DD2360 - Applied GPU Programming
Homework 2

Kai Fleischman - Gabriel Ballot

November 13, 2020

1 Exercise 1

Explain how the program is compiled and the environment that you used. Explain what are CUDA threads and thread blocks, and how they are related to GPU execution.

We ran this, and all CUDA programs for this assignment on a normal lab machine. Setup of the environment was done as described in "Tutorial: Using CUDA in the laboratory workstations" where we included the directory `/usr/local/cuda/bin` to our path and then compiled this and all CUDA programs with a command of the format `nvcc -arch=sm_50 your_cuda_program.cu -o your_cuda_program.o`. If successful (and our code is not full of nasty errors), this produces a file named `your_cuda_program.o` which an executable that may be run.

CUDA threads are a single running instance of a kernel function on the GPU. This corresponds directly to a thread being run on the GPU. Thread blocks, then, are a group of CUDA threads that can be controlled as a unit (for example using the function `__synchThreads()`). On the GPU, all threads within a block actually all run on the same SM and thus threads within the same block can used "shared memory" which is memory that is close to an SM (and thus much quicker than global memory) and which all threads within a thread block can access.

2 Exercise 2

Explain how you solved the issue when the `ARRAY_SIZE` is not a multiple of the block size. If you implemented timing in the code, vary `ARRAY_SIZE` from small to large, and explain how the execution time changes between GPU and CPU.

Since threads are created in groups known as thread blocks, the number of threads executing the kernel function will always be a multiple of the thread block size (aka threads per block.) Since you cannot created exactly the number of threads needed to solve a problem, but you always have to do a multiple of the block size, there will be some threads that have no work to do. In this case, when a thread calculates its ID, you check that ID against a "work threshold" which is a value that denotes how many threads are needed to complete the problem. If a thread has an ID greater than "work threshold" (in the case of exercise 2, this is an integer passed to the kernel function which denotes the number of elements in the arrays `X` and `Y`) then the thread simply returns from the function and does not attempt to do any work.

The Table (1) shows execution times of the GPU and CPU implementations of SAXPY with different numbers of elements in the `X` and `Y` arrays. Initially, until around 10000 elements, the CPU implementation is quicker than the GPU implementation. This is likely due to the overhead required to launch the GPU rather than the GPU struggling to compute only 10000 elements worth of SAXPY (in fact, you can tell that the runtime of the GPU is only overhead up until this point because the runtime stays about the same from 1 to

Number of Elements	GPU time (seconds)	CPU time (seconds)
10	0.000017	< 0.0000005
100	0.00002	0.000001
1000	0.00002	0.000003
10000	0.00002	0.000025
50000	0.000025	0.00012
100000	0.00003	0.00023
1000000	0.0005	0.0023

Table 1: Time analysis of exercise 2.

Number of particles	CPU time (seconds)
10	0.000744
50	0.003745
100	0.007672
500	0.037154
1000	0.072702
5000	0.3627
10000	0.72393
50000	3.626104
100000	7.240145

Table 2: Time analysis of the execution time on CPU according to the number of particles for exercise 3.

10000 elements.) After this point of about 10000 elements per array, the GPU implementation blows the CPU out of the water, which is what we would expect of the GPU given that SAXPY is easily computable in a parallel manner and can use oversubscribing to calculate different elements with SAXPY while other threads wait to read data from GPU global memory, whereas the CPU is having to serially do calculations, and with this implementation cannot hide latency of a SAXPY operation where slow memory access is required. It would be interesting to see how a multi-threaded implementation of SAXPY for CPU would compare to the GPU...

3 Exercise 3

1. In the table (2) and the figure (1) we can see the linear relationship between the number of particles and the execution time on CPU. This was expected as the complexity is linear in the number of particles.

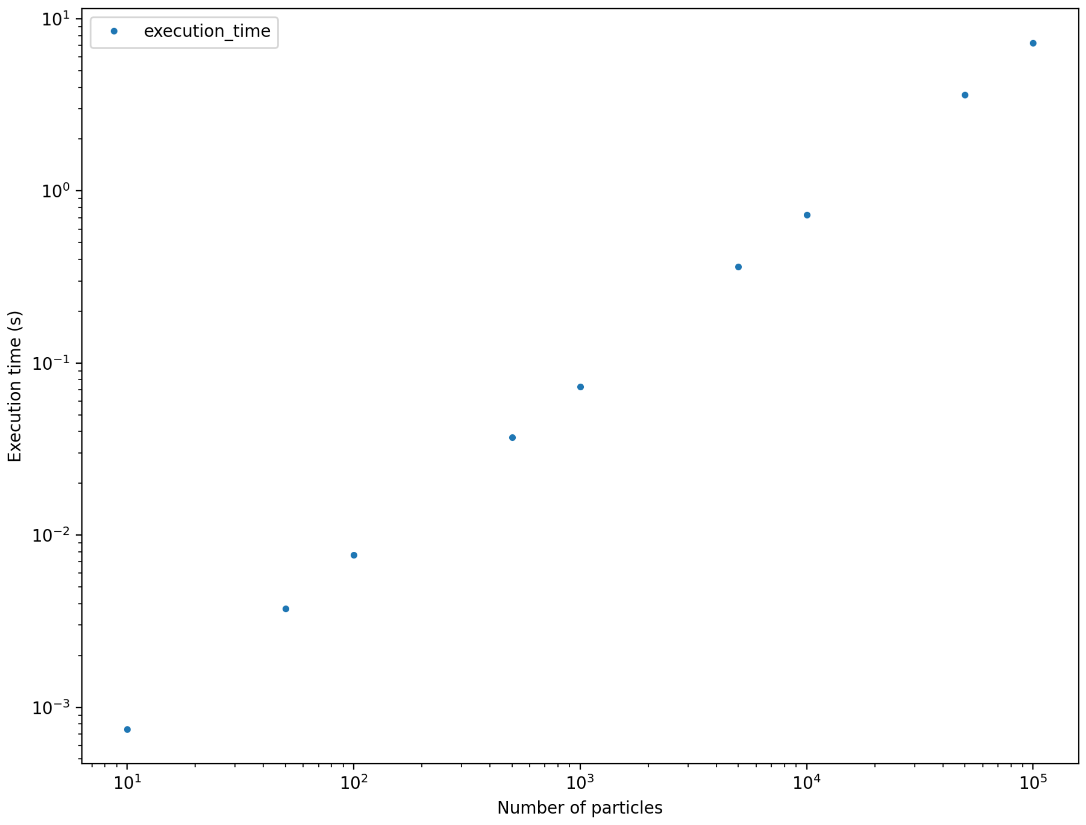

Figure 1: Time analysis of the execution time on CPU according to the number of particles on log log axis.

2. In the Table (3) and the Figure (2) we can see the linear relationship between the number of particles

Number of particles	Execution time (s)				
	16 TPB	32 TPB	64 TPB	128 TPB	256 TPB
10	0.000157	0.000157	0.000157	0.000157	0.000158
50	0.00016	0.000213	0.000202	0.000158	0.000206
100	0.001477	0.000204	0.000199	0.0002	0.000223
500	0.000289	0.00024	0.000237	0.000229	0.000234
1000	0.000454	0.00029	0.00027	0.000268	0.000331
5000	0.001787	0.000969	0.000994	0.000992	0.000969
10000	0.003466	0.001807	0.001835	0.001797	0.00203
50000	0.020116	0.008408	0.008449	0.00838	0.008467
100000	0.037888	0.017797	0.01807	0.01784	0.017937
500000	0.164911	0.082665	0.082692	0.082748	0.082848
1000000	0.329548	0.165118	0.165141	0.16512	0.165523
5000000	1.647648	0.824838	0.825125	0.825176	0.826749
10000000	3.295345	1.649964	1.650199	1.650721	1.653752

Table 3: Time analysis of the GPU according to the block size (TPB) and the number of particles. 10000 itérations.

and the execution time on GPU. This was expected as the complexity is linear in the number of particles as soon as $N_{\text{particles}} = o(TPB)$. However when the number of iteration is low we can see the overhead of transferring the data on GPU. Also we see that the configuration 16 TPB is twice worse than 32 TPB, but 32 TPB and above are equivalent.

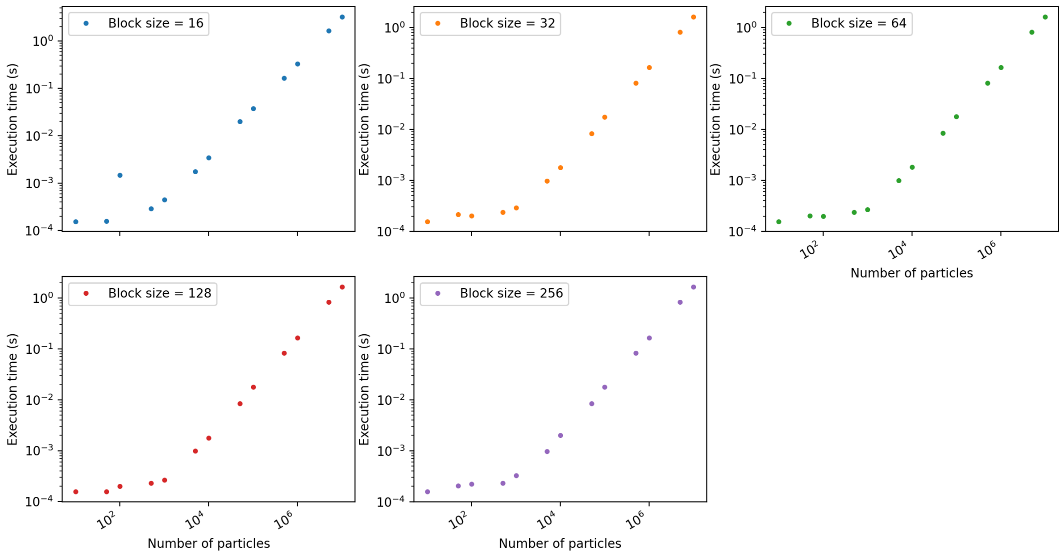

Figure 2: Time analysis of the execution time on GPU according to the number of particles and the block size, on log log axis. 10000 iterations.

3. For the Figure (1) and (2) we used the machine from the lab.

   GPU specification:

   - Model: Nvidia GeForce GTX 745
   - # of SMs: 3
   - Cores per SM: 128
   - Clock Frequency: 1033 MHz
   - Memory Size: 4 Gigabytes

NUM_ITER	GPU time (seconds)
1000000	0.00436
5000000	0.013
10000000	0.024
50000000	0.11
100000000	0.21
500000000	1.02
1000000000	2.055

Table 4: Time analysis of the bonus exercise.

TPB	GPU time (seconds)
16	0.22
32	0.21
64	0.205
128	0.202
256	0.21

Table 5: Time analysis of the bonus exercise with NUM_ITER = 100000000.

After a threshold of around 1000 particles, the execution time is linear in comparison with the number of particles. As soon as the block size is more than 32 TPB, the execution time doesn't change. For 16 TPB it's twice longer.

4. If we have to move the data for after each iteration, we would see the overhead of the transfert while the workload would be very low on the GPU. This is not a good idea except if we have a very large number or particles in which case it could be still worth it.

   Output of `nvprof` on tagner GPU Nvidia K420:

```
==20065== NVPROF is profiling process 20065, command: ./exercise_3.out 10000000 256
Simulating particles on the GPU... 0.000207 seconds
Comparing the output for each implementation... Correct!
==20065== Profiling application: ./exercise_3.out 10000000 256
==20065== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   65.90%  74.874ms         1  74.874ms  74.874ms  74.874ms  [CUDA memcpy DtoH]
                   34.10%  38.743ms         1  38.743ms  38.743ms  38.743ms  [CUDA memcpy HtoD]
      API calls:   50.11%  116.39ms         1  116.39ms  116.39ms  116.39ms  cudaMalloc
                   49.35%  114.62ms         2  57.312ms  38.703ms  75.920ms  cudaMemcpy
                    0.20%  468.62us        96  4.8810us     330ns  222.33us  cuDeviceGetAttribute
                    0.17%  406.00us         1  406.00us  406.00us  406.00us  cudaFree
                    0.07%  168.75us         1  168.75us  168.75us  168.75us  cudaDeviceSynchronize
                    0.06%  134.73us         1  134.73us  134.73us  134.73us  cuDeviceTotalMem
                    0.02%  47.990us         1  47.990us  47.990us  47.990us  cuDeviceGetName
                    0.01%  12.244us         1  12.244us  12.244us  12.244us  cuDeviceGetPCIBusId
                    0.00%  4.0870us         1  4.0870us  4.0870us  4.0870us  cudaLaunchKernel
                    0.00%  2.7760us         3     925ns     430ns  1.7230us  cuDeviceGetCount
                    0.00%  1.9460us         2     973ns     396ns  1.5500us  cuDeviceGet
                    0.00%     606ns         1     606ns     606ns     606ns  cuDeviceGetUuid
```

4 Bonus exercise

In Table (4), we have measured the execution time of the GPU version, varying `NUM_ITER`.

In Table (5), we have measured the execution time, varying the block size in the GPU version from 16, 32, . . . , up to 256 threads per block.

Change the code to single precision and compare the result and performance with the code using double precision. Do you obtain what you expected in terms of accuracy and performance? Motivate your answer. The accuracy and performance did not change. The fact that the accuracy did not change was not surprising as the result was already not very accurate - only a 5 decimals of pi were correct, and so to change the operations from single to double precision will not make a huge difference if any in this regard. However, the fact that the performance did not change was definitely a surprise. I thought there would be some sort of optimization when the operations being used were of a lower precision but I was incorrect in this regard. The GPU executed the Pi estimation program just as well for both double precision and single precision operations.