CSS 535 A (Winter 2020) Lab

Ardalan Ahanchi
Andrew Nelson
Rahil Mehta

Lab 2 - CUDA “Hello ~~World~~ Vector”

Introduction

CUDA is a parallel platform developed by NVIDIA which provides the ability to significantly improve the performance of certain operations by running the operations in parallel (NVIDIA, 2020). The goal of this lab was to perform vector addition on the GPU using CUDA and observe the performance on time for memory transferring and the performance of calculation.

Methodology

We wrote a shell script to run the program many times with different command line arguments. The program prints the output to the console and a text file.

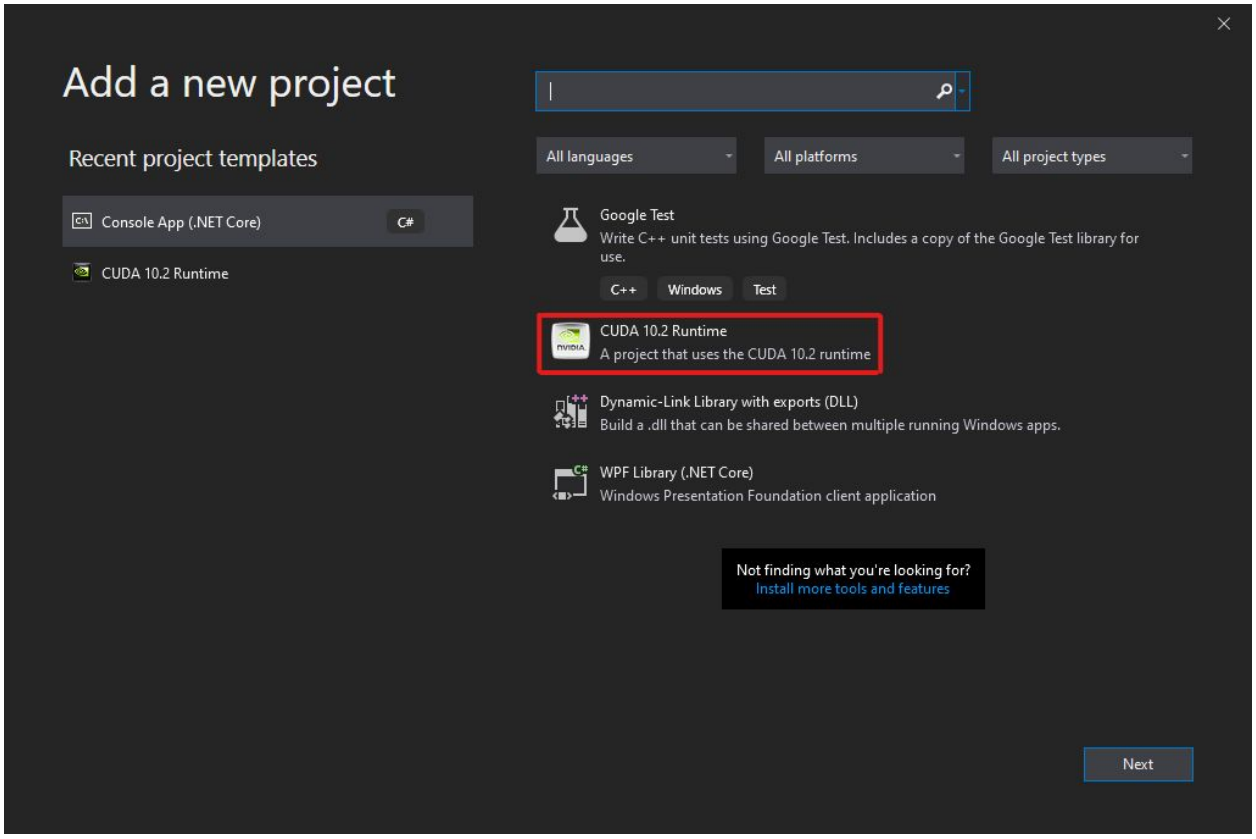
A parser was also written to parse out the text file that was generated to help provide different sub-data sets for observation.

Hardware Specification

Processor: Intel Core i7-9750h
◦ Frequency: 2.6 GHz to 4.5 GHz.
◦ Cores: 6 Physical Cores (12 logical cores using hyper-threading).
◦ Cache: 32K L1-data, 32K L1-instruction, 256K L2, 12288K L3.
◦ TDP: 45 Watts.
Graphics: Nvidia RTX-2070 MaxQ (8 GB GDDR5 VRAM, 85 Watts, Compute Capability 7.5).
RAM: 32GB DDR4 2666 MHz.

Project Configuration

The first step for each team member was to download and install the CUDA Toolkit (Link). Once the toolkit was installed, Visual Studio users were able to create a “CUDA Runtime” project to get going. Creating this project through Visual Studio creates a “Hello, World!” application for you making it pretty easy to start developing. Alternatively, for non-Visual Studio users, they were able to compile and run a new CUDA project using the NVCC compiler.

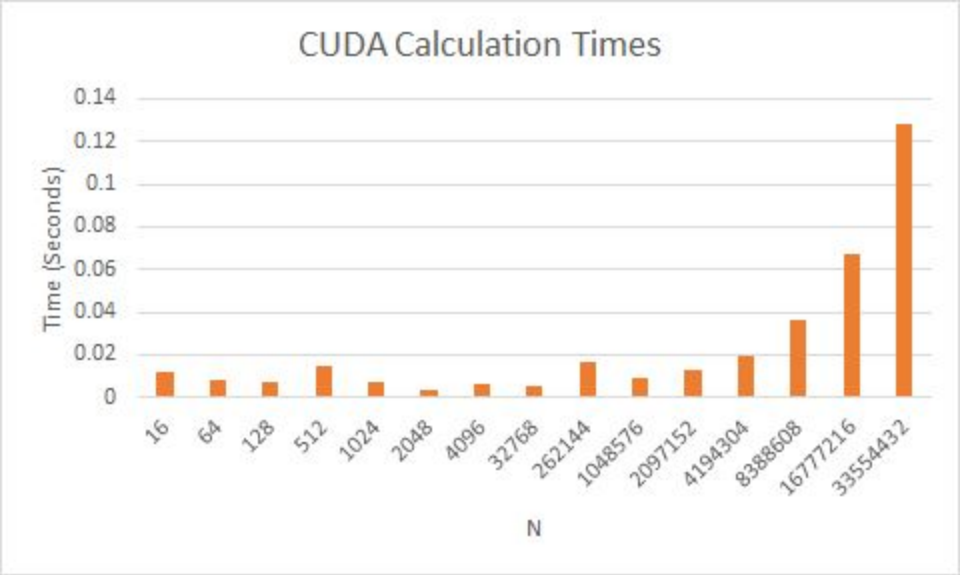


Timing Analysis:

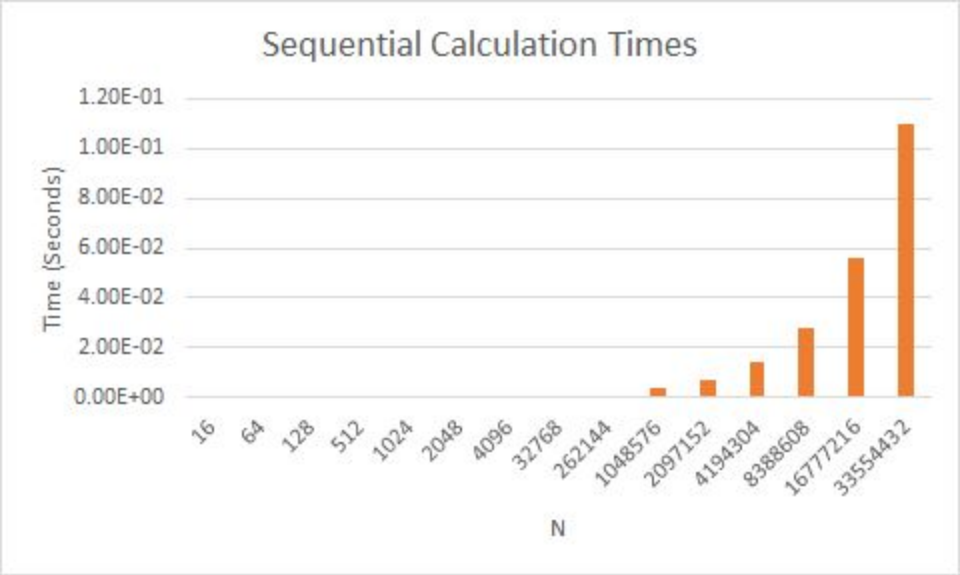
Below are a series of graphs displaying differences related to timing when running multiple tests with our team's program. These graphs include overall timing averages as well as timing results based on different configurations (blocks x threads). For the configurations used in these experiments, see Appendix A in this report.

Time Averages

The averages shown below show the overall computation time averages for both the sequential algorithm and CUDA algorithm. The data shown for the CUDA algorithm is shown for all configurations run in our experimentation.

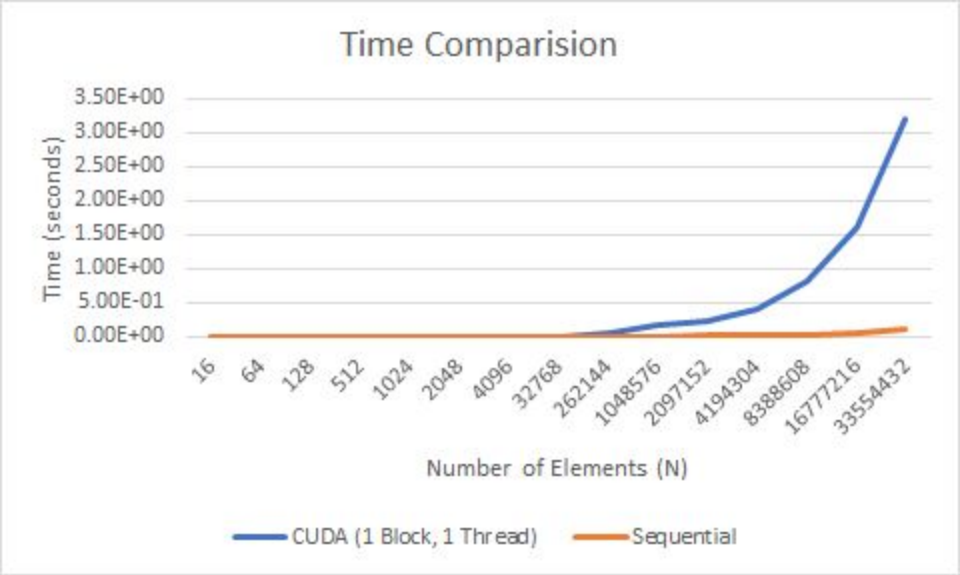


The graph above shows the average times for calculations with CUDA. The trend is increasing time for greater values of N. This is what we had expected because it takes time to process each element.

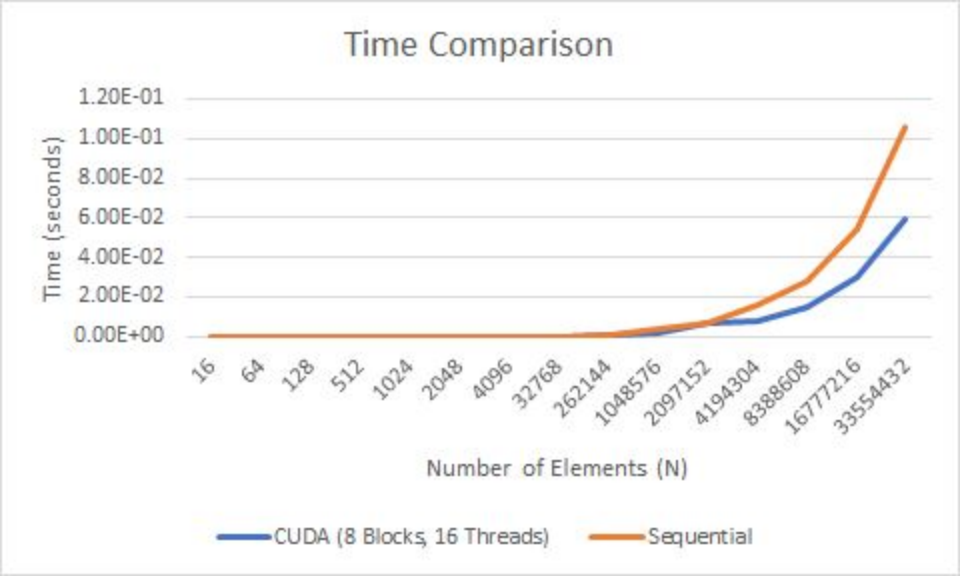


Time Comparisons

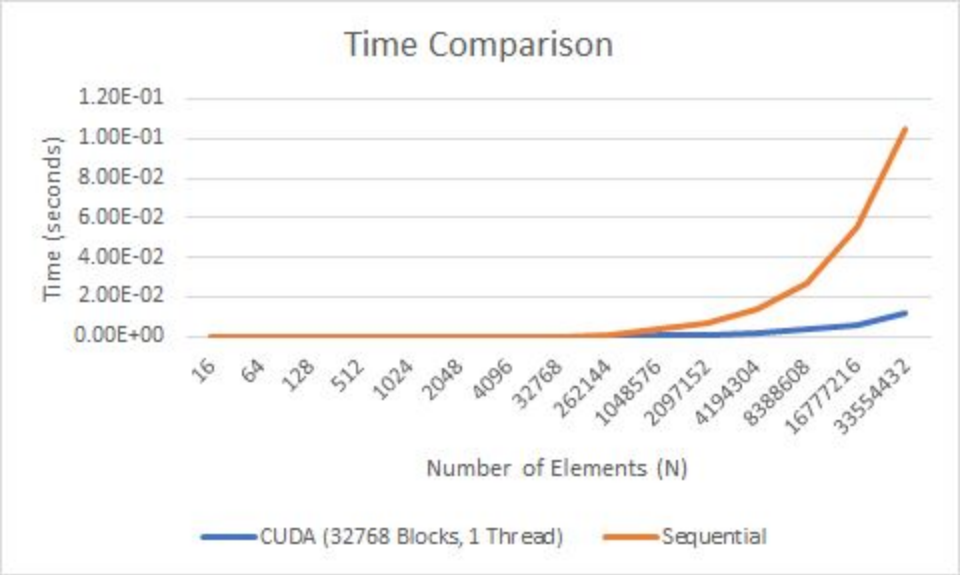
The following charts show a few time comparisons between CUDA operations and the sequential operation.



The performance of CUDA with one block and one thread is much worse than the sequential implementation. Only using one block and one thread will leave out most elements of the vectors in the calculations. It is not clear why CUDA is slower in this case.



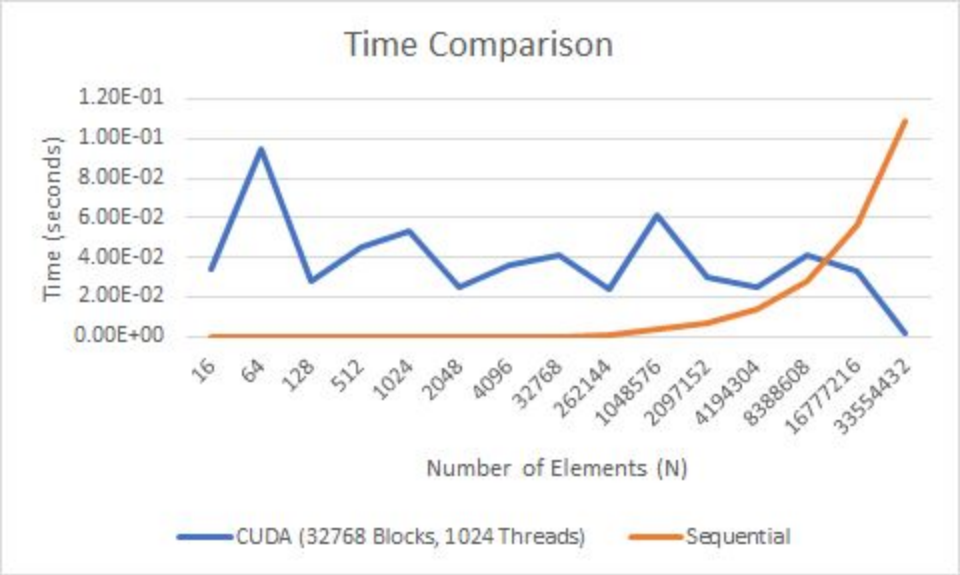
The calculations with CUDA for larger values of N are faster than the sequential implementation. Having many threads makes it possible to take advantage of the GPU's high capacity for parallelism.



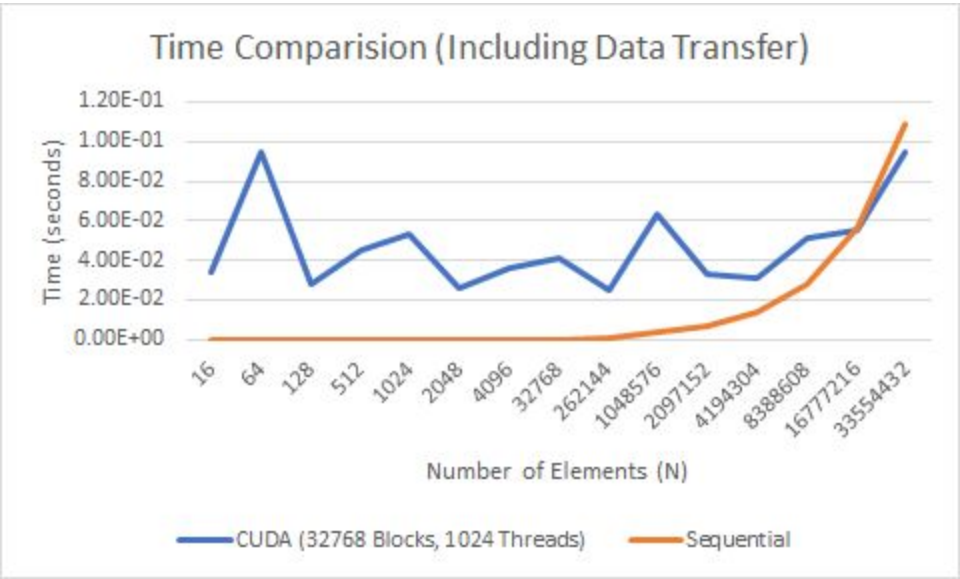
The graphs above show that it is better for performance to create more blocks before adding to the number of threads per block. The maximum number of threads is only 1024 but there can be tens of thousands of blocks.

Time Comparison Including Data Transfer

The time comparisons above strictly show the time comparisons when comparing the time of the execution done on both the device and the host. As discussed in lecture, one of the challenges of HPC that impacts performance is the time it takes to data transfer. The graphs below show the differences in overall performance when we include the time it takes for data transfer verses the calculation time without it.



In a configuration displayed in the graph above, we can see that the CUDA calculations were actually initially slower than the sequential algorithm with lower values of N; but eventually we started to see a significant improvement in performance as N gets extremely large. The reason that this configuration may be running slower with lower values of N could be due to the process waiting for each thread from each block to return, before returning the result vector back to the host.

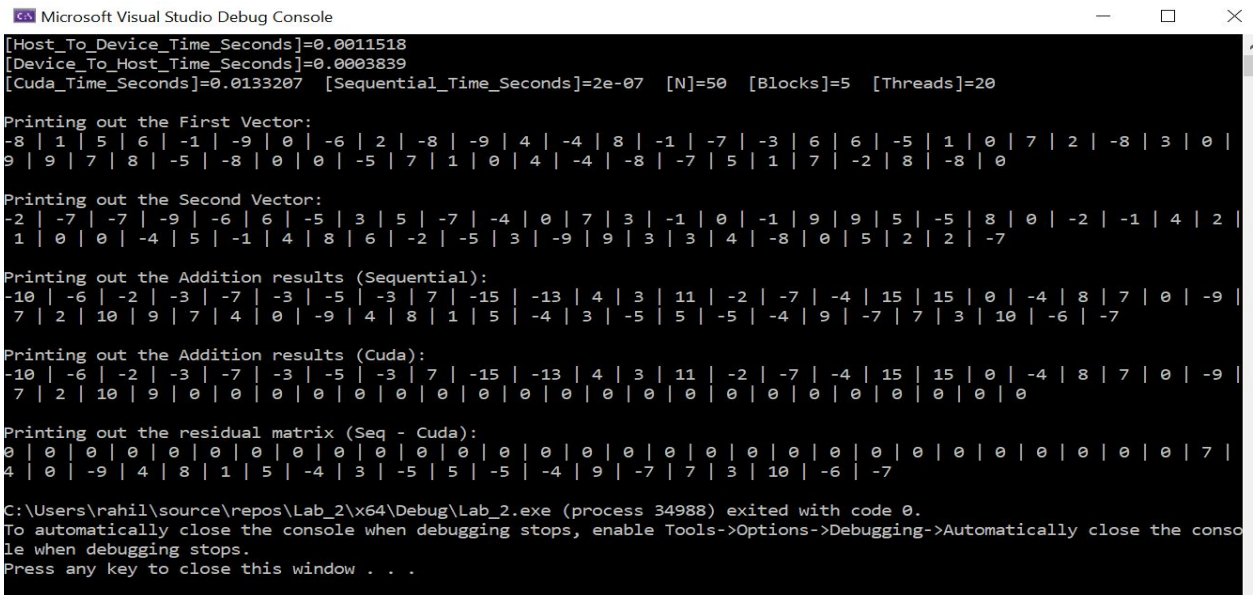


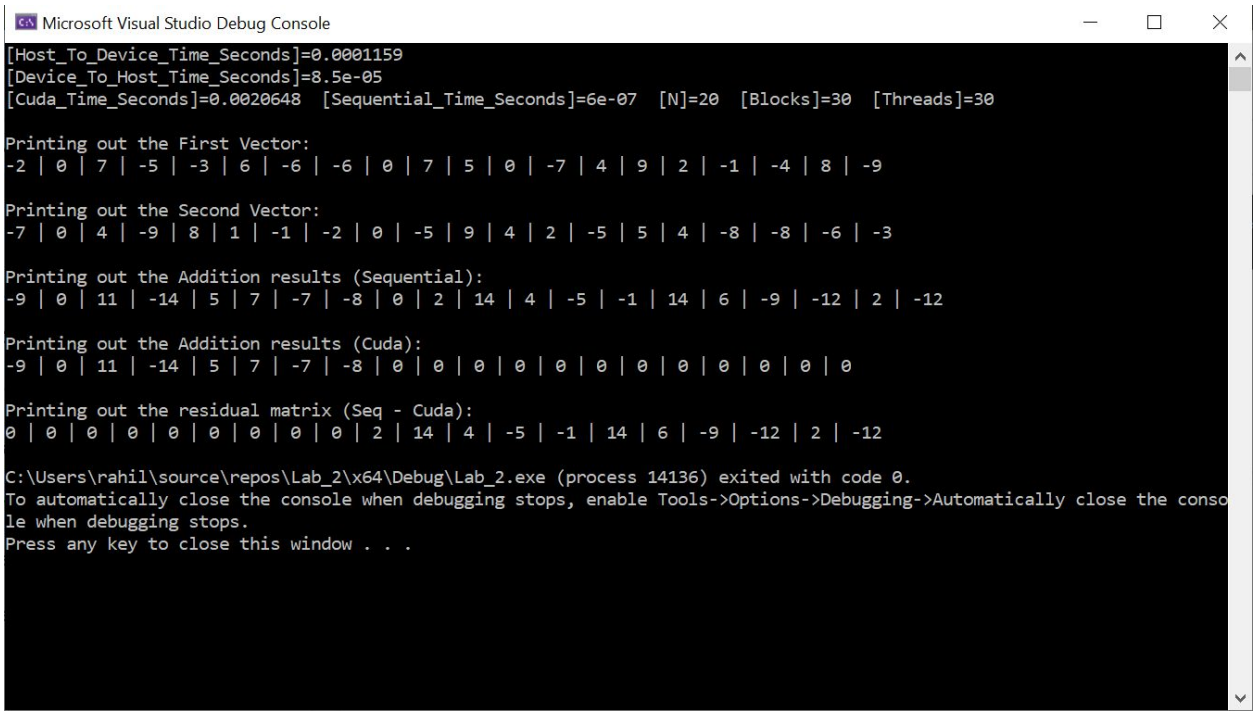
When we include the time it takes to transfer the data, we can see a similar trend in performance for both the sequential and the CUDA methods. However, although CUDA still shows an improvement over the sequential methods with large values of N, the margin of improvement is not as large. This is a great example of how the the data transfer between the

host and device can act as a huge “bottleneck” as discussed in lecture. It shows that it is important to consider data transfer time before deciding to use CUDA.

Output

Screenshots of the program running:





References

1. NVIDIA Developer. (2020). *CUDA Zone*. Available at: https://developer.nvidia.com/cuda-zone

Appendix A - CUDA Configurations

Our program was able to receive the thread and block configurations via the command line. With this, a shell script was created to help generate different configurations to help obtain test results.

The following block sizes were used: [1, 8, 32, 128, 512, 2048, 8192, 32768]
The following thread sizes were used: [1, 4, 16, 64, 256, 1024]

Each block size was tested using each thread size mentioned here.