Programming Assignment Report

# PA2: Matrix Multiplication

Chad Adams

CS791

Due: February 6th, 2018

# 1. Introduction

In this assignment we were tasked with implementing matrix multiplication while using shared cuda memory. The matrices had square dimensions of NxN with N having a minimum of 1000. First we were required to implement the matrix multiplication sequentially with all of the calculations being done on the CPU. Next we needed to implement the parallel version of matrix multiplication using CUDA and then compare the performance of the two approaches.

# 2. Methodology

### 2.1. Sequential

The sequential implementation was a fairly straightforward extension of the matrix addition from the previous assignment. As such the performance achieved was also fairly similar. The main difference between matrix addition and multiplication was that the elements of the given row from the first matrix are multiplied with the elements of the given column of the second matrix (e.g. first row of first matrix is multiplied with first column of second matrix).

### 2.2. Parallel

The parallel implementation was significantly different in that in order to implement shared memory we needed to use cudaMallocManaged to assign memory to our declared arrays. As the arrays used were 2D in nature, cudaMallocManaged was also needed in order to dynamically assign the memory of the first order array indexes. The striding within the GPU function also needed to be modified in order to handle arrays that could be bigger than the maximum number of blocks and threads that could be assigned.

# 3. Results and Discussion

The results were as expected, with the parallel implementation outperforming the sequential one by a significant margin. This may in part be due to the shared memory implementation which removes the costly memory transfer between the CPU and GPU. No specific block to thread ratio seemed to render significantly greater performance and increasing the matrix size had a marked increase in run time for the GPU.

| Matrix Dimensions | Sequential Running Time |
|---|---|
| 1000x1000 | 10.32ms |
| 1000x1000 | 11.51ms |
| 1000x1000 | 10.31ms |
| Average for 1000x1000 | 10.72ms |

As can be seen, the overall throughput of the sequential implementation was at about 93 million arithmetic operations per second, or 93 thousand per millisecond. Contrasting that with the performance of the GPU shown in the table below provides an example of why GPUs are prefered for large scale numeric computations.

| Matrix Dimensions | Block Number | Threads per Block | Running Time |
|---|---|---|---|
| 1000x1000 | 5 | 2 | 48.73ms |
| 1000x1000 | 256 | 32 | 2.89ms |
| 1000x1000 | 512 | 64 | 2.84ms |
| 1000x1000 | 1024 | 128 | 2.88ms |
| 1000x1000 | 2048 | 256 | 2.89ms |
| 1000x1000 | 8192 | 1024 | 2.93ms |
| Average 1000x1000 | - | - | 2.89ms |
| 4096x4096 | 2048 | 1024 | 36.49ms |
| 4096x4096 | 1024 | 512 | 37.34ms |
| 4096x4096 | 1024 | 256 | 36.13ms |
| Average 4096x4096 | - | - | 36.66ms |

The throughput of the GPU comes in at about 346 million arithmetic operations per second, or 1.7 million per millisecond. The speedup factor over the sequential implementation is roughly 1.87 times faster.

## 4. Conclusion

In this project we explored the application of shared CPU-GPU memory and its benefits. We compared both the sequential and parallel shared memory implementations of Vector multiplication in order to see the performance difference. What was shown was that the parallel implementation was roughly 1.87 times faster than the sequential version when using shared memory.