Assignment II: CUDA Basics

Group 12 - Chao Xiong, Chang Fu

Repository link: https://github.com/keuin19971118/DD2360

Exercise 1 - Hello World!

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 use Tegner's K80 with Cuda 10.0 with the following command (sm_30 means the code is for Kepler GPUs).

```
nvcc -arch=sm_30 exercise_1.cu -o exercise_1
```

All threads launched by a kernel Threads are organized as a grid and these threads in the same grid share the same global memory. Blocks and grids are software concepts. And one block is assigned to one SM and the SM divides these threads into a warp which is a collection of 32 threads as the basic execution unit. And in a kernel, we can use the global index of a thread to assign tasks to it for parallel computation.

Exercise 2 - Performing SAXPY on the GPU

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.

Allocate (ARRAY_SIZE+block_size-1)/block_size space to a 1-d grid, so that the grid_size can be the same as $\lceil \frac{ArraySize}{BlockSize} \rceil$. And when it comes to thread indexing, just assign those out of range threads nothing.

As the ARRAY_SIZE increases, the execution time for CPU SAPX also increases significantly while the execution time for GPU code remains approximately constant.

Exercise 3 - CUDA simulation and GPU Profiling

#Iteration = 100 with Telsa K80 in Tegner.

Particles Block Size	10k	1M	10M
16	0.110318	0.133916	0.503438
32	0.113341	0.115890	0.351539
64	0.112436	0.108855	0.273953
128	0.115863	0.107085	0.251878
256	0.111213	0.107330	0.251356
CPU	0.031320	2.302972	22.706387

Additionally with 1k iterations, it took cpu 4min while GPU only spent 0.8s with 256 block size and 10M particles.

And the block_size 128 seems optimal as 256 requires more resources. With the increasing number of particles, the GPU code gains huge advantage in terms of performance because in the kernel particles can be updated in parallel by each thread while in the CPU code, we have to traverse each particle and update them one by one.

The profiling results() are as followed.

```
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   85.96%  726.92ms         1  726.92ms  726.92ms  726.92ms  launch_mover(Particle*, int, int)
                    8.72%  73.743ms         1  73.743ms  73.743ms  73.743ms  [CUDA memcpy DtoH]
                    5.32%  44.969ms         1  44.969ms  44.969ms  44.969ms  [CUDA memcpy HtoD]
      API calls:   55.25%  1.04843s         1  1.04843s  1.04843s  1.04843s  cudaMalloc
                   38.31%  726.93ms         1  726.93ms  726.93ms  726.93ms  cudaDeviceSynchronize
                    6.32%  119.85ms         2  59.927ms  45.662ms  74.192ms  cudaMemcpy
                    0.06%  1.0783ms         1  1.0783ms  1.0783ms  1.0783ms  cuDeviceTotalMem
                    0.03%  524.74us        96  5.4660us     329ns  184.51us  cuDeviceGetAttribute
                    0.03%  494.61us         1  494.61us  494.61us  494.61us  cudaFree
                    0.01%  271.36us         1  271.36us  271.36us  271.36us  cudaLaunchKernel
                    0.00%  59.061us         1  59.061us  59.061us  59.061us  cuDeviceGetName
                    0.00%  15.867us         1  15.867us  15.867us  15.867us  cuDeviceGetPCIBusId
                    0.00%  3.4410us         3  1.1470us     347ns  1.8630us  cuDeviceGetCount
                    0.00%  1.9820us         2     991ns     400ns  1.5820us  cuDeviceGet
                    0.00%     606ns         1     606ns     606ns     606ns  cuDeviceGetUuid
```

From these we can see, the GPU is much faster compared to the CPU. Around 40% of total execution time was spent on data transfer.

And if the kernel is dependent on some CPU execution, the performance gap between the cpu version and the so-called gpu version may not be as huge as the independent one. Because, the bottleneck now should be the CPU part and threads on GPU will be blocked for synchronization with CPU, which is quite expensive in highly parallelized GPU and massive data transfers will also lead to slow executions.

Bonus Exercise - Calculate PI with CUDA

Measure the execution time of the GPU version, varying NUM_ITER.

TRIALS_PER_THREAD*	100 000	1 000 000	10 000 000
Execution Time / s	0.250815	1.383248	12.236707
Estimated Pi	3.141996	3.141805	3.141634

** In this experiment, we use one gpu block and 256 threads per block. Thus, with TRIALS_PER_THREAD set as 100 000, NUM_ITER is implicitly set as 25 600 000.*

Generally when we increase NUM_ITER, the execution time increases as well, and the more trials we run, the more precise pi we get.

Measure the execution time, varying the block size in the GPU version from 16, 32, …, up to 256 threads per block.

Block Size	16	32	64	128	256
Execution Time / s	0.761012	0.773458	0.775736	0.881581	1.373064
Estimated Pi	3.141974	3.141716	3.141702	3.141748	3.141805

** In this experiment, we set TRIALS_PER_THREAD as 1 000 000.*

Generally the execution time increases when the block size increases. Interestingly, the most precise pi we get is when the block size is 64, and the reason behind this might be poor randomness when the block size is large.

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.

TRIALS_PER_THREAD	100 000	1 000 000	10 000 000
Execution Time / s	0.141176	0.679934	5.190466
Estimated Pi	3.141997	3.141806	3.141635

** In this experiment, we use single precision instead of double precision.*

The program performs perfectly as well when we change it to single precision, which also shortens the execution time. The reason behind this is that we use integers to count the times that the coin falls in the circle, and no float calculation is involved in gpu computation.