Assignment III: GPU CUDA Basics

Link to GitHub repository: https://github.com/ChrisWe99/DD2360HT22

Exercise 1 – Your first CUDA program and GPU performance metrics

1. **Explain how the program is compiled and run**
   - Environment:
     Google Colab
   - How to compile:
     ```
     !nvcc -arch=sm_75 ./lab3_ex1.cu -o lab3_ex1
     ```
   - How to run:
     ```
     !./lab3_ex1 <vector length>
     ```
   - How to profile with Nvidia Nsight:
     ```
     !/usr/local/cuda-11/bin/nv-nsight-cu-cli ./lab3_ex1 <vector length>
     ```

2. **For a vector of length N**

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

N operations are performed. One for each element pair of the vectors that are added

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

2N global memory reads are being performed by the kernel. Each summand of the addition is read from the memory. (And then the result is written in the memory).

3. For a vector length of 1024:

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

According to Nvidia Nsight, 1024 threads and 8 blocks have been used (128 threads per block as defined in the code).
However, there was a warning which states that the multiprocessors are underutilized.

```
Section: Launch Statistics
-------------------------------------------------------------------------- --------------- ------------------------------
Block Size                                                                                                            128
Function Cache Configuration                                                                    cudaFuncCachePreferNone
Grid Size                                                                                                               8
Registers Per Thread                                                     register/thread                             16
Shared Memory Configuration Size                                                   Kbyte                          32.77
Driver Shared Memory Per Block                                               byte/block                              0
Dynamic Shared Memory Per Block                                              byte/block                              0
Static Shared Memory Per Block                                               byte/block                              0
Threads                                                                          thread                          1,024
Waves Per SM                                                                                                         0.03
-------------------------------------------------------------------------- --------------- ------------------------------
WRN   The grid for this launch is configured to execute only 8 blocks, which is less than the GPU's 40
      multiprocessors. This can underutilize some multiprocessors. If you do not intend to execute this kernel
      concurrently with other workloads, consider reducing the block size to have at least one block per
      multiprocessor or increase the size of the grid to fully utilize the available hardware resources.
```

Figure 0-1: Exercise 1 - part 3.1 number of blocks and threads

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

I've got an achieved occupancy of 12.13% according to Nvidia Nsight.

4. Now increase the vector length to 131070

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

Yes, my program still worked. So I didn't have to make any changes.

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

According to Nvidia Nsight, I used 131072 threads and 128 threads per block which results in 1024 blocks.
Since one thread per addition (131070) is required, two threads are "too much" or not really required.

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

I've got an Achieved Occupancy of 78.28%.

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.

For CPU timing I've used the timeval struct and these tips.

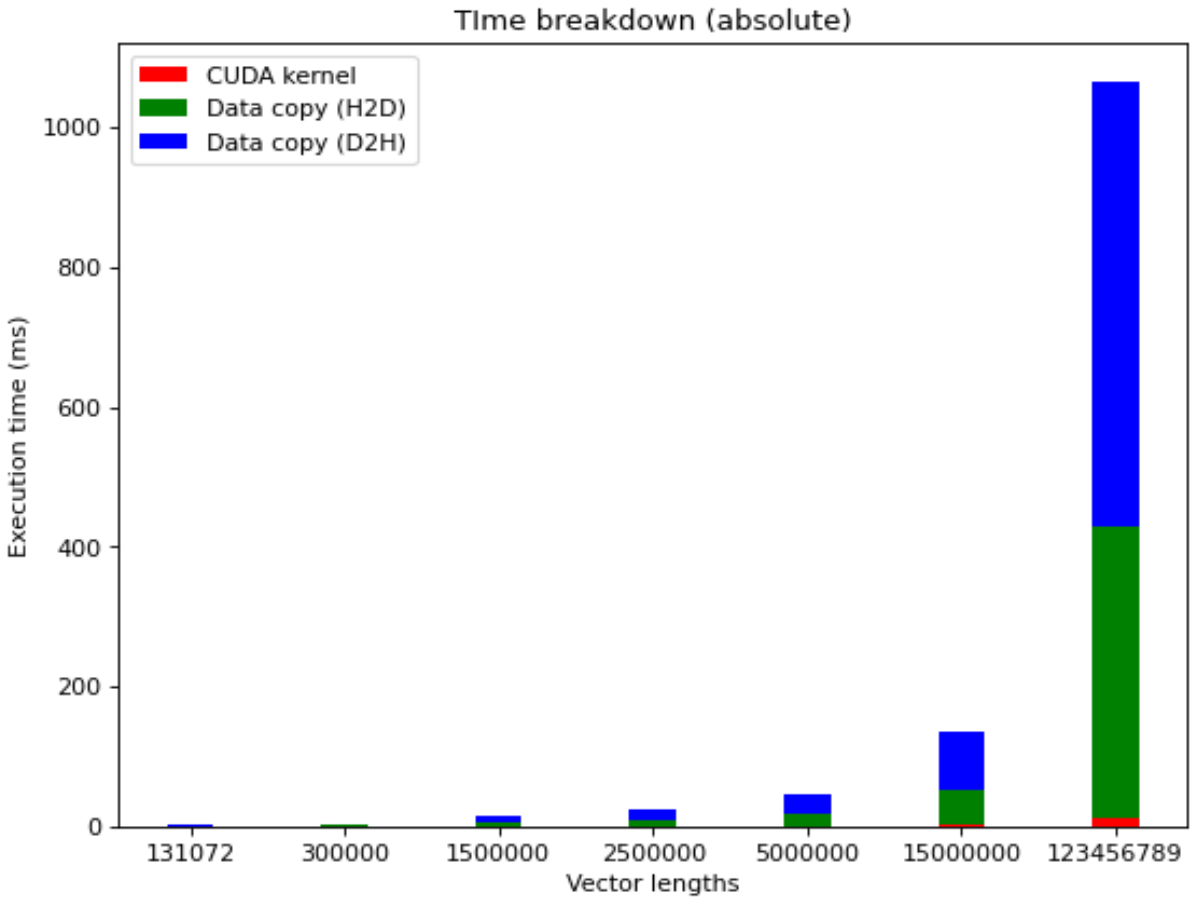

Figure 0-2: Exercise 1 - part 5. – absolute execution time measurements

Since the absolute time is difficult to show properly in one diagram (due to high difference of vector lengths), I also calculated the relative time.

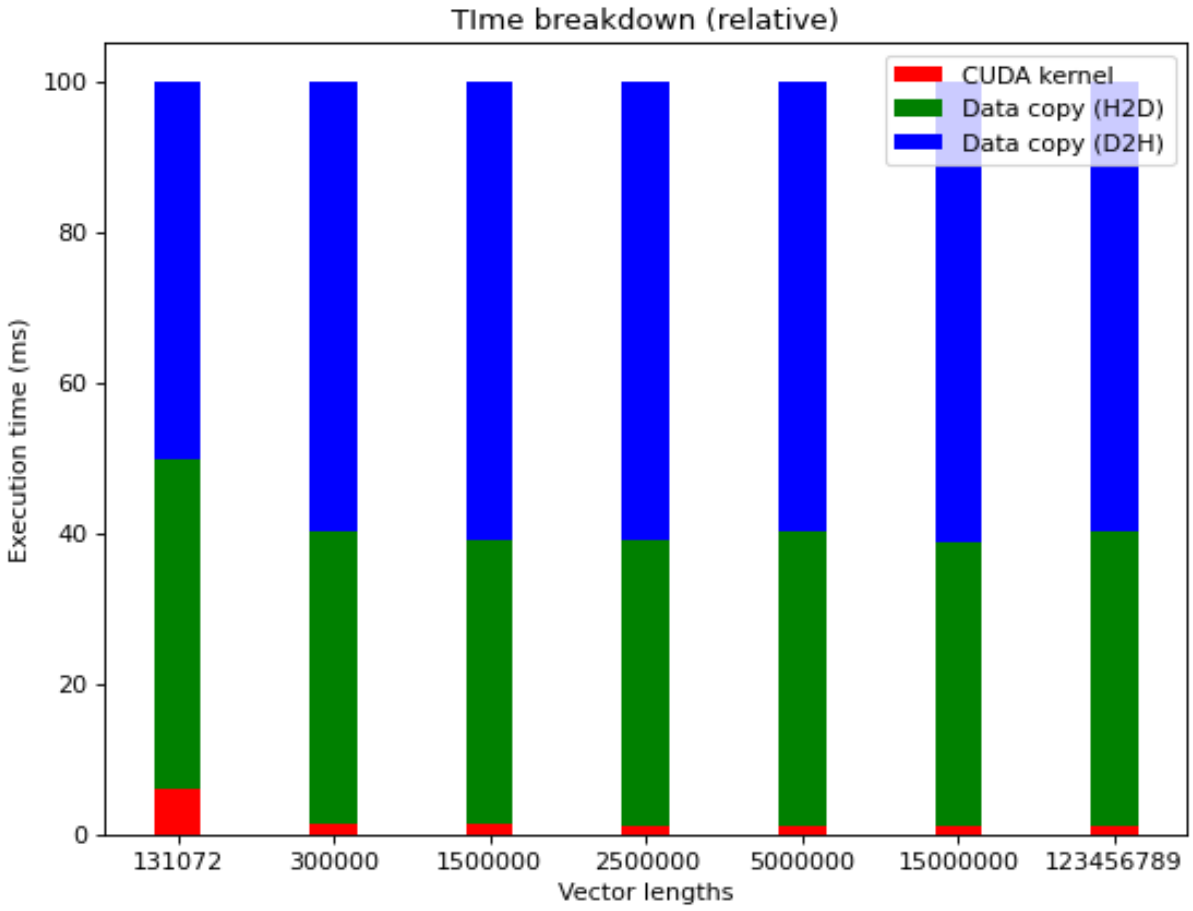

Figure 0-3: Exercise 1 - part 5. – relative execution time measurements

Exercise 2 – 2D Dense Matrix Multiplication

1. **Name three application domains of matrix multiplication**

Computer Vision, Reinforcement Learning, Graphics (simulations/games, …)

2. **How many floating operations are being performed in your matrix multiply kernel?**

The number of floating operations (FLOPs) can be calculated with the following formula:

For two matrices A (N x P) and B (P x M), the number of FLOPs is equal to:

N*M*(2*P-1)

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

For two matrices A (N x P) and B (P x M), the number of global memory reads is equal to:

N*M*P*3

I figured this out by just thinking of examples (e.g. two 2x2 matrices multiplications, …). It is required to read each element from each input matrix once per required multiplication. And then it is necessary to read the multiplication results again to sum them up. Here, it is assumed that the intermediate result is also stored in the global memory.

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

4.1. Explain how many CUDA threads and blocks you used

According to the profiling 16384 threads have been used. This is caused by the fact that 128 * 128 threads (one for each element of the output matrix) are required.

In my code, I initialized the block dimensions to 256 (16*16) threads per block. Consequently, I used 64 blocks.

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

My achieved occupancy is 42.78%.

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

5.1. Did you program still work? If not, what changes did you make?

Yes, it still worked and hence no changes were required.

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

As for the previous task (128x128), I used 256 threads per block (16*16). Since 2,097,152 threads are required, I have used a grid size of 8192 thread blocks.

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

The achieved occupancy is a lot higher at 99.19%.

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.

Since I've experimented with nvprof in my final project, I know that the required timing information can also be determined with this tool. Consequently, I've used nvprof for the timing:

```
!nvprof ./lab3_ex2 <numARows> <numACols> <numBCols>
```

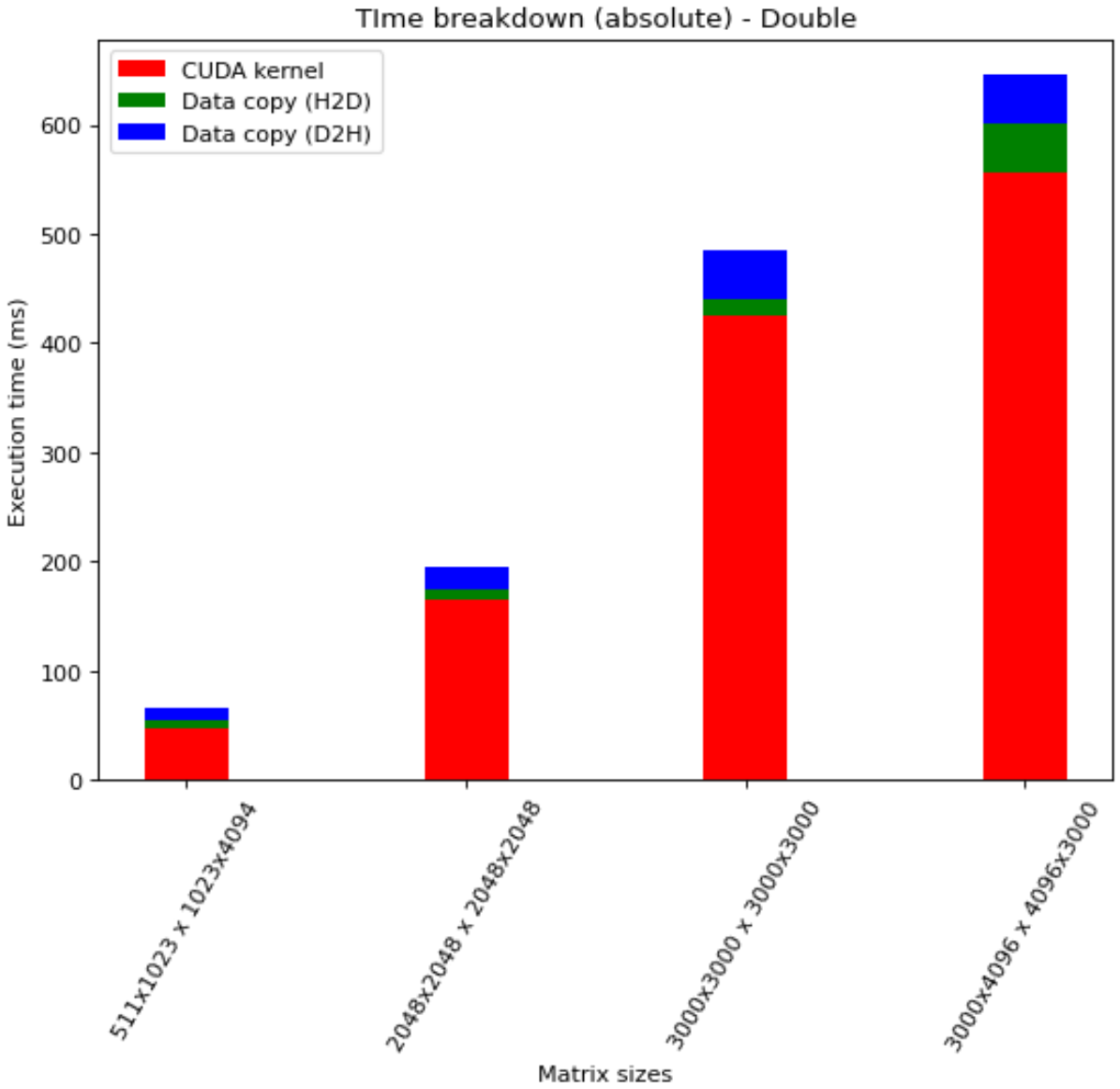

Figure 0-1: Exercise 2 - part 6. – Absolute time measurements – Datatype Double

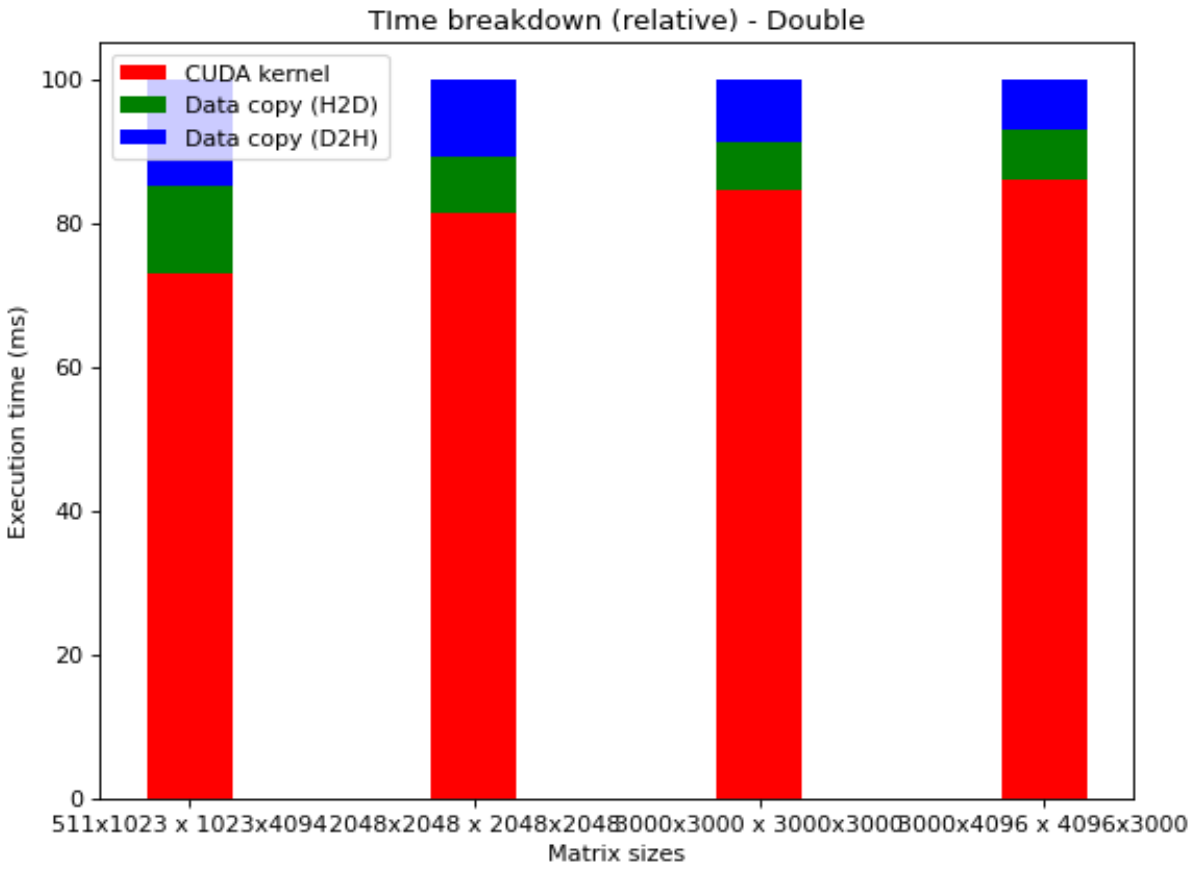

Figure 0-2: Exercise 2 - part 6. – Relative time measurements - Datatype Double

As for the vector task, the CUDA kernel requires most of the time. Moreover, for the relative time breakdown, it can be observed that the relative time of the CUDA Kernel increases with increasing matrix sizes. The reason for that is that the number of operations for the CUDA Kernel increases faster (see tasks 2 and 3) than the required operations for the data copy H2D and D2H (just the number of all elements = numARows * numACols + numBRows * numBCols).

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

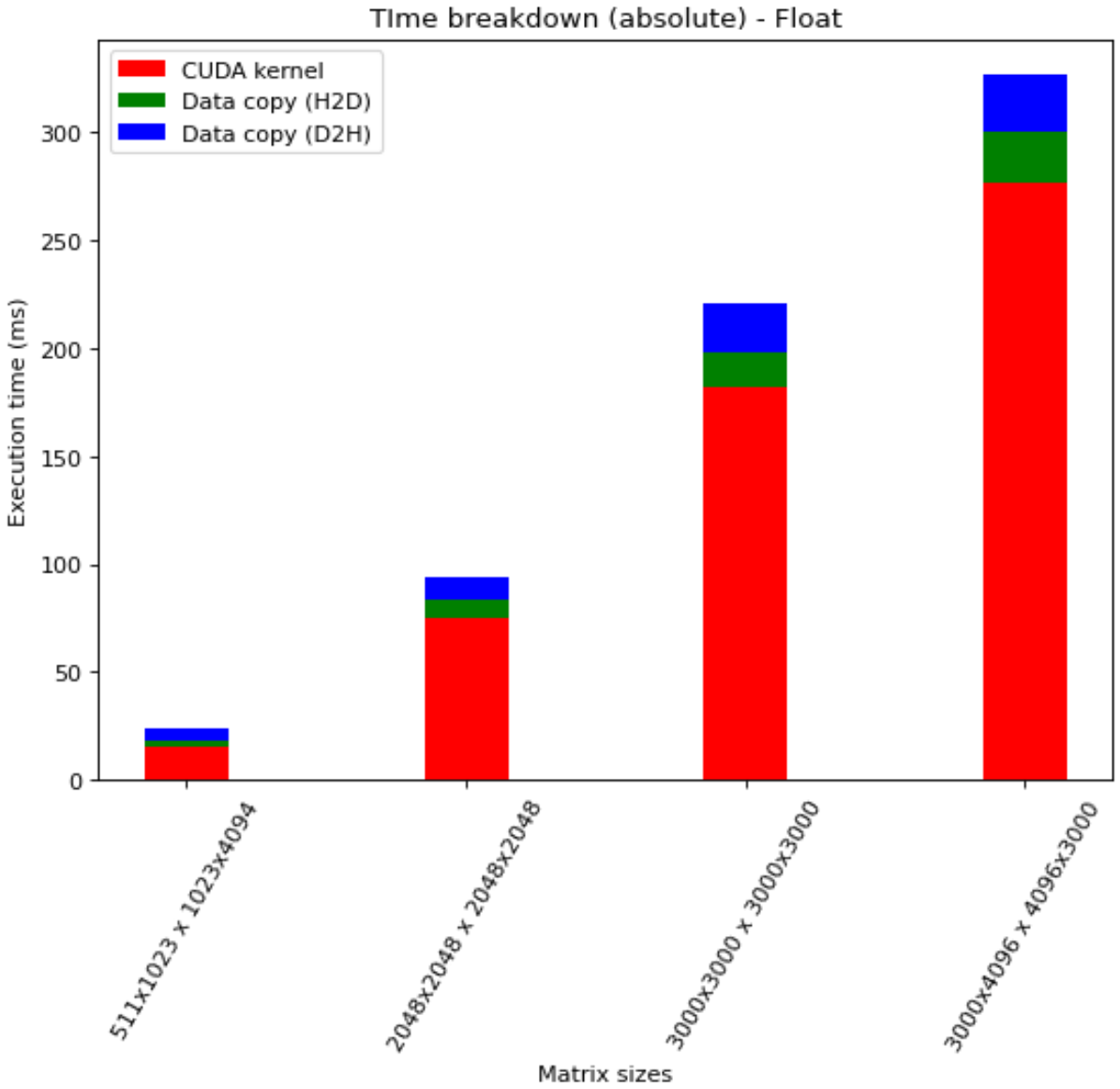

Figure 0-3: Exercise 2 - part 7. – Absolute time measurements - Datatype Float

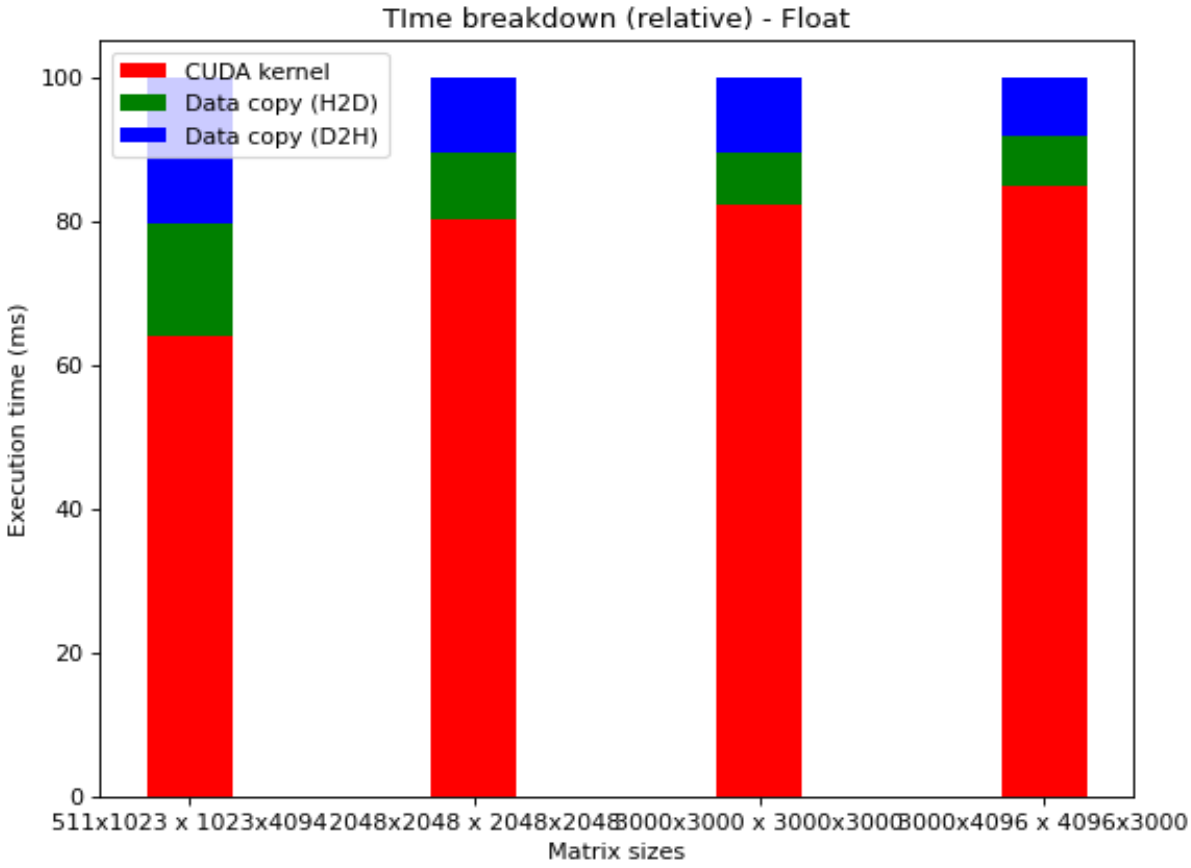

Figure 0-4: Exercise 2 - part 7. – Relative time measurements - Datatype Float

From a relative time-perspective, there is no relevant difference. However, the total time is almost smaller by factor 2 (~300ms vs ~600ms for the largest matrices). This is probably caused by the fact that the datatype float has only half the size of the datatype double.

Exercise 3 – Histogram and Atomics

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

I thought of 3 different optimizations and implemented 2 of them:

1.1. Sorting the input array and then using one of the subsequent other approaches

This was an additional idea which could improve the performance of the two approaches that are presented afterwards. I thought of using a sorting algorithm that can be highly parallelized. And then, the summation for the histogram could have been potentially faster. Since this would have required much more coding and would be very harmful for many cases, I decided to only evaluate it by thinking about it:
I'm quite convinced that the sorting before creating the histogram may only help for very large input problems in which also the number of different values is relatively small compared to the number of total elements.
Consequently, this approach would really hurt all other input problems (e.g., small input problems; input problems with almost no duplicated values; …)

1.2. Straight forward implementation: Using one thread per input element

Since it was the easiest implementation, I solved the task by just using one thread per input element. I did this by just launching the kernel and performing atomic operations for all threads until the thread ID is larger than the number of elements.

1.3. Splitting up the creation of histograms by using one thread block for one histogram and then combining all created histograms

Inspired by the lecture "CUDA Memories, shared and atomics", I decided to try to adapt the task so that I can use shared memory which is faster than global memory. In order to achieve that, I decided to split up the creation of the histogram. Consequently, temporary histograms are created in the different thread blocks. After the creation is finished, all of them will add up to a final histogram which is saved globally.

2. Which optimizations you chose in the end and why?

I decided to choose strategy 1.3 because this yields better performance especially for larger input problems. Moreover, the comment in the code template asked to implement it with shared memory which was not used in the strategy 1.2.

Furthermore, strategy 1.1 was not implemented due to the previously mentioned reasons.

Additionally, one could improve the algorithm further by splitting the input set to the different blocks so that the number of identical values per thread block are minimal (e.g., by using statistic methods and parameters like variance, …). This would help to avoid thread contention and increase the performance further.

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

For the chosen approach, the following number of global memory reads is performed:

- histogram_kernel:
  <number of elements> + <number of bins>
  Reasoning: There are two atomicAdd operations that require the addition of a value to the global bin variable. For the first one, it is performed for the number of elements and for the second one, it is performed for the number of bins.

- convert_kernel:
  <number of bins>
  Reasoning: For all number of bins, it has to be checked whether they exceed the maximum value.

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

For the chosen approach, the following number of atomic operations is performed:

- histogram_kernel:
  <number of elements> + <number of bins>

- convert_kernel:
  no atomic operations

5. How much shared memory is used in your code? Explain

For the chosen strategy, the following amount of shared memory is used:

<number of blocks> * <number of bins = 4096 in our case> * 4 byte = <number of bins> * 16384 byte.

Explanation:
<number of blocks> * <number of bins> is required because the implementation uses a shared variable of size <number of bins> for each block.

"* 4 byte" is needed because the stored datatype in the bins is an unsigned int. This datatype has a size of 4 byte in the C language.

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?

When atomic operations are performed, it is guaranteed that no threads do not interfere the operations of each other.

If the values of the input would all be different, we would achieve the fastest execution time.

However, when every element in the array has the same value, the contention would be maximal since every thread wants to operate on the same element.

➔ The more the input values are the same, the more contention will occur.

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

- Input length: 262144
- Thread block: 64 threads per block
- Grid: Grid size of 4096 for histogram_kernel and 64 for convert_kernel

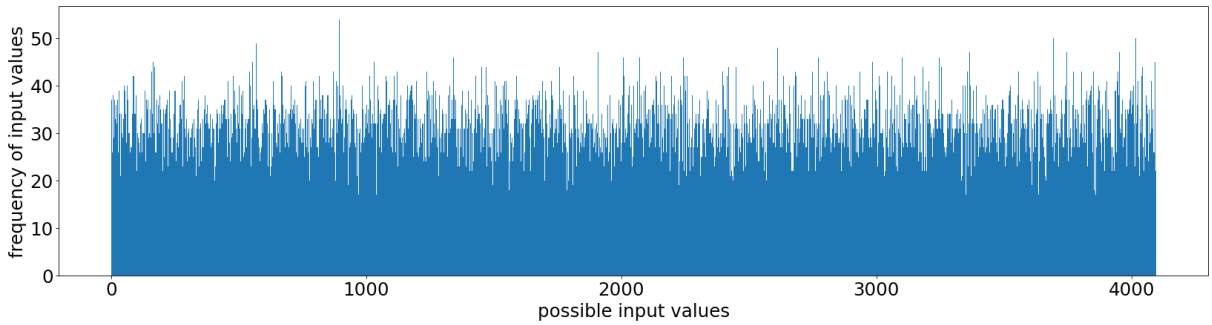

Figure 0-1: Exercise 3 - part 7. – histogram

8. For an input array of 1024 elements, profile with Nvidia Nsight and report Shared Memory Configuration Size and Achieved Occupancy. Did Nvisght report any potential performance issues?

For histogram_kernel:

- Shared Memory Configuration Size: 32.77 Kbyte
- Achieved Occupancy: 6.12%

For convert_kernel:

- Shared Memory Configuration Size: 32.77 Kbyte
- Achieved Occupancy: 9.79%

Yes there were multiple warnings related to performance:

For histogram_kernel:

- `WRN  This kernel grid is too small to fill the available resources on this device, resulting in only 0.0 full waves across all SMs. Look at Launch Statistics for more details.`
- `WRN The grid for this launch is configured to execute only 16 blocks, which is less than the GPU's 40 multiprocessors. This can underutilize some multiprocessors. If you do not intend to execute this kernel concurrently with other workloads, consider reducing the block size to have at least one block per multiprocessor or increase the size of the grid to fully utilize the available hardware resources.`

For convert_kernel:

```
WRN  If you execute __syncthreads() to synchronize the threads of a block, it is recommended to have more than the achieved 1 blocks per multiprocessor. This way, blocks that aren't waiting for _syncthreads() can keep the hardware busy.
```

```
WRN   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.
```

Exercise 4 – A Particle Simulation Application

1. **Describe the environment you used, what changes you made to the Makefile, and how you ran the simulation.**
   - Environment: Google Colab (with same configurations as all previous exercises, e.g., with Tesla T4 GPU)
   - Makefile changes: ARCH=sm_30 has been changed to ARCH=sm_75 as also in all previous tasks and assignments
   - Running the simulation
     make clean
     make all
     ./bin/sputniPIC.out inputfiles/GEM_2D.inp

2. **Describe your design of the GPU implementation of mover_PC() briefly.**

In general, the strategy of the previous tasks has been used here again.

1. Initialize required variables on the GPU
2. Allocate required memory on GPU
3. Copy the memory to the GPU
4. Copy arrays to GPU. This took me multiple hours to figure out why it was not working properly. When copying structs with arrays inside, it is required to copy the arrays manually. Otherwise you will get memory errors since the array pointers are not pointing to the new memory location. This thread was helpful for me to solve that.
5. The parallelization idea is that each particle is simulated in parallel by the kernel. This means that the new mover_PC_GPU() function is very similar to the given mover_PC() function but instead of iterating over all the particles, it is just handled by the threads when the kernel is launched with that function.

6. In the end, again, the arrays must be copied separately. Otherwise, there will be errors again.

(Personal feedback: I'm not sure if I was just completely lost in this problem but it really took me many hours just to find a good way how to debug and solve the memory / result problems with the arrays that were not correctly copied. Maybe it's possible to give a hint in the future lectures with regards to this. Moreover, I would be very interested in how to handle the copying of structs which include other structs which again include arrays, i.e., in general copying multiple embedded datatypes. I didn't find anything useful on the web and for my final project I'm currently trying to achieve something similar).

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

In the provided pdf file (Introduction of spunitGPU.pdf), rho_net was proposed as main quantity for testing. For the file rho_net_10.vtk, both the CPU and the GPU implementation yield the same result.

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

The execution time was compared by using the GEM_2D input file.

```
!./bin/sputniPIC.out inputfiles/GEM_3D.inp
```

Execution time for the CPU:

```
*************************************
   Tot. Simulation Time (s) = 57.0875
   Mover Time / Cycle   (s) = 3.00628
   Interp. Time / Cycle (s) = 2.31729
*************************************
```

Execution time for the GPU:

```
*************************************
   Tot. Simulation Time (s) = 35.0564
   Mover Time / Cycle   (s) = 0.876344
   Interp. Time / Cycle (s) = 2.23858
*************************************
```

It can be observed that the GPU execution time is faster than the CPU version due to the successful exploitation of the parallelization. Almost twice as fast.