Russell Folk
CS 491/CS 521 Parallel Programming
Homework 5 Report

CUDA Matrix-Multiplication

Disclaimer

For these tests, I chose to run on the computers named borg. This machine was chosen from the ones available ones because I like *Star Trek*. To run the program, please refer to the readme, `README.md`.

Kernel Design

I designed the kernel similar to how it is implemented in the slides. After researching about CUDA, I decided that by and large the kernels can be implemented nearly the same way, only some work needs to be done to share memory. Because there was no specification on what data-type the matrices had to be, I went with integer matrices. The reason is that it's merely faster, especially on the CPU. The changes to datatypes is actually trivial and as such I don't see this as a problem.

For the non-shared-memory model the kernel is extremely simple, each thread will select a row from matrix X and a column from matrix Y and calculate the associated element in Zgpu. I used A, B, and C in the program to avoid any namespace issues. The general flow of the program is to copy data from the CPU to the GPU with error checking, then compute, then copy back the data.

For the shared-memory model, if you look closely you'll see that I basically copied the code and added some changes. I found out that you must have a few `__device__` functions if you are going to actually share the memory. The other change is making the grids smaller so that each matrix is split into sub-grids and then the program is able to compute sub-matrices and eventually combine the results into the correct answer. This kernel was heavily influenced by the slides presented in class.

Experimental Results

For these experiments I ran 3 sets of regular experiments, the first with N = 512, the second with N = 1024, and the third with N = 2048.

I also ran a special experiment in which the CPU multiplication is commented out with N = 4096. Results from the GPU only experiments were not included because shared memory involved CPU timeouts. I was unable to diagnose if this was a machine error or configuration error due to lack of time. The reason behind trying to do this configuration is that I believed it would show that non-shared memory would fail to perform better at this point. The percent of improvement with shared memory is 53.5% for N=512, 2.80% for N=1024, and 2.78% for N=2024.

I was surprised to see that shared memory performs better than non-shared memory. To me, it seems that an implementation where each thread is responsible for a row and column should

work better than splitting into sub-blocks and working together. Couple this with the lack of overhead of setting up shared memory matrices and I figured we would have a winner there. Runtimes are shown in figure 1.

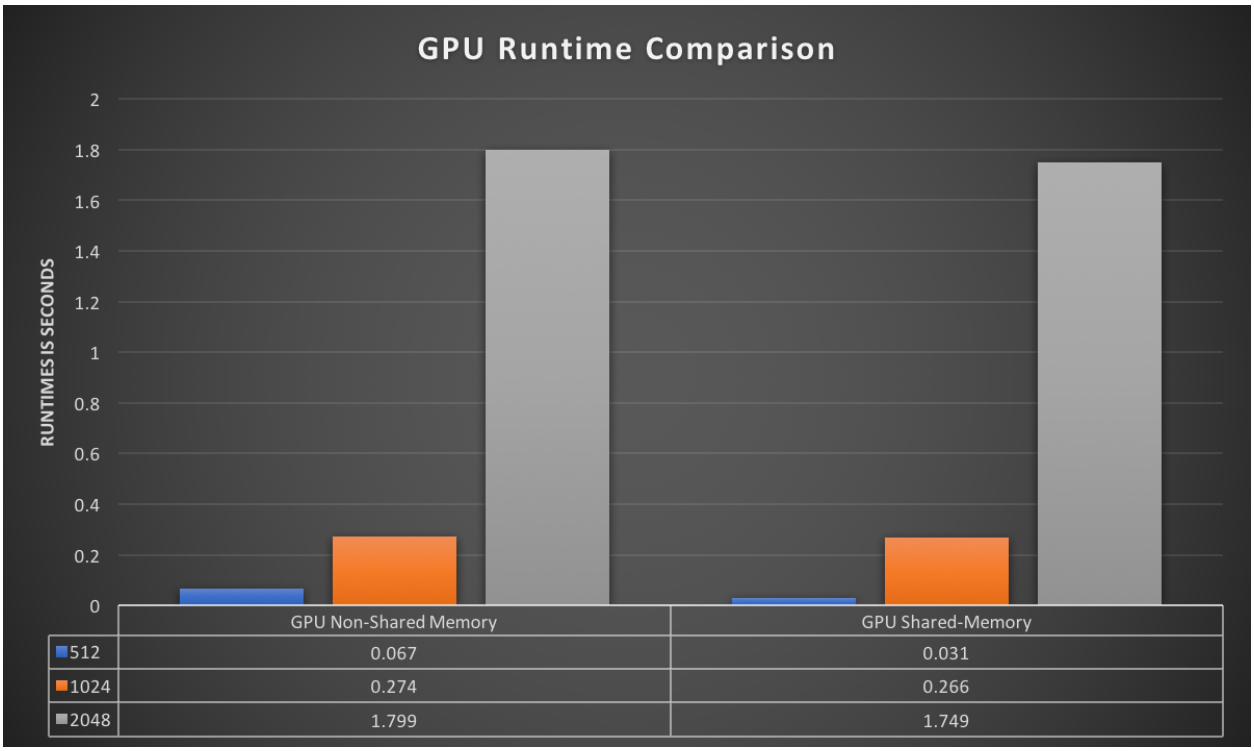

FIGURE 1: RUNTIME MEASURED IN SECONDS OF GPU CONFIGURATIONS. THIS IS THE AVERAGE OF THREE ITERATIONS. CPU RESULTS ARE OMITTED FROM THIS GRAPH BUT CAN BE VIEWED IN THE README DUE TO THE DIFFERENCES BETWEEN IT AND THE TWO GPU CONFIGURATIONS.