Programming Assignment Report

PA1: Matrix Addition

Chad Adams

CS791

Due: January 30th, 2018

1. Introduction

In this assignment we were tasked with adding two NxN matrices together, where N is at least equivalent to 1000, using several different methods. The first method used was the sequential method used in standard C programming with each element being iterated through and added to the corresponding element in the second array. The second method was using gpu parallel processing without striding, with each element being assigned a single thread that lies within its own block. The third method was gpu parallel processing with striding, allowing multiple threads per box.

2. Methodology

2.1. Sequential

The sequential implementation was a straightforward affair, creating two 2D arrays and looping through them using nested for loops to add them together. The arrays were initialized with the value of their ‘x’ and ‘y’ indexes multiplied together, to give each individual a unique value.

2.2. Parallel

The parallel implementation required only a few additions to the sequential implementation in order to enable parallel computing on the gpu without using striding, adding input variables for the number of blocks and threads to be used and an if statement separating the sequential and parallel implementations. The parallel implementation that used striding required modifying ‘add.cu’ with the 1D grid of 1D blocks striding formula from the provided cheat sheet at: https://cs.calvin.edu/courses/cs/374/CUDA/CUDA-Thread-Indexing-Cheatsheet.pdf, keeping the same implementation for the add function otherwise. This choice required looping through the data assignment to the cuda memory and call to the add function. This choice in hindsight is most likely responsible for the significant slowdown between the sequential and parallel implementations that will be discussed later in this paper. Either initializing the arrays on the GPU, sending a 2D array to the GPU once instead of a 1D array N times, or both would have possibly caused a significant speed up in run time.

3. Results and Discussion

I ran the sequential and non-striding parallel implementation each once, while I ran the striding parallel implementation multiple times each time using varying numbers of blocks and threads per block. As can be seen in the table below the sequential

implementation performed at orders of magnitude faster than the parallel implementation. This can most likely be explained by my looping through the data transfer between the GPU and CPU rather than trying to minimize the number of data transfers. Of note is that the run that used ‘50’ blocks and ‘5’ threads per block performed the best of the striding runs, possibly pointing towards a favourable number of threads to block ratio for the cubix machines.

Type of Run	Blocks Used	Threads per Block	Time Taken
Sequential	-	-	6.3 ms
Parallel (non-striding)	1000000	1	7778.8 ms
Parallel (striding)	100	20	6678.3 ms
Parallel (striding)	200	10	6656.5 ms
Parallel (striding)	200	20	6664.7 ms
Parallel (striding)	500	10	6675.3 ms
Parallel (striding)	5	100	6672.9 ms
Parallel (striding)	5	5	6673.4 ms
Parallel (striding)	1000	1000	6668.2 ms
Parallel (striding)	50	5	6645.3 ms

4. Conclusion

In this project we explored the implementation differences between adding 2D vectors together sequentially on a CPU and in parallel on GPUs. Multiple block and thread structures were used to test the parallel implementation in order to compare with the sequential version. This project demonstrated the speed loss from data transfer between the CPU and GPU well. With more time and more CUDA techniques the data transfer could be limited to show that the GPU adds numbers faster than the CPU.