CS492B Assignment 2

Karl Gylleus
20166041

April 28, 2016

1 CUDA implementation of Gaussian Elimination

This CUDA program takes as input the width n of the A matrix and the number of CUDA threads to be contained in a single block. The number of blocks needed to cover the entire A matrix is then calculated from this.

There are two kernel functions. One for partial pivoting and one for the creation of the upper triangular matrix. Since I found no way to synchronize all threads in the grid within the kernel the kernel functions have to be invoked iteratively by the CPU; once for every row iteration.

1.1 Partial Pivoting Function

The partial pivoting function is not parallelized, but is instead run by a single GPU thread. This might be a quite poor implementation idea, however. I tried writing a parallelized function for this, but finding the maximum float value in the column proved difficult when using multiple functions. Using atomic functions would be a good idea, but they were only supported for integer values which would make me lose precision. Thus I tried to parallelize only the swapping of the rows but ended up with unexpected values. I did not find time to correct this error and instead settled for a single thread implementation.

The reason that I have a kernel function for partial pivoting instead of running it on the CPU is because copying the memory back forth from the CPU and GPU would be too costly.

1.2 Upper Triangular Matrix Function

The function for the upper triangular matrix is however highly parallelized. Each thread is responsible for updating one value in the matrix, and each thread responsible for the end of each row will also update the value in vector B. If a threads index is outside the bounds of the matrix, or if it belongs to an index that should not be updated in this iteration, it will immediately terminate. A small optimization would be for the CPU to calculate exactly how many threads would be needed for each kernel invocation, since they decrease for each row iteration. This would reduce overhead of thread creation, but I believe that it would only be a minor performance increase.

Inside the function the blocks have some shared memory. Since each element on the same row will use the same m value when updating, a vector of m values can be shared within the block with a certain value for each row. Furthermore each thread in the same column will use a certain value when updating their index, which should thus be made shared.

After the functions is completed the result is copied back to the CPU, after which it performs the backwards solving single threadedly.

1.3 Encountered Problems

For some reason the result of the program has a high error factor in the result. At low matrix sizes the result seems very correct, but the larger the matrix gets the larger the error also becomes. First I thought that this was due to a synchronization error between kernel calls, so that smaller matrices had time to complete between each kernel invocation but larger matrices do not. This does not seem to be the case, however, since synchronizing between kernel calls had no effect on the result at all.

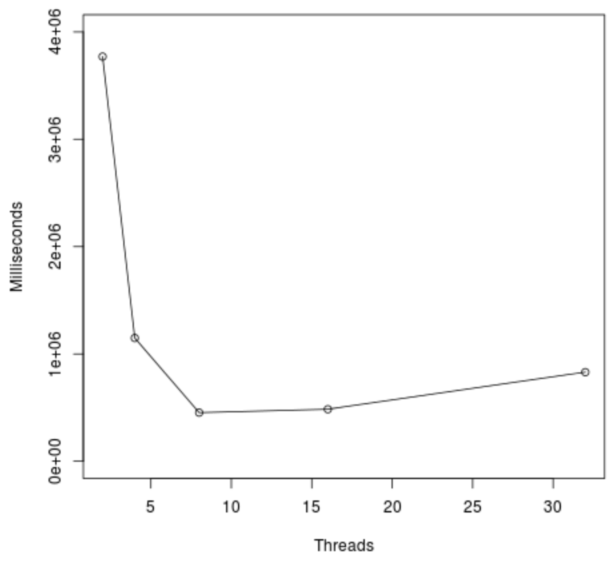
I then thought that it was an error when partial pivoting. But through testing I have concluded that the partial pivoting works as it should. The resulting A matrix is always an upper triangular one, so the error is not easy to spot. Even after hours of testing I can not conclude the reason for this error.

There also seems to be a huge performance error in this implementation as it runs very slowly. I have tried to find what the bottleneck is through profiling, but what I find does not make sense to me. The reason seems to be that invocing the kernel takes much longer time than actually running it. When profiling on a matrix size of 1000x1000 each kernel execution took 660us, but total time of the CPU invoking the kernel was 6160us. This thus slows down the program by a factor of almost 10, which is enormous. I believe that this is caused by the overhead of invoking the kernel function, which is why I wanted to perform the entire creation of the upper trianguar matrix within one kernel call, but was unable to.

2 Results

The following results are run on matrix size $n = 8000$. The value n represent the number of threads $(n \cdot n)$ per thread block.

n	Execution time (ms)
2	3770258
4	1148620
8	453562
16	485211
32	830731



The overall performance of the CUDA solver appears low due to the problem mentioned earlier. However, the effect of the parallelism can clearly be seen in the graph. When testing, the best block size seems to be 8x8 since it had the fastest execution.