DD2360/FDD3360 HT22 Applied GPU Programming

Assignment 3: CUDA Basics

Weichun Li weichun@kth.se
https://github.com/WCL-26/DD2360HT22

January 4, 2023

1 Your first CUDA program and GPU performance metrics

1.1 Explain how the program is compiled and run.

The process of compiling and running the program is shown in file hw3_ex1.ipynb

1.2 For a vector length of N:

1.2.1 How many floating operations are being performed in your vector add kernel?

N floating point operations.

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

2N global memory reads.

1.3 For a vector length of 1024:

1.3.1 Explain how many CUDA threads and thread blocks you used.

The block size is 128. And there are 1024 threads. So there are 8 blocks.

1.3.2 Profile your program with Nvidia Nsight. What Achieved Occupancy did you get?

12.07%

1.4 Now increase the vector length to 131070:

1.4.1 Did your program still work? If not, what changes did you make?

Yes, it still works.

1.4.2 Explain how many CUDA threads and thread blocks you used.

The block size is 128. And there are 131,072 threads. So there are 1024 blocks.

1.4.3 Profile your program with Nvidia Nsight. What Achieved Occupancy do you get now?

78.14%

1.5 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.

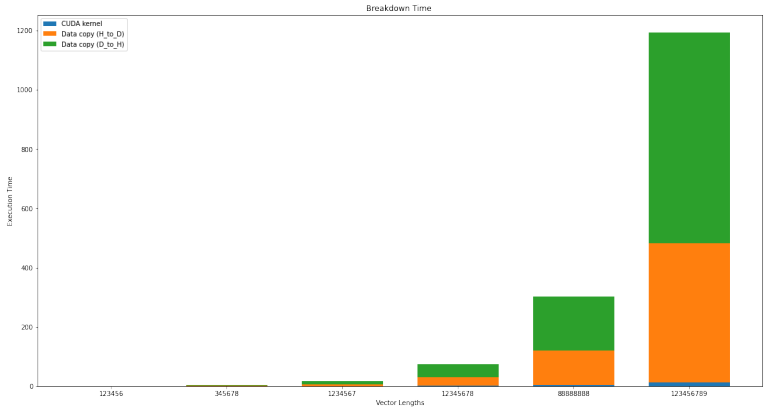

Figure 1: ex1

2 2D Dense Matrix Multiplication

2.1 Name three applications domains of matrix multiplication.

Machine learning, Computer vision, Scientific computing

2.2 How many floating operations are being performed in your matrix multiply kernel?

2*numARows*numBColumns*numAColumns

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

3*numARows*numBColumns*numAColumns

2.4 For a matrix A of (128x128) and B of (128x128):

2.4.1 Explain how many CUDA threads and thread blocks you used.

The block size is 64. And there are 16384 threads. So there are 256 blocks.

2.4.2 Profile your program with Nvidia Nsight. What Achieved Occupancy did you get?

24.44%

2.5 For a matrix A of (511x1023) and B of (1023x4094):

2.5.1 Did your program still work? If not, what changes did you make?

Yes, it still works.

2.5.2 Explain how many CUDA threads and thread blocks you used.

The block size is 1024. And there are 2097152 threads. So there are 2048 blocks.

2.5.3 Profile your program with Nvidia Nsight. What Achieved Occupancy do you get now?

88.24%

2.6 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.

It can be observed that the most of the time is taken by CUDA kernel.And the execution time of the CUDA kernel increase with increasing matrix sizes.

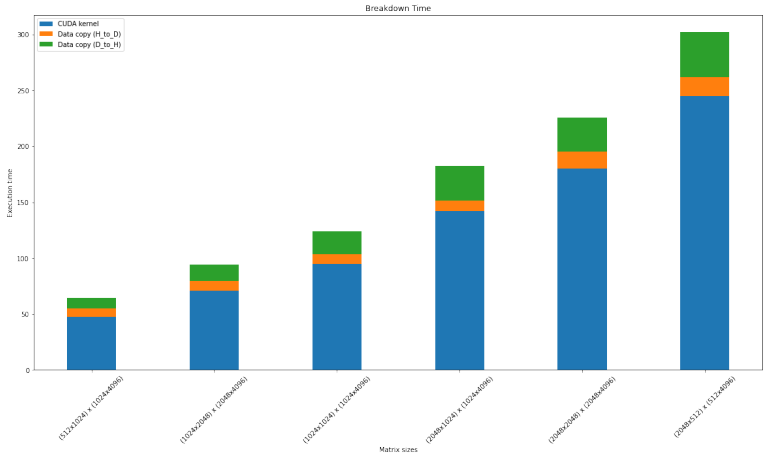

Figure 2: ex2-1

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

It can be observed that using float is much faster than double. That is because float only has half the size of double.

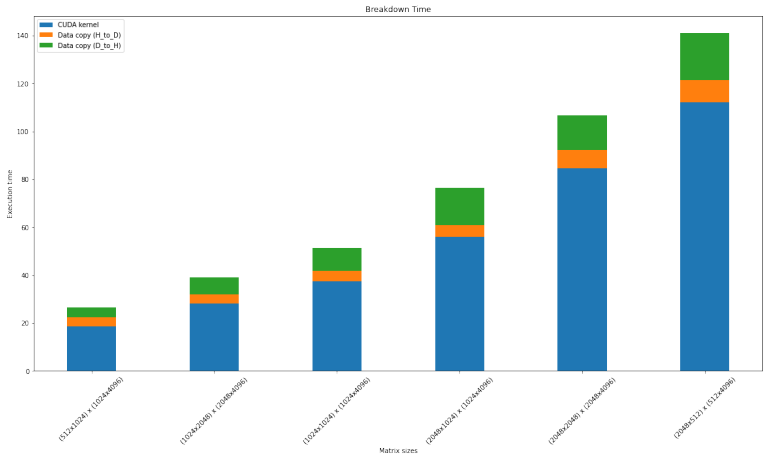

Figure 3: ex2-2

3 Histogram and Atomics

3.1 Describe all optimizations you tried regardless of whether you committed to them or abandoned them and whether they improved or hurt performance.

I have tried two methods. Use atomic operations in global memory and shared memory, respectively.
In global memory, make the number of threads equal to the number of elements in the input array. Then

each thread is responsible for increasing the bin count. Also the atomicAdd() operation must be used to avoid invalid sums.
In shared memory, each thread block has its own histogram. and the results of the histogram produced by each thread block are added together to produce the final histogram result. Clearly the shared memory approach is better suited to the case of a large number of inputs.

3.2 Which optimizations you chose in the end and why?

I chose to use atomic operations in shared memory. Since I think the input will have a big scale.

3.3 How many global memory reads are being performed by your kernel? Explain

histogram_kernel: num_bins+num_elements
convert_kernel: num_bins

3.4 How many atomic operations are being performed by your kernel? Explain

histogram_kernel: num_bins
convert_kernel: none

3.5 How much shared memory is used in your code? Explain

num_blocks*num_bins*4 byte

3.6 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 every element int the array has the same value then that's the max contention.

3.7 Plot a histogram generated by your code and specify your input length, thread block and grid.

Input length: 131072
Thread block: 64 threads per block
Grid: 2048 blocks

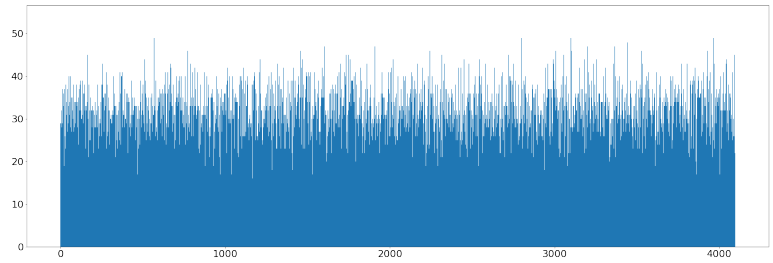

Figure 4: ex3

3.8 For a 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?

Shared Memory Configuration Size:32.77 kbyte
Achieved Occupancy:9.77%
Yes, there is a warning that This kernel grid is too small to fill the available resources on this device, resulting in only 0.1 full waves across all SMs. Look at Launch Statistics for more details.

4 A Particle Simulation Application

4.1 Describe the environment you used, what changes you made to the Makefile, and how you ran the simulation.

Enviroment: Google Colab
Change in makefile: sm_30 to sm_75
The simulation is shown in file hw3_ex4.ipynb.

4.2 Describe your design of the GPU implementation of mover$_PC()briefly.$

I mostly followed the steps in earlier exercises, the steps of serial computing in cpu are translated into parallel computing on different threads in a gpu. The detailed steps including: allocate device memory, copy from host memory to device memory, initialize thread block and kernel grid dimensions, invoke CUDA kernel, copy results from GPU to CPU, free device memory.
Specifically, mover_PC() is very similar to the corresponding mover_PC_GPU() code, except that mover_PC_GPU() lacks a loop over particles and the result must be copied to the host.

4.3 Compare the output of both CPU and GPU implementation to guarantee that your GPU implementations produce correct answers.

Yes, GPU implemnetations produce correct answers.

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

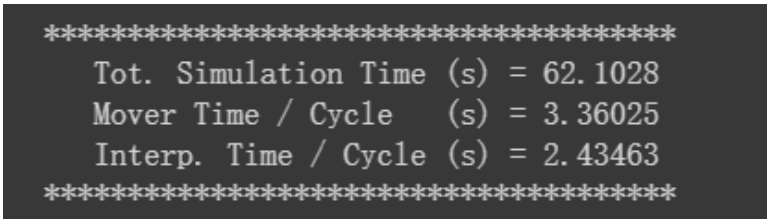

Figure 5: CPU Version

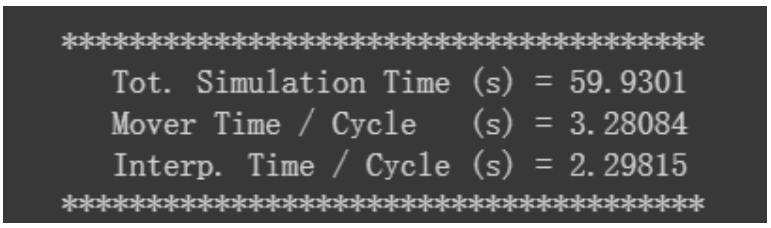

Figure 6: GPU Version