Applied GPU Programming - Assignment II

DD2360 HT20

Pontus Asp

November 10, 2020

1 Git repository

I uploaded my git repository to GitHub. I use the same git repository for the entire course but the folder structure requested is still followed under the root folder. I also have 2 extra directories, one for this report and one where I have code from lectures.

Here is the link to my git repository:
`https://github.com/pontusasp/kth-dd2360/tree/master/Assignment_2`

2 Exercise 1

My output from my program was `Hello World! My threadId is %d`, where `%d` was the thread id. The prints did not come in the "right" order, but that was according to my expectations, since the execution on the GPU is in parallel. I compiled my source code with the `nvcc` command without any special parameters (`nvcc exercise_1.cu -o bin/out`). I compiled and ran my software in Ubuntu 20.04 and used a Nvidia GeForce GTX 1070 graphics card.

CUDA Threads are threads running the functions on the GPU (kernels). When you launch a kernel from the code, you specify how many thread blocks and how many threads per block that the GPU should do the work with. The number of CUDA Threads is the number of thread blocks times the number of threads per block, and each and everyone of these threads will run the kernel.

3 Exercise 2

I solved the issue of the `ARRAY_SIZE` not being a multiple of the `BLOCK_SIZE` by using the method mentioned in the lecture material, namely by doing `(ARRAY_SIZE + BLOCK_SIZE - 1) / BLOCK_SIZE`. This works with the help of integer division, which makes `(BLOCK_SIZE - 1) / BLOCK_SIZE` equal zero. So then we can make sure that if `ARRAY_SIZE` is not a multiple of the block size we get one extra CUDA thread per block to make up for it so we will not miss any of the work.

When varying the `ARRAY_SIZE` from small to large I could see that the smaller it was the better the CPU performed against the GPU. While increasing the size of the `ARRAY_SIZE` the GPU performed better compared to the CPU the bigger it was. This is because of the latency of launching kernels and transferring data to the GPU. This means that if the amount of parallelization possible is small (for instance a small amount of data that should be processed) the GPU usually performes worse than the CPU, while the opposite is true if the amount of parallelization possible is a lot. The small vs large `ARRAY_SIZE` is a good example of this.

4 Exercise 3

I did a number of different test runs on the CPU and GPU. Each time measured in the graphs is an average out of five test runs measured with the system clock directly in the programs. I ran these tests on my home PC which runs on a Nvidia GeForce GTX 1070 graphics card and an Intel Core i7 7700k processor, the OS used was Ubuntu 20.04.

The observations I made when the number of particles was increasing was that both the CPU and GPU of course had an increased execution time. However the execution time was increasing drastically faster on the CPU than the GPU. If we compare the 10k particle run on the CPU (Figure 1) and the 10k particle run on the GPU (Figure 2a) the execution times does not differ that much. However if we look at the 10000k (10 M) run we can see that there is a big difference between the execution time of the CPU (Figure 1) and the GPU (Figure 3b).

In the GPU graphs there are also 5 different measurements for a few different block sizes. In these tests it is clear that there is no clear optimal block size. For insteance in the 10k run in Figure 2a the optimal block size is 128, and the worst one is 16, while if we look at Figure 3b the results are the opposite. Out of the 4 different graphs the only ones that seem to agree on the performance of the different block sizes are Figure 2b and Figure 3a.

If the simulation could not be fully offloaded to the GPU but some parts had to be calculated on the CPU then I am guessing that the observations made here would not hold. If we for instance needed to copy data from the GPU to the CPU between every simulation step then the memory copy and latency would become much more apparent and slow the program down a lot compared to only copying the data in the start and the end. We would probably need to synchronize the code as well which could also slow down the progression.

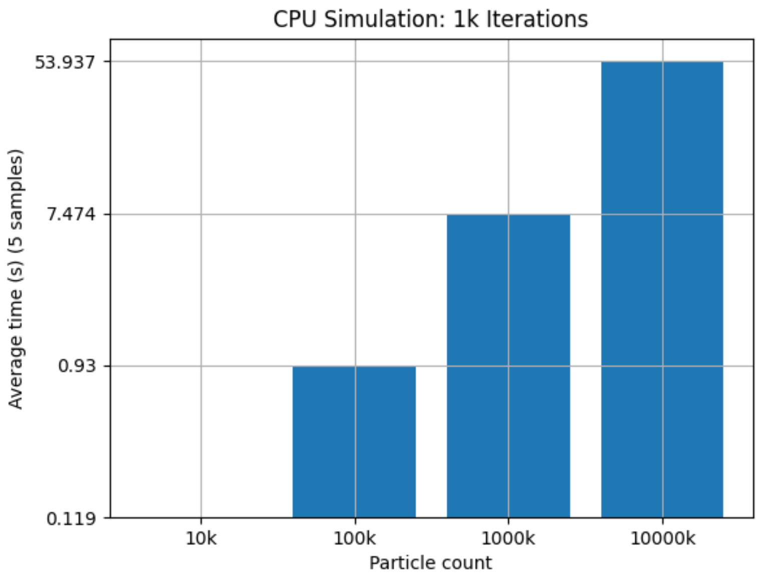
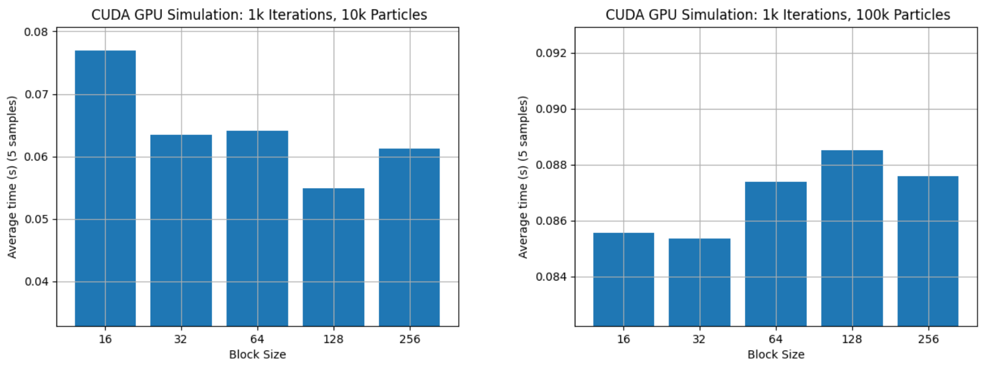
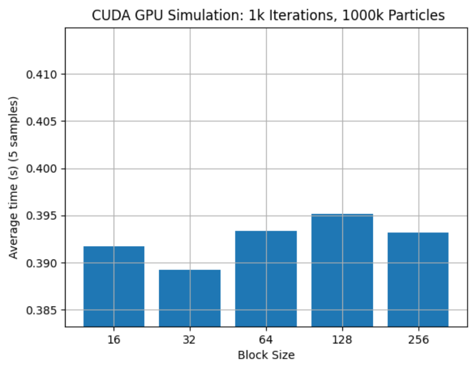


Figure 1: CPU measurements.

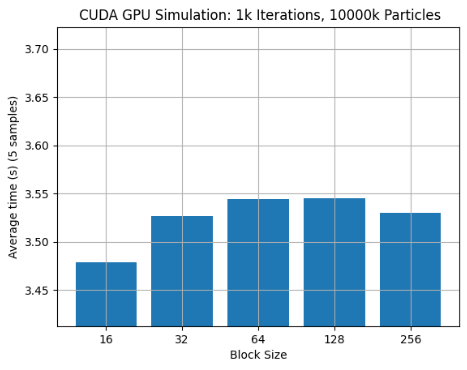


(a) 10k (b) 100k

Figure 2: GPU measurements for 10k and 100k particles.



(a) 1000k

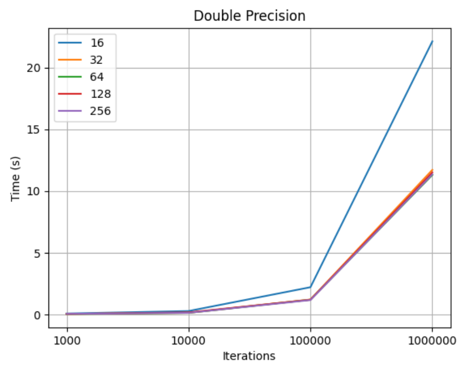


(b) 10000k

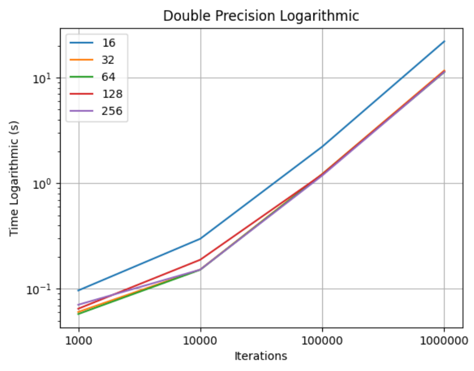
Figure 3: GPU measurements for 1000k and 10000k particles.

5 Bonus Exercise

I measured performance for a few different number of iterations and also tried different block sizes for each of those runs, and also did the same runs with both single and double precision and summed up my data in six graphs. My findings were unsurprisingly that single precision was less precise, and also faster than double precision. However I was surprised by just how much faster single precision was than double precision. In test with the most particles the program finished approximately 10 times faster when using single precision compared to double precision. It did cost a bit of accuracy tho, the same run yielded an error of approximately 0.0002% for the double precision and 0.014% for single precision. One surprise when looking at Figure 6b was that the error seemed to grow with the number of iterations. My hypothesis is that the error is growing bigger with the massive amount of computations towards the end when using less precision.

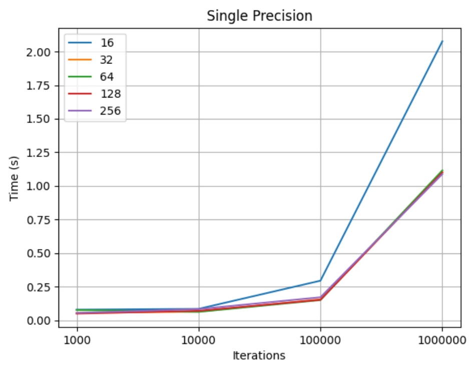


(a) Linear y-axis

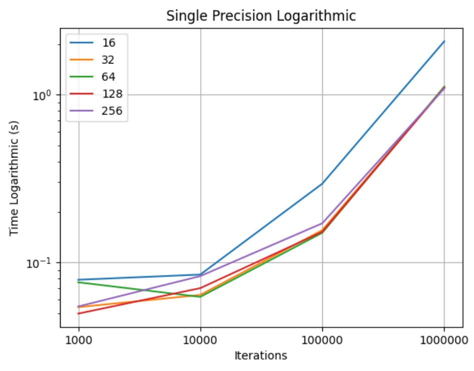


(b) Logarithmic y-axis

Figure 4: Graphs of execution time when using double precision for different block sizes.

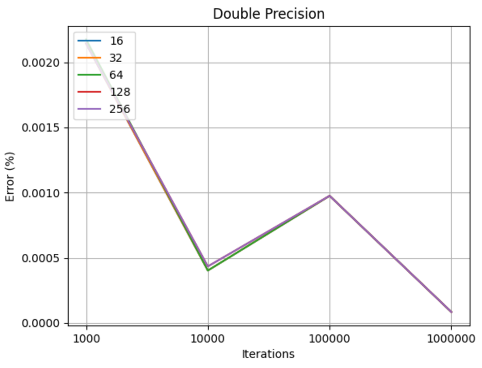


(a) Linear y-axis

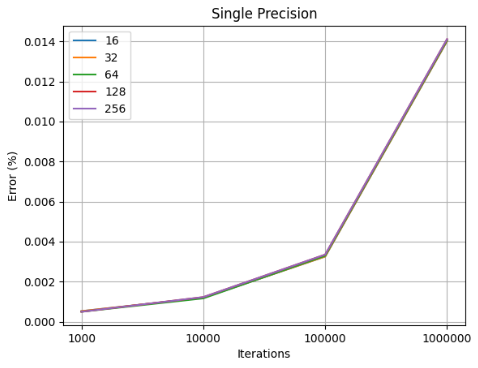


(b) Logarithmic y-axis

Figure 5: Graphs of execution time when using single precision for different block sizes.



(a) Double Precision



(b) Single Precision

Figure 6: Graphs of error using double and single precision with different amounts of iterations and block sizes.