Filip Zachrisson
970820-9110

Fredrik Lind
960325-4195

Lab 1 - OddEven Sorting

DV2581 - Multicore Programming

Acknowledgements

- Terms that we will be using that refers to our program:
    - Step - The stride of thread coarsening.
    - ARRAYSIZE - number of ints to be sorted.
    - NUM_THREADS - total number of threads being used.
    - BLOCK_SIZE - number of threads in each block
- The program saves the array in a file called “gpuSorted.txt”, there is also commented out code that sorts the array on cpu as well as some debugging code that is also commented out.
- Flag -noFile, prevents the program to save to file “gpuSorted.txt”.
- The testing was done by using an NVIDIA GTX 1080.
- The increasing array size testing was done with a BLOCK_SIZE = 128, NUM_THREADS = 8192.
- The increasing number of threads testing was done with BLOCK_SIZE = 128, ARRAYSIZE = 50000.
- Used command in Windows PowerShell:
    - *Measure-Command {start-process ‘.\Multicore Lab 1.exe’ “-noFile” -Wait}*

  which times our program that runs without saving any result.
- The code works for any number of ARRAYSIZE.

Introduction

This report will explain how we went about creating a program which is able to sort huge integral arrays on the GPU using CUDA and compare the results between sorting on a CPU verses sorting on a GPU.

The sorting is done by using the OddEven-sorting technique.

Implementation

We implemented our OddEven sort by letting any number of threads be used with any number of blocks (maximum supported by GPU), the problem we had to solve in order for this to work is to somehow synchronize all the blocks for each odd/even phase, we accomplished this by doing more than one kernel call from the __host__ which then functions as a sync-barrier. We call first

EvenSort then OddSort. In order for the code to run on fewer threads than ARRAYSIZE/2, we implemented thread coarsening.

Method

We started by creating the OddEven sorting algorithm on the _host_, this was because we wanted a comparison. After that we started to get CUDA working by using memcpy and the kernel call. Then we simply wrote a simple OddEven code on _global_ which calls _device_ whenever a swap is needed. The _device_ has the swap-part of the code.

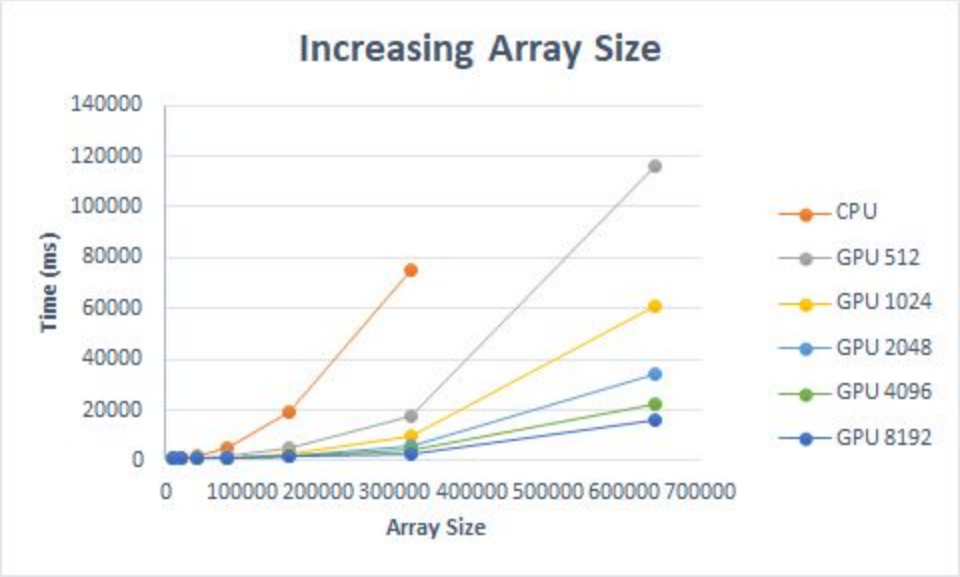
We attempted to sort but we did not use any thread coarsening which allowed us to only sort lists with array size of 1024. Therefore, we added thread coarsening on the _device_ part of the code, which allowed one thread to do more job than one pair inside a sort phase. Though, we also had stackoverflow since we allocated some arrays on the stack, so instead, we allocated this data on the heap to prevent it. Furthermore, we noticed that the rand() function has a RAND_MAX of 32767 so we switched it to a std::random_device with a std::uniform_int_distribution which gave us a bigger maximum and better spread on the values. This allowed us to have a maximum array size of 65535 before the GPU runs out of space since we used block size = 1.

We did not think at the time that __syncthreads() only synchronize the threads and not the blocks. To allow usage of more blocks we had to sync the blocks at the cpu by moving the loop inside the _global_ to the _host_, while splitting the kernel-function into two kernel-calls, one for “even” and one for “odd”. This allowed us to use more blocks which increases the total number of threads used.

Because we could only run the program with a limited ARRAYSIZE, we had to rethink and find the problem and it seems like we had misunderstood how you are supposed to program the GPU. Before we had kernel calls as following <<<NUM_BLOCKS, NUM_THREADS>>> which is not very good since a SM only generally have 128 cores, which means that if we input NUM_THREADS > 128, there is a high chance for the memory to go out of bounds. We now have a kernel call as following <<< ceil((float)NUM_THREADS / (float)BLOCK_SIZE), BLOCK_SIZE >>> which makes sure that we have as many threads per block as we need.

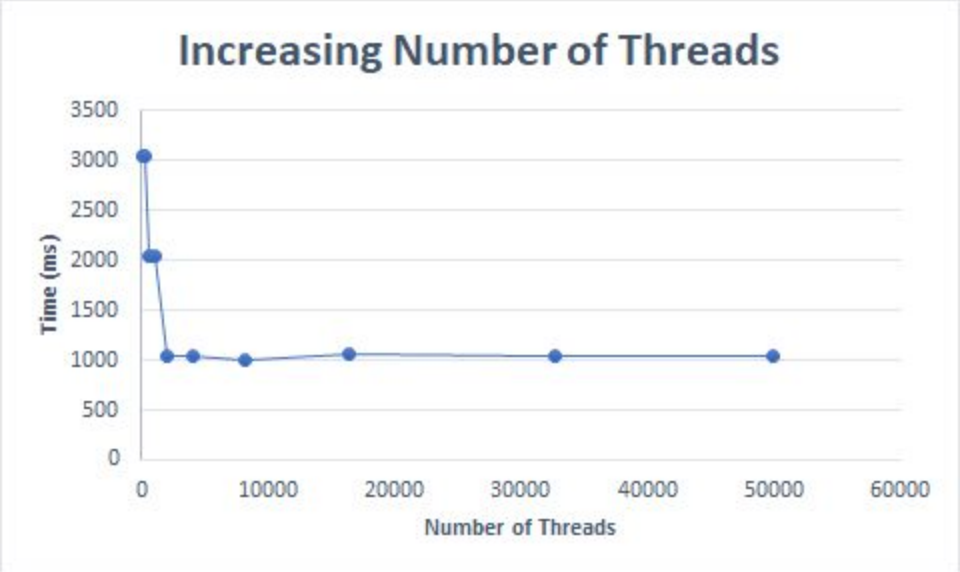
Findings

Figure 1. A time comparison between the CPU and GPU, when increasing the array size. The number to the right of GPU is how many threads were used.



We gain a tremendous speed up when using the gpu.

Figure 2. Total time when increasing the number of threads on the GPU.



As the number of threads increases, the runtime decreases.

Result

The data from *Figure 1* shows that as the number of threads used approaches the array size, we get better performance. This is probably because of the thread coarsening since there will be a serialization with thread coarsening which hinders the GPU from working as efficient.

Figure 2 shows that using too few number of threads will increase the runtime with a huge amount.