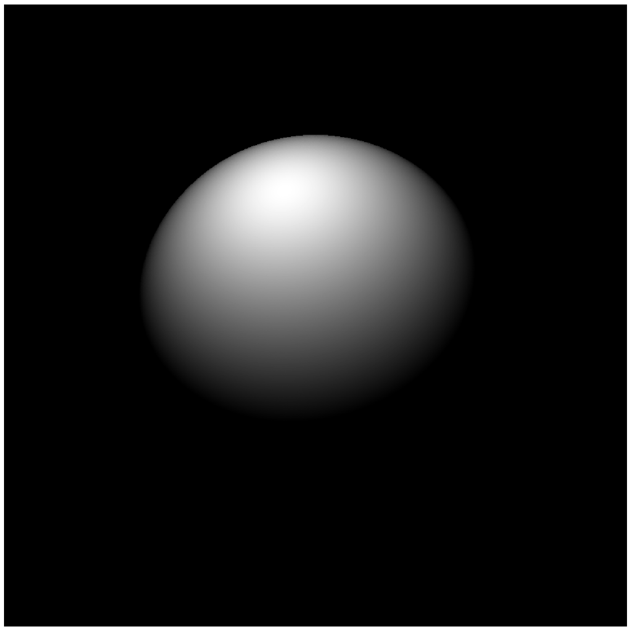
(c) CPU Double 16 threads 100



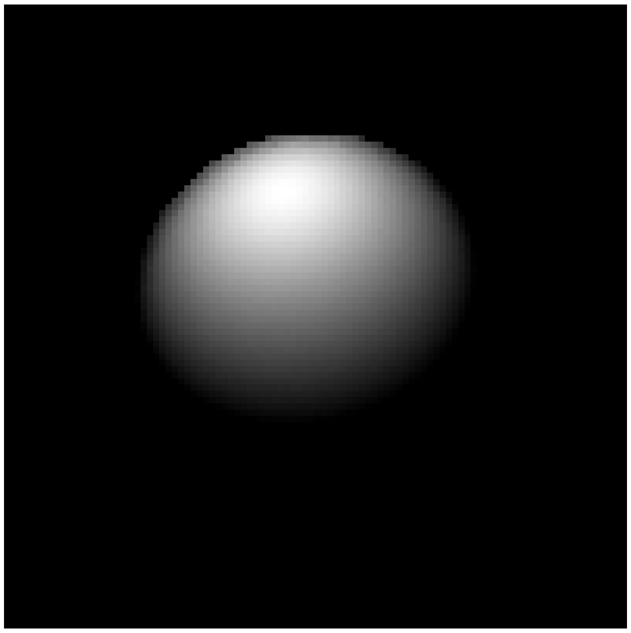
(d) CPU Double 16 threads 1000



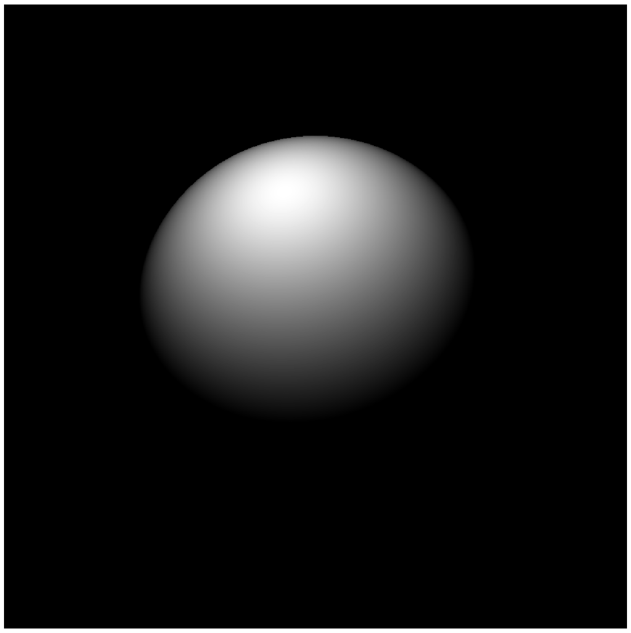
(a) CPU Float Serial 100



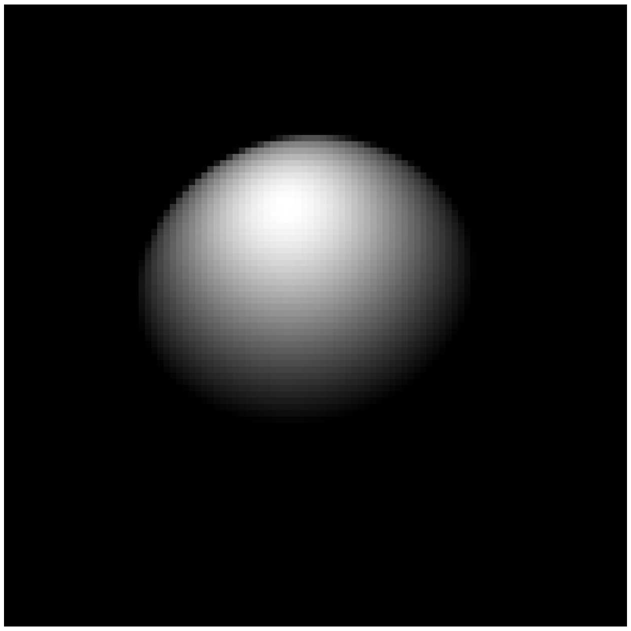
(b) CPU Float Serial 1000

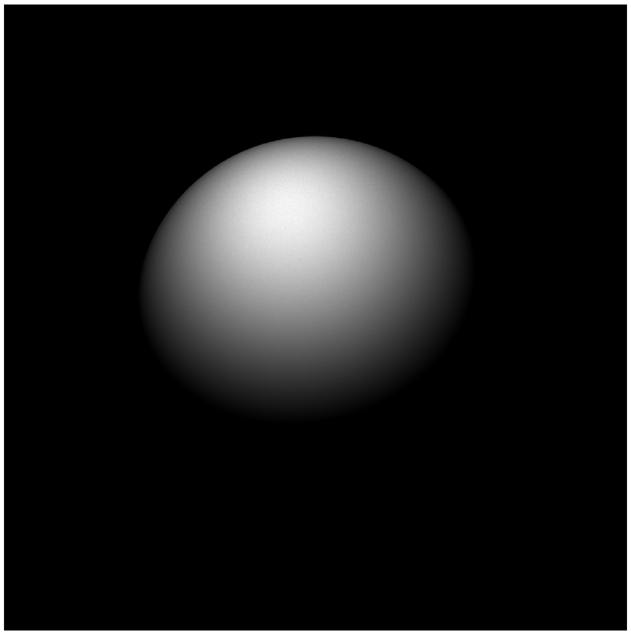


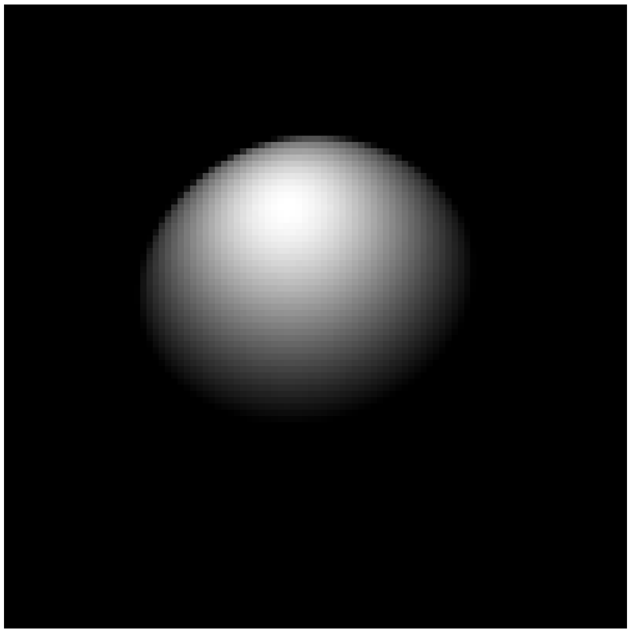
(c) CPU Float 16 threads 100

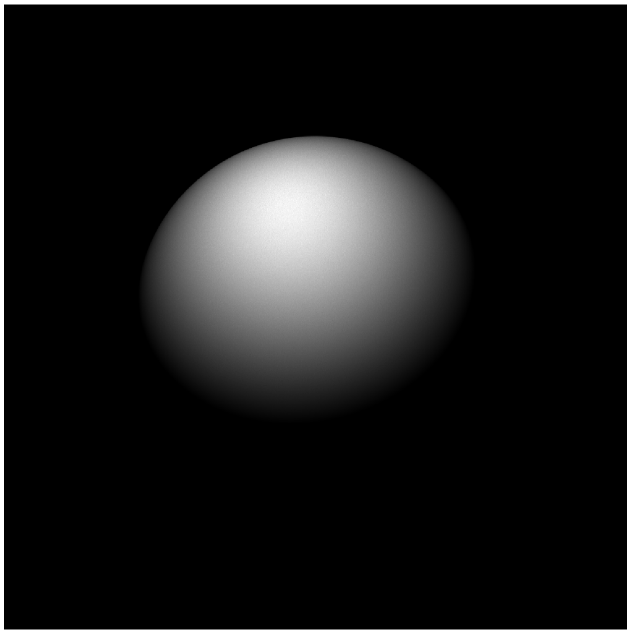


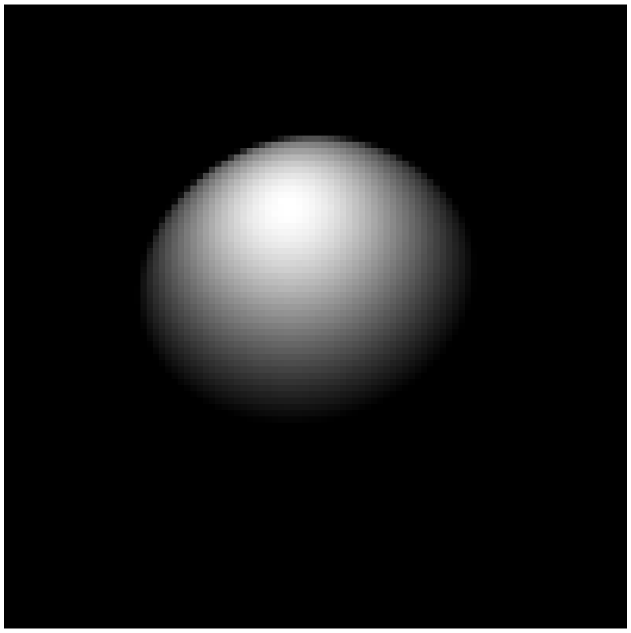
(d) CPU Float 16 threads 1000

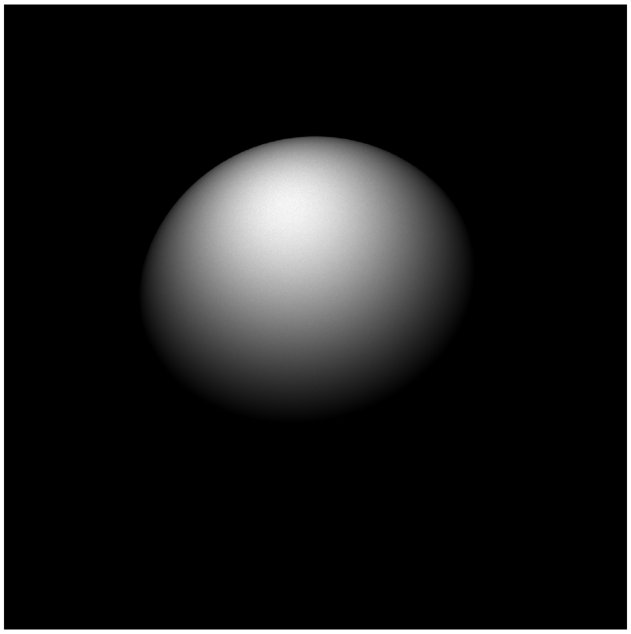

(a) RTX Float 100

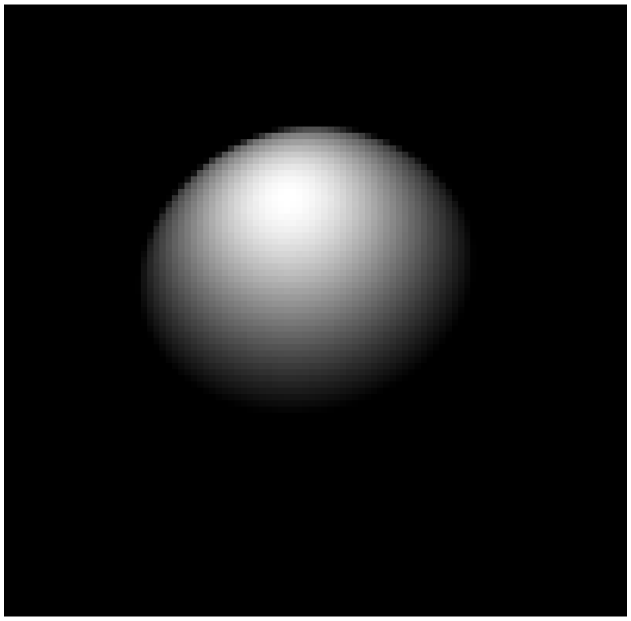

(b) RTX Float 1000

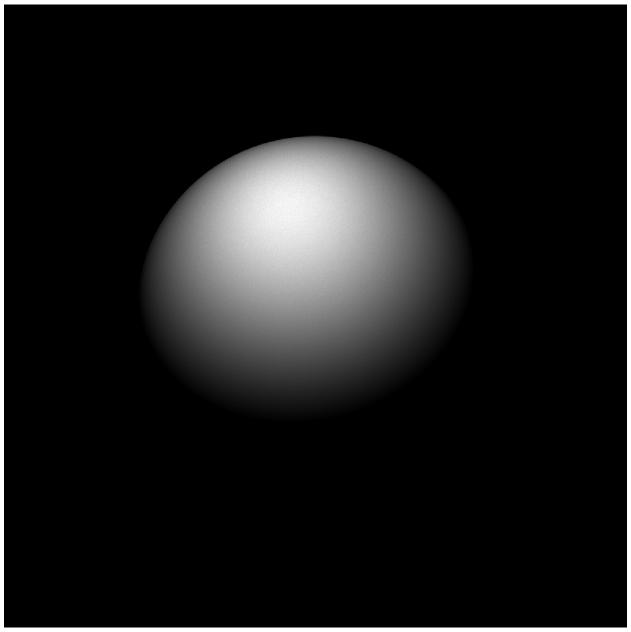

(c) RTX double 100

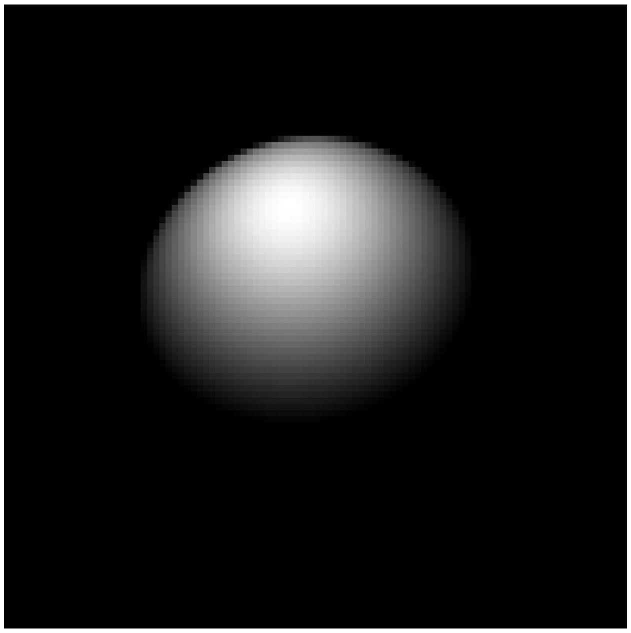

(d) RTX double 1000

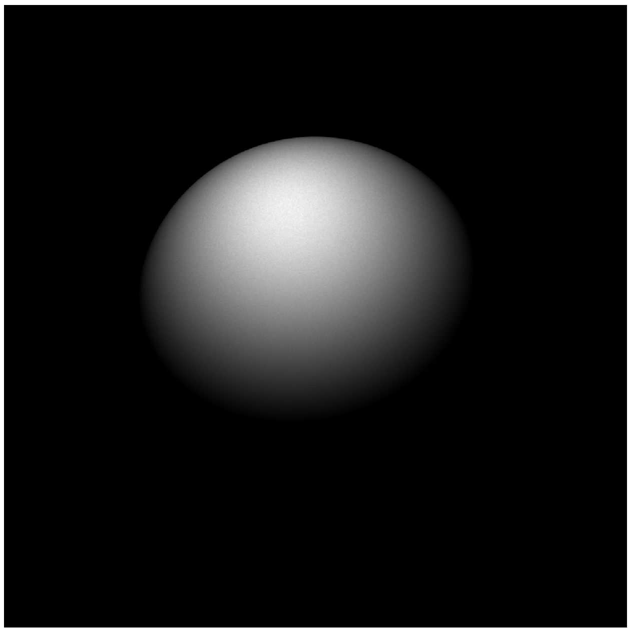

(a) V100 float 100

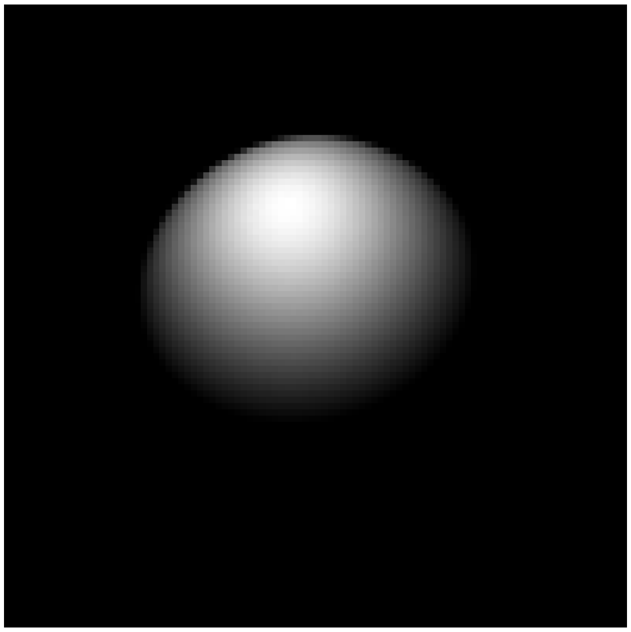

(b) v100 float 1000

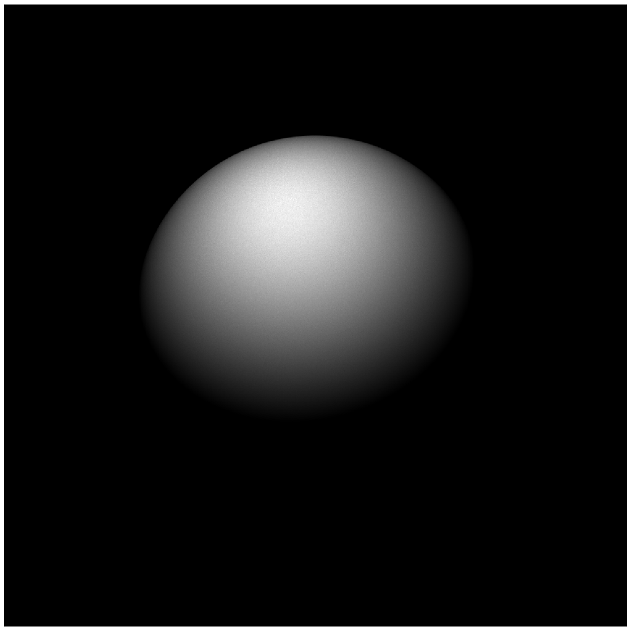

(c) v100 double 100

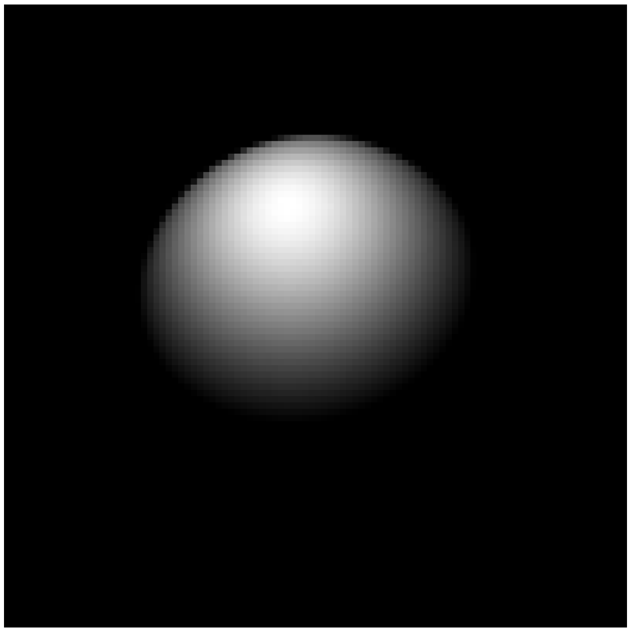

(d) v100 double 1000

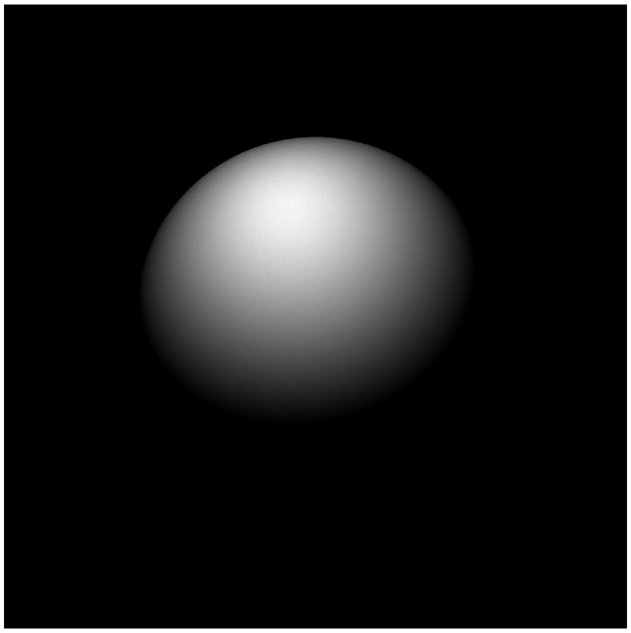

(a) v100 MPI 2 float 100

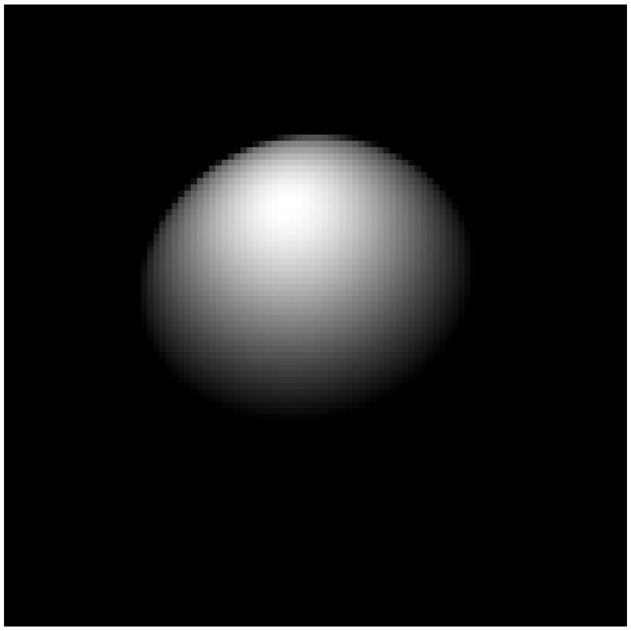

(b) v100 MPI 2 float 1000

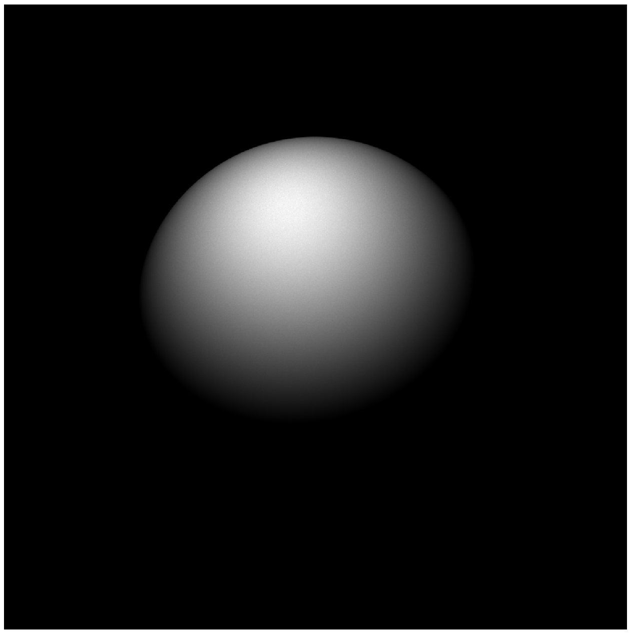

(c) v100 MPI 2 double 100


(d) v100 MPI 2 double 1000


(a) v100 MPI 2 float 100


(b) v100 MPI 2 float 1000


(c) v100 MPI 2 double 100


(d) v100 MPI 2 double 1000