Programming Assignment 4 – CUDA and Ohio Supercomputer

# Part 1: Matrix Multiplication

Part 1 of this experiment compared two implementations of an algorithm that multiplies a large 1024x1024 matrix of floats with its transpose: one serial, and one parallelized with CUDA. A sample of the console output of this program is at the end of this section. In order to compare the performance of these two implementations, timing results were gathered over 5 trials with the clock\_gettime() utility in the <time.h> library. This timing information was then used to calculate the GFlops per second benchmark for each implementation. Since each iteration of the does a single calculation composed of one floating-point multiplication and one floating point-sum, the number of total floating-point operations required for the matrix multiplication is:

The table below shows the timing results and the GFlops per second benchmark for all of the trials.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Time to execute using clock\_gettime()**  **(seconds)** | | **Benchmark**  **(GFlops per second)** | |
| Serial | CUDA | Serial | CUDA |
| Trial 1 | 8.5806 | 0.3492 | 0.2331 | 5.7274 |
| Trial 2 | 8.5715 | 0.2839 | 0.2333 | 7.0447 |
| Trial 3 | 8.6382 | 0.2971 | 0.2315 | 6.7317 |
| Trial 4 | 8.5818 | 0.3430 | 0.2331 | 5.8309 |
| Trial 5 | 8.6010 | 0.2944 | 0.2325 | 6.7935 |
| **Average** | **8.5946** | **0.3135** | **0.2327** | **6.4256** |

The CUDA compute structure used in this experiment had 1 grid of 1024 blocks each having 1024 threads. When the code was first being tested, the compute structure was 1 grid and 1 block with 1024 threads, but this was modified once preliminary testing showed that increasing the number of blocks decreases the time. This makes sense because having more blocks means more of the GPU’s hardware is being utilized at one time to do these calculations in parallel. The specific values of 1024 threads per block was chosen because the matrix dimensions are 1024x1024. Each block of threads could do an entire row’s worth of calculations, meaning that the innermost (k) loop of the serial implementation could be taken out of the kernel function and only two levels of looping were required. Furthermore, having 1024 blocks means that even though the code sets up the kernel to use cyclic distribution of the inner (j) loop, there are enough blocks that each block only executes one iteration of this loop. However, while having 1024 threads per block is intentional and makes the code simpler by eliminating an entire while loop, having 1024 blocks is somewhat arbitrary. In testing, having 2 blocks was almost twice as fast as having only one block, but having 1024 blocks wasn’t significantly faster than having 10. The main reason that 1024 was finally chosen as the number of blocks in the compute structure was because the compiler didn’t stop me from choosing such a large grid, and 1024 is a round number that matches the parameters of the problem, even if it is not the ideal grid dimension.

asdjfhajksdhfalskjd [console output]

# Part 2: Sobel Edge Detection

Part 2 of this experiment involved two implementations of a Sobel edge detection algorithm: one serial, and one parallelized with CUDA. Because both algorithms executed in nearly the same time with the small provided image coins.bmp, a larger (1200x1920) bitmap named larger.bmp was used to get a better idea of the how the execution times of the two algorithms compare. As in Part 1, the clock\_gettime() utility was how time was tracked and five trials were done with each algorithm.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  | **Time to execute using clock\_gettime()**  **(seconds)** | | **Convergence Threshold** | |
|  | Serial | CUDA | Serial | CUDA |
| coins.bmp | Trial 0 | 0.2369 | 0.2788 | 51 | 51 |
| larger.bmp | Trial 1 | 3.1291 | 0.4997 | 27 | 27 |
| Trial 2 | 3.1194 | 0.4984 | 27 | 27 |
| Trial 3 | 3.1281 | 0.4996 | 27 | 27 |
| Trial 4 | 3.1145 | 0.5007 | 27 | 27 |
| Trial 5 | 3.1554 | 0.4975 | 27 | 27 |
| **Average** | **3.1293** | **0.4992** | **27** | **27** |

The CUDA compute structure used for this algorithm

Timing, threshold convergence

CUDA organization

Performance improvements with GPU? (support with numbers)

.

**Table 1:** Run-time and thread count comparisons for the four programs at AFFECT\_RATE=0.05 and EPSILON=0.05.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Req. # threads** | **Disposable pthreads** | | **Persistent pthreads** | | **Disposable OpenMP** | | | | **Persistent OpenMP** | | |
| wall-clock | clock() | wall-clock | clock() | Actual # threads | wall-clock | clock() | Actual # threads | | wall-clock | clock() |
| 2 | 3m43s | 300.2s | 2m38s | 261.0s | 2 | 2m59s | 335.4s | 2 | | 3m01s | 303.9s |
| 8 | 2m54s | 353.1s | 1m53s | 289.6s | 8 | 3m02s | 283.2s | 8 | | 2m29s | 283.0s |
| 16 | 3m45s | 471.0s | 3m16s | 302.9s | 16 | 4m01s | 274.5s | 16 | | 2m23s | 312.6s |
| 32 | 6m38s | 669.5s | 4m09s | 323.0s | 32 | 4m02s | 308.2s | 32 | | 2m46 | 357.2s |
| **Avg.** | 4m15s | 448.0s | 2m59s | 294.1s | - | 3m31s | 300.3s | - | | 2m40s | 314.2s |