Multiprocessor Programming DV2606

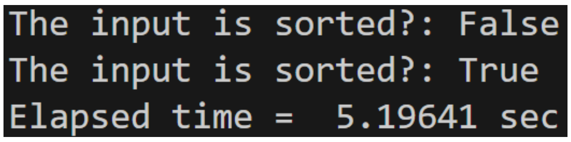
Odd-even transposition sorting

Odd-even sorting algorithm works by swapping numbers with the one after the index in focus. It varies in two phases, an odd phase and an even phase. Where the odd phase uses all odd indexes as the first value to check if it should be swapped and the same method applies for the even phase. It is basically a bubble sort that is performed in two phases alternating on odd and even. It is a sorting algorithm that greatly benefits from multithreaded and parallel computation.

Task One – Parallel Odd-even sort using a single kernel launch

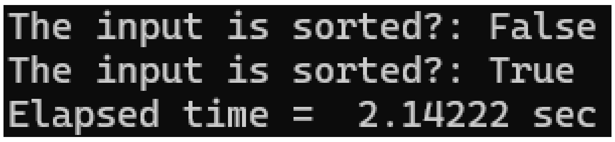
For the first task we decided to parallel with one kernel launch that was a block of 1024 threads. The reason we chose 1024 is the upper limit of hardware for one block. Then we allocated memory for GPU and copied over relevant data to it with cudaMalloc() and CudaMemcpy(). Following, the kernel launched and started the algorithm. To make sure that we still followed the odd-even structure we calculated odd even indexes for all threads assigning work in a cyclic pattern. After each phase we used __syncthreads() to make sure that operations on data was controlled since we used shared memory between all threads. We also needed to rewrite the swap function threads to be able to use it since this built in function is not accessible for GPU threads.

The results from parallelisation with one kernel launch with 1024 threads showed great improvement over sequential odd-even sorting with a speedup of 2.4X. The results from sequential sorting were 5.2 seconds while parallel single kernel sorting was 2.14 seconds.



```
The input is sorted?: False
The input is sorted?: True
Elapsed time =  5.19641 sec
```

Sequential Odd-even sort



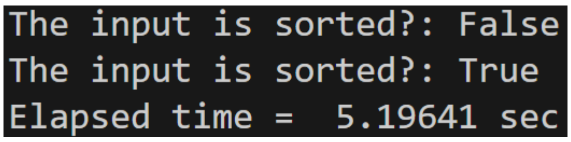
```
The input is sorted?: False
The input is sorted?: True
Elapsed time =  2.14222 sec
```

Single kernel Odd-even sort

Task Two – Parallel Odd-even sort using multiple kernel launches

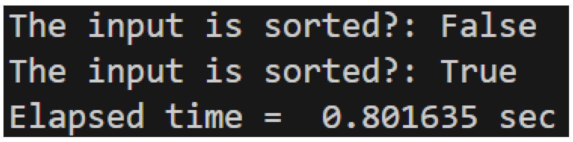
The main difference between task one and task two is where the synchronisation happens. Since we use multiple kernel launches, we can have multiple blocks of 1024 threads. This allows for each thread getting their own task and we can remove the cyclic task assignment that we had in task one. The loop for each odd/even phase is now moved to CPU level and is used as a synchronisation step, this is done because we cannot synchronize between threads that are in different blocks/kernels, but we still need to keep data intact since the GPU uses shared memory space.

The results from utilizing multiple kernel launches are not that surprising. Since we get even more threads to work with, we get even more speedup then previously in task one. This time the time to sort is 0.8 seconds which is a speedup of 6.48X compared to the sequential sorting. Even compared to task one we get a speedup of 2.6X, which is great improvement.



```
The input is sorted?: False
The input is sorted?: True
Elapsed time =  5.19641 sec
```

Sequential Odd-even sort

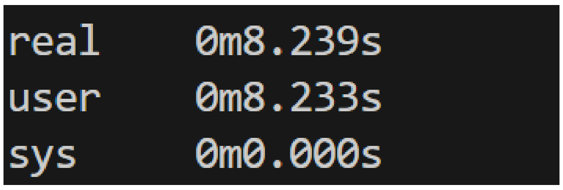


```
The input is sorted?: False
The input is sorted?: True
Elapsed time =  0.801635 sec
```

Multi kernel Odd-even sort

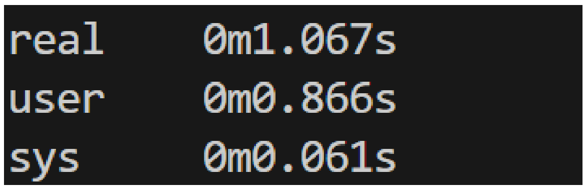
Task 3. Parallel Gauss-Jordan row reduction

The problem presented in this task were to parallelize the Gauss-Jordan row reduction using Cuda, which is pretty similar to the previous assignment. When programming with Cuda, instead of having a for loop that iterates over a specific range, the Cuda uses the amount of threads set in the block and then limits which of the threads that gets to do work by using if statements.
The first thing we did was to move the division step and the two elimination steps are into separate functions. The reason for this is to be able to launch them as kernels and parallelize them. The kernels are launched with a thread and block configuration where the threads per block is set to 1024 and the number of blocks scale accordingly. Then the matrix A and the arrays b and y got copied into the device/graphics cards memory. To simplify the structure when copying from host to device and vice versa, the matrix A was converted from a 2d matrix into a long one-dimensional array. This meant that instead of accessing a specific cell like A[i][j], it instead got accessed by A[i * N + j] where N is the size of the original matrix. Because we use a 1d array we only need one thread index, but because it still should be treated as a matrix, from the thread index we derived two new indices i and j which represent the row and column respectively. In each kernel there's two sections where it's important that all threads are done with the first section before continuing, to achieve this we added a __syncthreads() function between the sections.
Below are the results from the sequential and Cuda parallelized programs for a 2048x2048 matrix. As can be seen the parallelized version achieved 8x performance over the sequential version.



```
real    0m8.239s
user    0m8.233s
sys     0m0.000s
```

(Runtime for *Sequential Gauss Elimination)*



```
real    0m1.067s
user    0m0.866s
sys     0m0.061s
```

(Runtime for Cuda parallel implementation)

Instructions to run the code

To run the different programs, we've made a makefile.

To run task one: **make run_oddeven1**

To run task two: **make run_oddeven2**

To run task three: **make run_gaussian**