Filip Zachrisson
970820-9110

Fredrik Lind
960325-4195

Lab 2 - Gaussian Elimination

DV2581 - Multicore Programming

Acknowledgements

- Terms that we will be using that refers to our program:
    - MAX_SIZE - Width and Height of the matrix.
    - NUM_THREADS - Total number of threads to be used.
    - BLOCK_SIZE - Number of threads per block.
- We were measuring the data by comparing our own host-code (not the given code) with our device-code.
- All output was disabled when gathering results.
- All testing was done on the school computers (GTX 1080).
- Used command in Windows PowerShell:
    - *Measure-Command {start-process '.\MrGaus.exe' -Wait}*

Introduction

This report will explain how went about implementing Gaussian Elimination on the GPU by using parallel programming with CUDA.

Implementation

The code works by diagonally looping through the given matrix (eg, *(1,1), (2,2), (3,3) … (k,k)*) on the host, which means that each diagonal value is a new iteration. Here, each iteration calls the device twice.

The first kernel-call gets looped k times and does the gauss elimination each k by using threads as rows. We calculate a factor f that this row should be divided by with an iteration k, then divides all the elements in the matrix as well as y. Then after the first kernel-call, the matrix has 0's in a lower triangle. The second kernel-call then divides all the rows with its diagonal (pivot) value.

Now to solve the linear equation we need to back substitute the equations. We have a function that does this but this is not used because it was not included in the assignment.

Method

First we transformed the given matrix to an 1D-array where the length is the width of the matrix multiplied by the height of the matrix (MAX_SIZE * MAXSIZE), while also creating a function which receives x- and y-values and returns the proper position of the 1D-array. This made it easier to allocate memory for the GPU and send the array to the it.

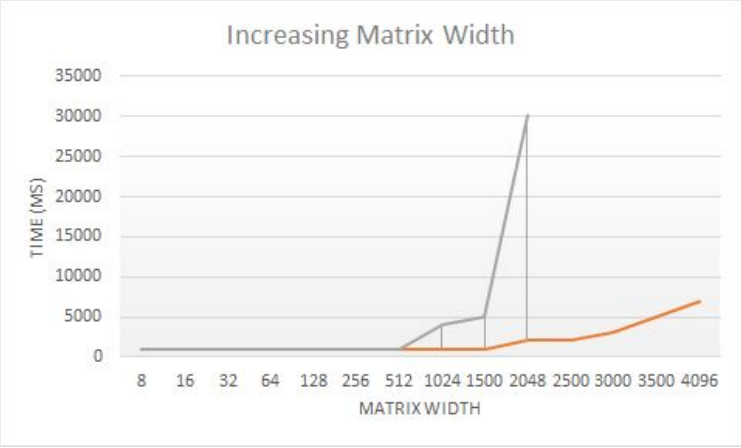
The next step was to copy the given Gaussian Elimination code and paste it in the device while changing it to work in parallel. However, we did not seem to get it to work so we decided to create our own Gaussian Elimination from scratch, where we first created the code for the host and when we acknowledged that our code gave the same results as the given code, we moved on to create it on the device.

The third step was to get the code to work by using “Thread Coarsening“. This was easily done by just implementing a while loop where the x-value has to be smaller than the length of one row of the matrix (MAX_SIZE).

The final step was to optimize the code by using shared memory. But we had to use thread coarsening to be able to fill it with the correct data.

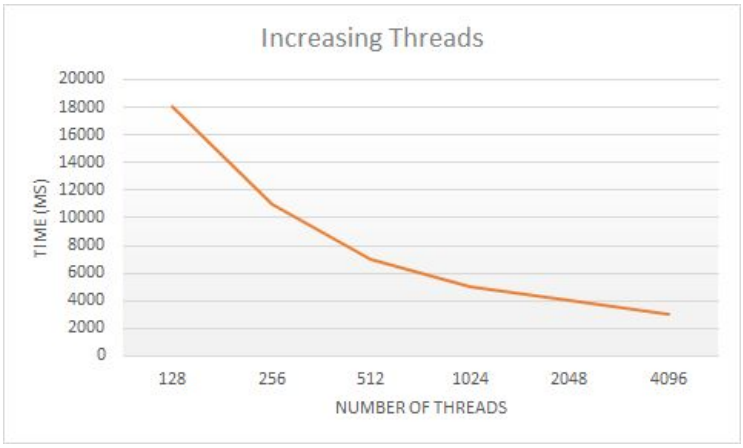
Findings

Figure 1. A time comparison between the CPU and GPU, when increasing the matrix size, while using 4096 variables and 128 threads per block.



We gain a tremendous speed up when using the GPU. This can clearly be noticed when using 1500 variables or more where the CPU-time skyrockets

Figure 2. Total time when increasing the number of threads on the GPU, while using 4096 variables and 128 threads per block.



The time decreases exponentially when we are increasing the total number of threads.

Result

The Gaussian elimination was faster on the GPU than the CPU which is clearly demonstrated in *Figure 1* and *Figure 2*.