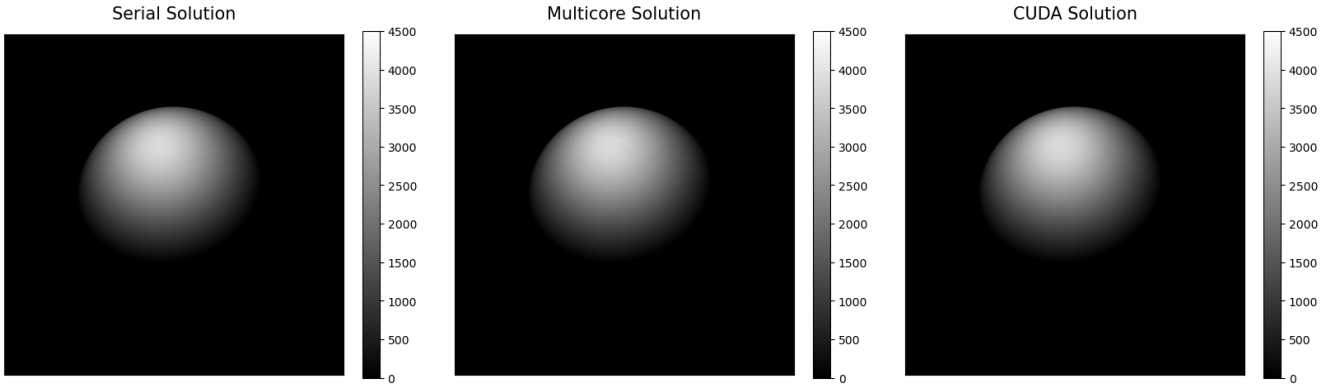
Project 2, Milestone 2 — Ray Tracing with OMP and CUDA

Algorithmic Approach

To map this problem onto the GPUs, the goal is to maximize the parallelism by executing the ray generation loop on the devices and keeping the allocation and management of the shared data structures on the host. Each thread independently generates a ray and and updates the brightness of the matrix, using the atomic operation to avoid race conditions. The host is responsible for the memory allocation, kernel launch, and result aggregation. We also adapted the PRNG for the GPU execution using the cuRAND library.

To map this problem onto the NVIDIA GPUs, the goal is to maximize the parallelism by executing the ray generation on the devices and keeping the allocation and management of the shared data structures on the host. Each thread independently generates a ray and and updates the brightness of the matrix, using the atomic operation to avoid race conditions and data corruption. The host is responsible for the memory allocation, kernel launch, and result aggregation. We also adapted the PRNG for the GPU execution using the cuRAND library provided by CUDA.

Validation



Results

Run Type	PRNG	Processor	Cores/Block Size	Time (s)	Time Spent in PRNG
Serial	xoshiro256++	Apple M1	N/A	266.40	208.59
Serial	xoshiro256++	Caslake	N/A	375.69	317.11
Multicore	xoshiro256++	Apple M1	16 threads	66.99	48.60
Multicore	xoshiro256++	Caslake	16 threads	188.19	160.55
GPU	cuRAND	NVIDIA	3,906,250 blocks and 256 threads	10.70	0.068

Table 1: Benchmark Performance Results. Time refers to the total execution time, including all setup, i/o, and data copies to and from device, etc. Time spent in PRNG should be measured within the full code, not as a standalone experiment. Extra rows (for GPU and Multicore runs) are optional, in case you want to report results on more than one processor.