High Performance Computing

MPCS 51087

Problem Set 4

GPU Ray Tracing with CUDA

Spencer Zepelin

August 20, 2019

Compiling and Running the Code

The serial program was written in C with the c99 standard. The CUDA program was written and tested with CUDA 6.5, the default version of CUDA on Midway. Source code and makefiles for each can be found in the directories “src_serial” and “src_cuda” respectively. Running the command “make” in either directory will make the executable for that directory by running one of the following commands:

```
gcc -std=c99 -Wall -pedantic -Werror raytrace.c -o raytrace -lm
nvcc -arch=sm_37 raytrace_cuda.cu -o raytrace_cuda -lm
```

Running the command “make run” will make the desired executable and run it with the following commands:

```
./raytrace 1000 100000000
./raytrace_cuda 1000 100000000
```

For both executables, the first argument after the executable is the grid dimension, 1,000 in both test cases, and the second is the number of rays, 100 million in the test cases.

Specifics about the given run—sphere location and radius, light source location, window size and location, and threads per block (for CUDA)—have been hardcoded, but they could easily be parameterized for more customizability.

Runtime data will be printed to screen after any successful execution. Binary data of the ray-traced imaged will be saved in the file “data.bin”.

Code Design

The serial code follows the algorithm laid out in the prompt very closely. To improve readability, the code makes ample use of helper functions for vector operations, random sampling, and conversion of window position back to grid index.

The CUDA code follows a very similar pattern except the bulk of the main function from the serial version was refactored into the __global__ function which initializes the kernel, “ray_thread”. Other than that function, all other helper functions use the __device__ prefix and are only run on the accelerator. The main function on the CPU primarily handles the boilerplate setup of variable allocation and sharing for the GPU, writing the calculated data to file, and breaking down the allocated memory. Based on experimentation, the kernel is initialized with 128 threads per block. The number of blocks initialized is then dependent on argument for the number of total rays such that each ray is given its own thread. In order to avoid issues with the PRNG, the function is fast-forwarded 200 spaces times the calling thread’s ID. Brightness addition is performed atomically to account for race conditions.

In both cases, timing was performed on the entire program, including writing data to file.

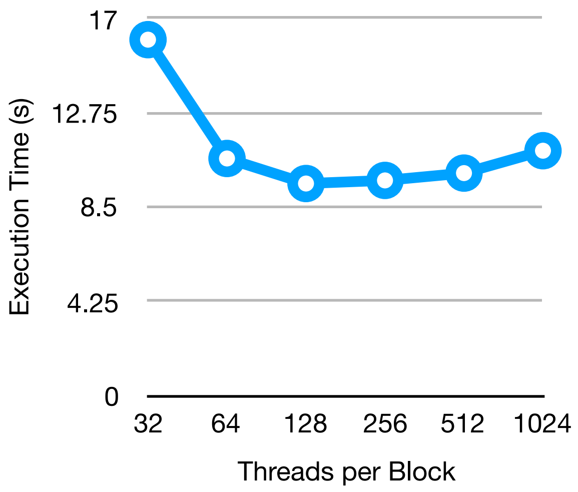
Optimal Thread Settings

The CUDA code was run with one billion rays, varying the threads per block to determine the ideal configuration. The number of blocks is a function of the number of threads per block to ensure each sample was run by its own thread. Based on the tests performed, data for which is available in the table below, threads per block was set in the code at 128.

Optimal Threads per Block

Threads per Block	Execution Time (s)
32	16.05
64	10.69
128	9.56
256	9.7
512	10.03
1024	11.04

Execution Time vs Threads per Block



Serial CPU vs CUDA GPU Runtime

The table below shows the measured values for runtime on each version of the code as a function of the number of rays. Serial time—as one would expect—scaled remarkably linearly with problem size with the runtime for one billion rays nearly exactly one thousand times that of the runtime for one million rays. If nothing else, this result highlights the need for performance improvements and the value of weak scaling in supporting advancements in the computer-generated imaging.

The CUDA version staggeringly outperformed the serial version, increasing to a speedup of over two orders of magnitude as the problem size grew. So too, given that my timing was performed over the entire program, we see a better than linear improvement as small problem sizes increase as the linear portion of the code becomes increasingly amortized.

Runtime Comparison

Number of Rays	Serial Execution Time (s)	CUDA Execution Time (s)	CUDA Speedup (serial/CUDA)
1,000,000	1.05	0.20	5.3
10,000,000	10.63	0.30	35.4
50,000,000	53.05	0.71	74.7
100,000,000	105.97	1.24	85.5
500,000,000	529.17	4.91	107.8
1,000,000,000	1058.74	9.56	110.7

Image Generation

The image on the following page was generated using a 1000 x 1000 grid and sampling one billion relevant rays. The plotting script used a grayscale, representing zero as white and the highest numeric values as black. The plotting script and generated plots can be found in the Jupyter Notebook file “ps4_plotting.ipynb” in the “plotting” directory. For convenient examination, the code from that file has also been made available in “plotting.py” though it will not execute properly outside the Jupyter Notebook environment. The data generated by the program the script references is in the same directory.

As I thought it might be of general interest, an earlier version of the notebook called “tests.ipynb” has been included as well. This notebook shows images generated with the light source at different locations, various resolutions, experimentation with non-accumulative sampling, and my struggles with image striping resulting from insufficient random number generation that I worked through with Dr. Tramm. Note that the cells should not be executed since data files for those images have not been provided.

Sample Image