**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 (**W\_x** and **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.

**A white sheet with black text

Description automatically generated**

|  |  |  |  |
| --- | --- | --- | --- |
| *Image* | *Processor* | *Grid* | *Precision* |
| A close-up of a planet  Description automatically generated | RTX6000 (2 GPUs) | 10002 | Single |
| A close-up of a sphere  Description automatically generated | RTX6000 (2 GPUs) | 1002 | Single |
| A close-up of a planet  Description automatically generated | RTX6000 (2 GPUs) | 10002 | Double |
| A close-up of a planet  Description automatically generated | RTX6000 (2 GPUs) | 1002 | Double |
| A close-up of a planet  Description automatically generated | RTX6000 | 10002 | Single |
| A close-up of a sphere  Description automatically generated | RTX6000 | 1002 | Single |
| A close-up of a planet  Description automatically generated | RTX6000 | 10002 | Double |
| A close-up of a sphere  Description automatically generated | RTX6000 | 1002 | Double |
| A close-up of a planet  Description automatically generated | V100 (2 GPUs) | 10002 | Single |
| A close-up of a sphere  Description automatically generated | V100 (2 GPUs) | 1002 | Single |
| A close-up of a planet  Description automatically generated | V100 (2 GPUs) | 10002 | Double |
| A close-up of a planet  Description automatically generated | V100 (2 GPUs) | 10002 | Double |
| A close-up of a planet  Description automatically generated | V100 | 10002 | Single |
| A close-up of a white sphere  Description automatically generated | V100 | 1002 | Single |
| A close-up of a planet  Description automatically generated | V100 | 10002 | Double |
| A close-up of a sphere  Description automatically generated | V100 | 1002 | Double |
| A close-up of a black and white image  Description automatically generated | CPU Serial | 10002 | Single |
| A close-up of a white sphere  Description automatically generated | CPU Serial | 1002 | Single |
| A close-up of a white sphere  Description automatically generated | CPU Serial | 10002 | Double |
| A close-up of a planet  Description automatically generated | CPU Serial | 1002 | Double |
| A close-up of a white sphere  Description automatically generated | CPU OMP | 10002 | Single |
| A close-up of a planet  Description automatically generated | CPU OMP | 1002 | Single |
| A close-up of a white sphere  Description automatically generated | CPU OMP | 10002 | Double |
| A close-up of a planet  Description automatically generated | CPU OMP | 1002 | Double |