Assignment III: CUDA Basics

Franz Kaschner

07.12.2022

Link to Github repository: https://github.com/TheKastenkarl/DD2360HT22

Exercise 1 - Your first CUDA program and GPU performance metrics

1. **Explain how the program is compiled and run.**

Compiling:
```
!nvcc -arch=sm_75 ./lab3_ex1.cu -o lab3_ex1
```

Running:
```
!./lab3_ex1 <vector-length>
```

Profiling:
```
!/usr/local/cuda-11/bin/nv-nsight-cu-cli ./lab3_ex1 <vector-length>
```

2. **For a vector length of N:**
    a. **How many floating operations are being performed in your vector add kernel?**
    N floating point operations (additions).
    b. **How many global memory reads are being performed by your kernel?**
    2N global memory reads for the summands.

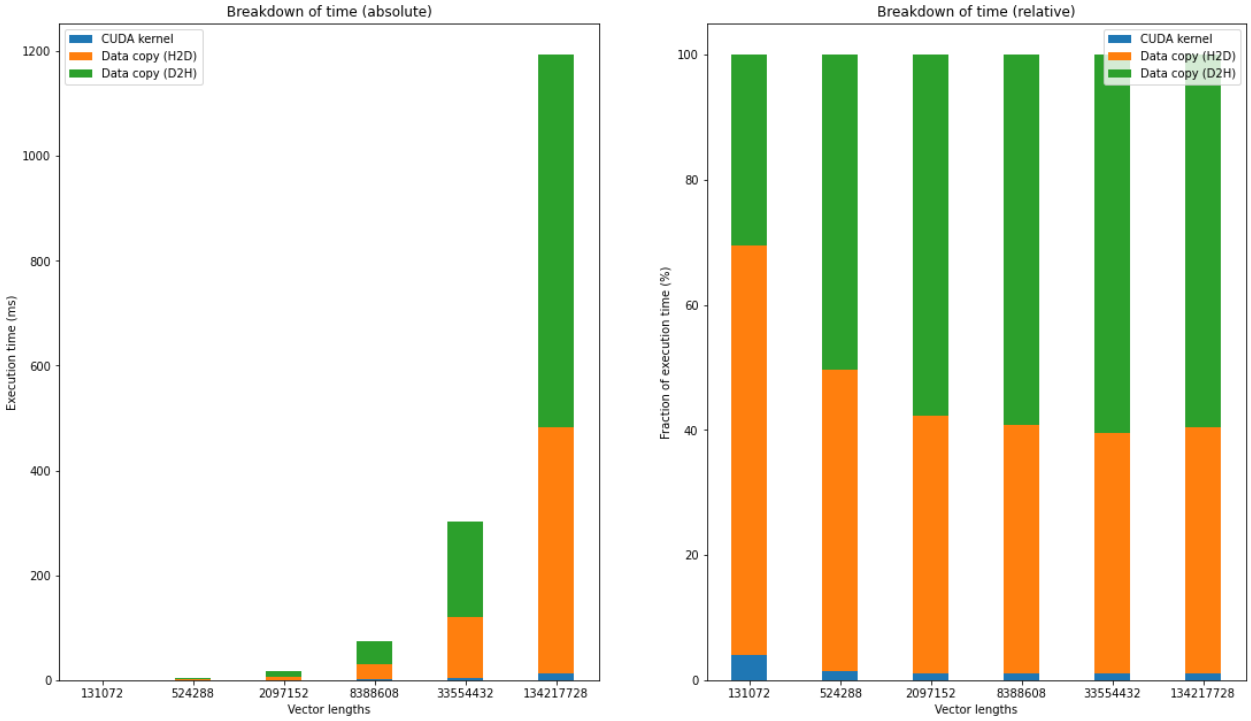
3. **For a vector length of 1024:**
    a. **Explain how many CUDA threads and thread blocks you used.**
    In total, 1024 threads are required as one thread for every addition is created. I use 64 threads per block and [(1024+64-1)/64] = 16 blocks.
    b. **Profile your program with Nvidia Nsight. What Achieved Occupancy did you get? You might find https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html#nvprof-metric-comparison useful.**
    The measured Achieved Occupancy is 6.19%.

4. **Now increase the vector length to 131070:**
    a. **Did your program still work? If not, what changes did you make?**
    Yes, it still worked.
    b. **Explain how many CUDA threads and thread blocks you used.**
    In total, 131070 threads are required as one thread for every addition is created. I use 128 threads per block and [(131070+128-1)/128] = 1024 blocks. Hence, not all threads are really required because we get 128 * 1024 = 131072 threads in total.
    c. **Profile your program with Nvidia Nsight. What Achieved Occupancy do you get now?**
    The measured Achieved Occupancy is 78.04%.

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.**

I use nvprof for measuring the times as it is simpler and more accurate (https://stackoverflow.com/questions/30371030/understanding-cuda-profiler-output-nvprof).

Command: `nvprof ./lab3_ex1 <vector-length>`



Exercise 2 - 2D Dense Matrix Multiplication

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

Robotics, Computer Vision, Neural Networks

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

2 * numARows * numBColumns * numAColumns

Times 2 because for every element you have to do an addition and a multiplication (https://math.stackexchange.com/questions/3512976/proof-of-of-flops-in-matrix-multiplication).

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

2 * numARows * numBColumns * numAColumns + numARows * numBColumns (if a temporal variable is used)

3 * numARows * numBColumns * numAColumns (if no temporal variable is used)

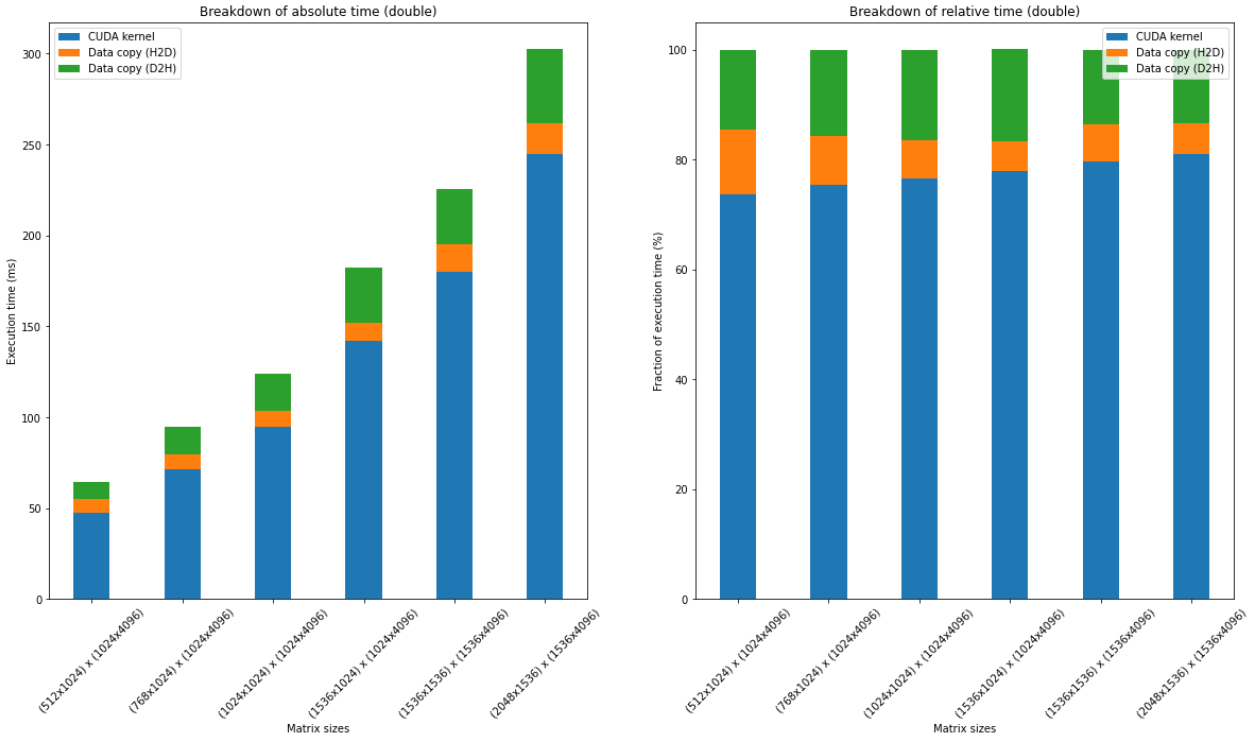
4. **For a matrix A of (128x128) and B of (128x128):**
    a. **Explain how many CUDA threads and thread blocks you used.**
    In total, 128 * 128 = 16384 threads are required as one thread for every element of the output matrix C is created. I use 8 * 8 = 64 threads per block and ⌈(128+8-1)/8⌉ * ⌈(128+8-1)/8⌉ = 16*16 = 256 blocks.
    b. **Profile your program with Nvidia Nsight. What Achieved Occupancy did you get?**
    The measured Achieved Occupancy is 24.21%.

5. **For a matrix A of (511x1023) and B of (1023x4094):**
    a. **Did your program still work? If not, what changes did you make?**
    Yes, it still worked.
    b. **Explain how many CUDA threads and thread blocks you used.**
    In total, 511 * 4094 = 2092034 threads are required as one thread for every element of the output matrix C is created. I use 16 * 16 = 256 threads per block and ⌈(511+16-1)/16⌉ * ⌈(4094+16-1)/16⌉ = 32*256 = 8192 blocks. Hence, not all threads are really required because we get 256 * 8192 = 2097152 threads in total.
    c. **Profile your program with Nvidia Nsight. What Achieved Occupancy do you get now?**
    The measured Achieved Occupancy is 95.86%.

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.**

I use nvprof for measuring the times as it is simpler and more accurate (https://stackoverflow.com/questions/30371030/understanding-cuda-profiler-output-nvprof).

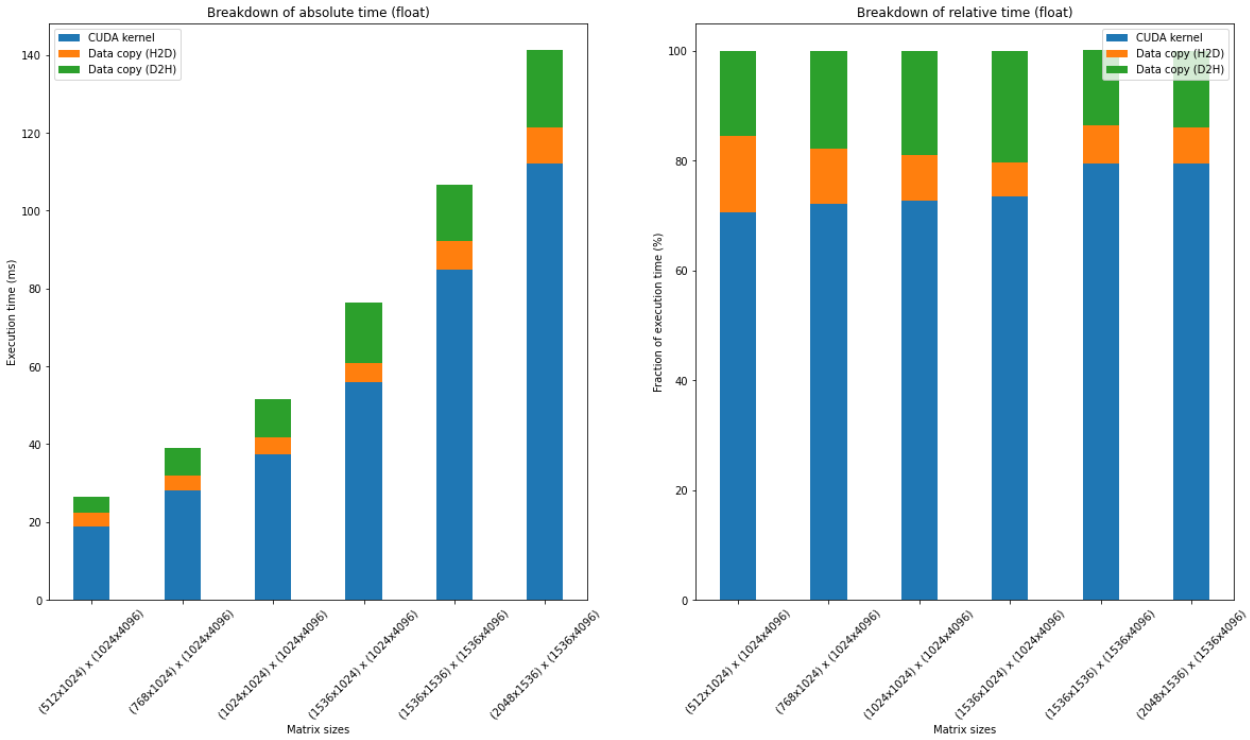
Command: `nvprof ./lab3_ex2 <vector-length>`



It can be observed that most of the time is required for the CUDA kernel. Furthermore, you can see that the fraction of execution time of the CUDA kernel increases with increasing matrix sizes. This is because the number of FLOP increase with O(numARows * numBColumns * numAColumns) whereas the number of elements to copy between host H and device D only increase with O(numARows * numAColumns + numBRows * numBColumns) for H2D and with O(numCRows * numCColumns) for D2H.

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

Bar chart using floats:



The major difference is that the total time is much lower, approximately by factor 2. This is because a float only has half the size of a double.

Exercise 3 - Histogram and Atomics

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

   a) Using atomics to write directly to global memory.
   In approach a), I chose the execution configuration such that the number of threads is equivalent to the number of elements in the input array. Then each thread was responsible for increasing the bin count stored in global memory for one element of the thread. It is necessary to use the atomic operation atomicAdd() to avoid a race condition.

   b) Using atomics in shared memory with final summation of partial histograms.
   In approach b), I used shared memory so that each thread block has its own shared histogram. After creating the shared histograms, the values are copied to the global histogram in global memory. The execution configuration is as in approach a). That made it difficult because for setting the shared histograms at the beginning to zero and for copying the results to global memory, ideally one would have <num_bins> threads and not only <num_elements> threads.

   Using shared memory increases the performance for large input arrays (e.g., with a length of 1,000,000 elements). However, for smaller inputs (e.g., with a length of 1024 elements) approach a) is faster.

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

I would guess that we are more interested in large inputs. Hence, I decided to use approach b) because of the better performance for large inputs.
However, I designed my code such that one could easily switch between the two approaches.

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

histogram_kernel:
Approach a): 2 * <num_elements>
Approach b): <num_bins> (to copy shared histograms to global histogram) + <num_elements>

convert_kernel:
Both approaches: <num_bins>

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

histogram_kernel:
Approach a): <num_elements>
Approach b): <num_bins> + <num_elements> (but only with shared memory)

convert_kernel:
Both approaches: no atomic operations

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

Approach a): no shared memory
Approach b): <num_blocks> * <num_bins> * 4 byte = <num_blocks> * 4096 * 4 byte = <num_blocks> * 16384 byte

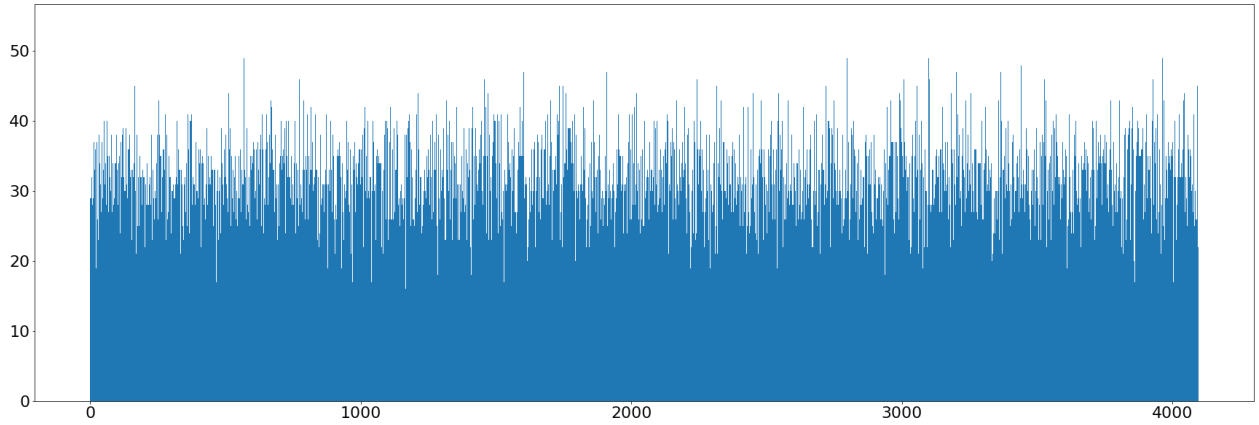
Hence, we need less than 48KB of shared memory per block/SM which is the maximum limit.

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?**

The less uniform the thread is, the larger the contention among threads is. If all array elements had the same value, we would get max. contention.

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



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: 6.13%

Yes, there is the warning: *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
- Changes in Makefile: `ARCH=sm_30` → `ARCH=sm_75`
- Running the simulation:
```
make clean
make all
./bin/sputniPIC.out inputfiles/GEM_2D.inp
```
- For debugging, `cuda-gdb` can be used.

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

To design mover_PC_gpu() is used a similar approach as in the three previous tasks. Hence, first I allocate four variables on the GPU and copy the input arguments of mover_PC_gpu() (→ device-Part, deviceField, deviceGrd, deviceParam) to the GPU. It is really important to copy the arrays in the structs separately because if you don't do it, the pointers in the structs still point to host memory. After that I define the execution configuration and call the kernel. The kernel parallelizes over the particles. Thus, each particles is propagated concurrently. The code of the kernel is quite similar to the code of mover_PC(), except that the loop over the particles is missing. Then the result (only the "part" struct gets modified) has to be copied back to the host. Here, again you must separately copy the arrays to avoid segmentation faults.

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

When you compare the file rho_net_10.vtk as suggested in the file "Introduction of spunitGPU.pdf", you see that both implementations yield the same result.

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

Command: `./bin/sputniPIC.out inputfiles/GEM_2D.inp`

CPU version:
```
**************************************
   Tot. Simulation Time (s) = 86.4147
   Mover Time / Cycle   (s) = 3.47079
   Interp. Time / Cycle (s) = 4.7041
**************************************
```

GPU version:
```
**************************************
   Tot. Simulation Time (s) = 51.5889
   Mover Time / Cycle   (s) = 0.838036
   Interp. Time / Cycle (s) = 3.94097
**************************************
```