DD2360 Assignment III

CUDA Basics

Code link: https://github.com/tonycyz/DD2360/hw_3

Yuze Cui

yuzec@kth.se

Exercise 1:

1. Explain how the program is compiled and run.
   The code is compiled and run in Google Colab. When compiling the cuda code, the -arch parameter in nvcc should be modified to sm_75 to fit the Tesla T4 GPU.
   To compile the code: !nvcc -arch=sm_75 vectorAdd.cu -o vectorAdd
   To run the output file: ./vectorAdd *inputLength(modified when running)*
   To run the file using nv-nsight: !/usr/local/cuda-11/bin/nv-nsight-cu-cli ./vectorAdd *inputLength*

2. For a vector length of N:
   (1) How many floating operations are being performed in your vector add kernel?
   In my kernel, N times of floating operations are performed.
   (2) How many global memory reads are being performed by your kernel?
   2*N timed of global memory reads

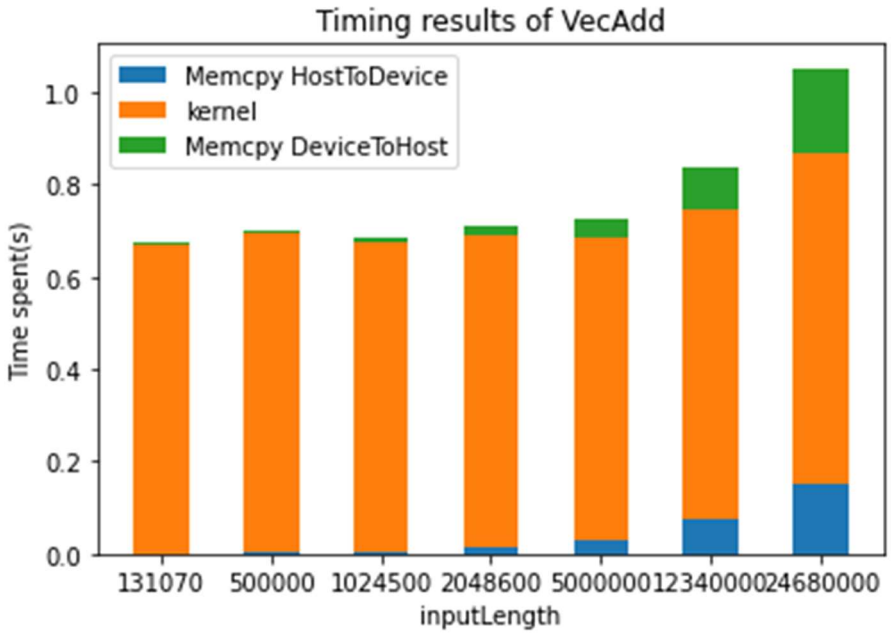
3. For a vector length of 1024:
   (1) Explain how many CUDA threads and thread blocks you used.
   I set Threads per Block to 256. So there are 1024/4 blocks I used. The number of threads is the MAX_THREAD, which is 1024.
   (2) Profile your program with Nvidia Nsight. What Achieved Occupancy did you get?
   It achieved 23.45% occupancy.

4. Now increase the vector length to 131070:
   (1) Did your program still work? If not, what changes did you make?
   It still works.
   (2) Explain how many CUDA threads and thread blocks you used.
   The program use 1024 threads because the inputLength exceeds the maximum number of threads. (131070+256-1/256)=512 blocks will be used.

(3) It achieved 76.85% occupancy.

5. Further increase the vector length (try 6-10 different vector length), plot a stacked bar chart showing the breakdown of time including (1) data copy from host to device (2) the CUDA kernel (3) data copy from device to host. For this, you will need to add simple CPU timers to your code regions.
   After adding the timing codes, I choose the inputLength of 131070 500000 1024500 2048600 5000000 12340000 24680000 and collect the data to form the chart.

	Size	Datacpy HtoD	Kernel	Datacpy HtoD
1	131070	0.001086	0.670755	0.001112
2	500000	0.003311	0.693620	0.003785
3	1024500	0.006397	0.668674	0.008055
4	2048600	0.012492	0.679471	0.016489
5	5000000	0.030461	0.653734	0.041028
6	12340000	0.076117	0.669683	0.092989
7	24680000	0.151735	0.718206	0.182792



Exercise 2

1. Name three applications domains of matrix multiplication.
   Scientific computing, pattern recognition and counting the paths through a graph.

2. How many floating operations are being performed in your matrix multiply kernel?
   In the kernel, there are add and multiply operations when multiply the matrix, *2*numAColumns*numBColumns*numARows*

3. How many global memory reads are being performed by your kernel?

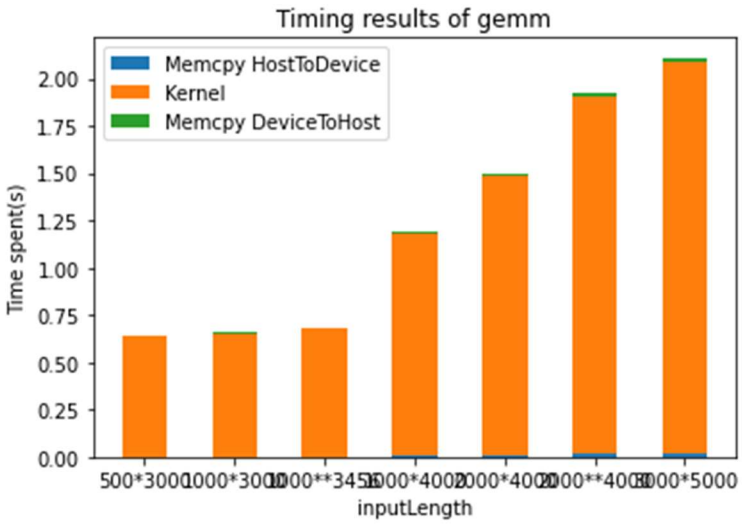
In the kernel, there are *numAColumns*numBColumns*numARows+ numBRows*numBColumns*numARows= 2*numAColumns*numBColumns*numARows* global memory reads.

4. For a matrix A of (128x128) and B of (128x128):
   (1) Explain how many CUDA threads and thread blocks you used.
   Since I set the TPB to 32. There will be 32(in X direction) *32(in Y direction) threads per block. So 4*4=16 blocks will be used.
   (2) Profile your program with Nvidia Nsight. What Achieved Occupancy did you get?
   Achieved Occupancy is 96.28%

5. For a matrix A of (511x1023) and B of (1023x4094):
   (1) Did your program still work? If not, what changes did you make?
   Yes. It still works.
   (2) Explain how many CUDA threads and thread blocks you used.
   Since I still set the TPB to 32. There will be 32(in X direction) *32(in Y direction) threads per block. And (511+31/32) *(4094/32)=16*128=2048 blocks are used.
   (3) Profile your program with Nvidia Nsight. What Achieved Occupancy do you get now?
   Achieved Occupancy is 98.57%

6. Further increase the size of matrix A and B, plot a stacked bar chart showing the breakdown of time including (1) data copy from host to device (2) the CUDA kernel (3) data copy from device to host. For this, you will need to add simple CPU timers to your code regions. Explain what you observe.
   Here I increase the size and obtain the data below:

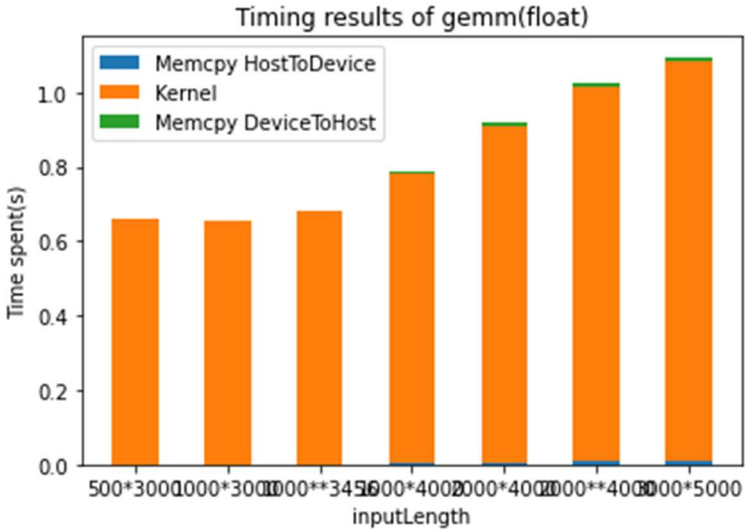
	Size	Datacpy HtoD	Kernel	Datacpy HtoD
1	(100*234)*(234*300)	0.000215	0.643000	0.000193
2	(500*234)*(234*300)	0.000431	0.654666	0.001028
3	(100*1234)*(1234*1000)	0.002491	0.679696	0.000637
4	(500*1234)*(1234*3000)	0.007768	1.175394	0.008021
5	(1000*1500)*(1500*3000)	0.010462	1.472597	0.016018
6	(1000*2001)*(2001*3456)	0.015366	1.887082	0.018674
7	(1000*2000)*(2000*4000)	0.017723	2.072227	0.020620



From the chart, it can be found out the kernel takes more and more computation cost. All parts of computation increases when matrix is larger but cuda kernel always takes the most time.

7. Now, change DataType from double to float, re-plot a stacked bar chart showing the time breakdown. Explain what you observe.

	Size	Datacpy HtoD	Kernel	Datacpy HtoD
1	(100*234)*(234*300)	0.000138	0.660578	0.000133
2	(500*234)*(234*300)	0.000289	0.653341	0.000481
3	(100*1234)*(1234*1000)	0.001273	0.678294	0.000319
4	(500*1234)*(1234*3000)	0.003855	0.777841	0.004179
5	(1000*1500)*(1500*3000)	0.005632	0.905066	0.008309
6	(1000*2001)*(2001*3456)	0.007999	1.006080	0.009066
7	(1000*2000)*(2000*4000)	0.008855	1.076331	0.010695



It can be found out that the portion each operation takes is likely the same with the double type one but float computation is faster.

Exercise 3

1. Describe all optimizations you tried regardless of whether you committed to them or abandoned them and whether they improved or hurt performance.
   In this exercise, I used atomic function: atomicAdd() and shared memory to solve the problem.
   atomicAdd() is essential to use when using global memory. Since the sequencing is controllable in this application, atomic access must be used to avoid race condition.
   Shared memory improves the performance because the memory local to SM will be faster compared with global memory.

2. Which optimizations you chose in the end and why?
   I chose global memory because it is faster.

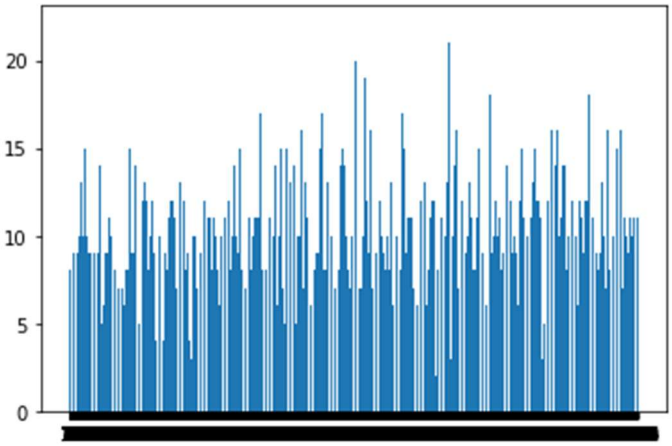
3. How many global memory reads are being performed by your kernel? Explain
   In the histogram kernel, the size is *sizeof(unsigned int)*num_elements*, in order to read in all inputs.
   In the convert kernel, the size is *sizeof(unsigned int)*num_bins*, in order to remove the exceeding number of bin outputs.

4. How many atomic operations are being performed by your kernel? Explain
   There are *num_bins+num_elements* atomic operations performed in my kernel. As in the code, the atomicAdd is used in read the input and get back results from each block.
   In the convert kernel, there is no atomic operations.

5. How much shared memory is used in your code? Explain
   The size will be *sizeof(unsigned int)*num_bins*number of blocks*, because each block will use all the of bins to do computation.

6. How would the value distribution of the input array affect the contention among threads? For instance, what contentions would you expect if every element in the array has the same value?
   If the elements become more concentrated, the contention will be large because it will be like a serial operation especially when all elements are the same.

7. Plot a histogram generated by your code and specify your input length, thread block and grid.
   InputLength=40960; thread block=32 threads per block; grid:1280



8. For an input array of 1024 elements, profile with Nvidia Nsight and report Shared Memory Configuration Size and Achieved Occupancy. Did Nvsight report any potential performance issues?
   To find the result, I adjusted the number of bins of 1024.
   Shared Memory Configuration Size: 32.77 KB
   Achieved Occupancy: 3.12%
   Yes. The kernel grid is too small(resulting in only 0.1 full waves across all SMs) and the launch only execute 32 blocks which is less than the GPU's 40 processors. This will underutilize the hardware.

Exercise 4

1. Describe the environment you used, what changes you made to the Makefile, and how you ran the simulation.
   I used Google Colab environment with an NVIDIA Tesla T4 GPU. TO compile the codes, I adjust the arch parameter to sm_75 to fit the GPU.
   The commands to run the simulation:
   !make
   !./bin/sputniPIC.out inputfiles/GEM_2D.inp

2. Describe your design of the GPU implementation of mover_PC() briefly.
   To modified the mover_PC function, I removed the cycles to analyze each particles because the GPU will create threads to run the each analysis. The analyzing functions are moved to GPU kernel where the number of particles are removed by thread id. In the mover_PC_gpu function, it will perform memory copy and launching kernel to finish the computation.

3. Compare the output of both CPU and GPU implementation to guarantee that your GPU implementations produce correct answers
   Yes. It produced correct answers.

4. Compare the execution time of your GPU implementation with its CPU version.

The execution time of GPU is much improved than CPU version. Shown as below:

```
****************************************
   Tot. Simulation Time (s) = 58.7061
   Mover Time / Cycle   (s) = 3.19148
   Interp. Time / Cycle (s) = 2.24732
****************************************
```

```
****************************************
   Tot. Simulation Time (s) = 26.3247
   Mover Time / Cycle   (s) = 0.0919365
   Interp. Time / Cycle (s) = 2.1557
****************************************
```