TD #2: CUDA

Jonathan Rouzaud-Cornabas (jonathan.rouzaud-cornabas@insa-lyon.fr)

You can fetch incomplete versions of the different codes on Moodle (the same than the one for OpenMP TP).

2 week after the practical session, you should send a report (Evaluation of performance and a short analysis for each part) of what you have done and also your completed codes. The performance evaluation and analysis is as important as the code itself.

You can do the TP alone or by pair (not more). This report will be graded.

The TP contains 5 parts. Each one is more complexe than the previous one. The bare minimum is the first 2 parts.

Performance evaluation methodology:

1. You should modify the performance metric part of the codes to store the value (and it will be better to also store the parameters *e.g.* size of the matrix) into a file (*e.g.* CSV files are perfect for this). It will ease the aggregation of performance data from different runs. It will also help you to store the value without doing manual copies of time into a excel for example (it is a very error prone steps).
2. You should automatize the multiple execution (given a range for each parameter)
3. You should repeat each execution multiple time (at least 10 times)
4. Your figure should contain aggregated value (*i.e.* mean, median and/or standard deviation)
5. You are free to use whatever scripting language (but bash seems to be the minimal language for this and Jupyter Notebook are more certainly the proper way of doing it)
6. Same for drawing the figure, you can use whatever language you want (but Python Numpy/Matplotlib or R/ggplot2 seem to be good choice, you should avoid Excel even if it works)
7. All the experiments should be done and evaluated on the same computer (at least the same configuration, in this case, GPU)
8. If you want, you can compare the performance on different computer (see Moodle if you want to access Grid'5000 computers)
9. If you want, you can try to have a reproduceable approach to ease the replication of your TP even on another computer (for you and for me)

1 Computing PI

The sequential code is available in the file `tp_openmp_part_1_pi.cpp`

Question 1.1 Port the code to CUDA (kernel + memory allocation and copy to/from GPU) with basic parallelism (1 thread per block, N block, with each thread computing a range of value)

Question 1.2 Improve the previous code to support multiple threads per block (2 level reduction)

Question 1.3 Improve the code to use shared memory and atomics

Question 1.4 Implement a full multi-stage reduction

Question 1.5 Evaluate the performance for number of steps : 1000000, 100000000, 10000000000, 1000000000000 and different number of steps per thread/blocks (*e.g.* 1, 32, 64, 128, 256 threads per block and 1, 64, 256, 1024 steps per thread)

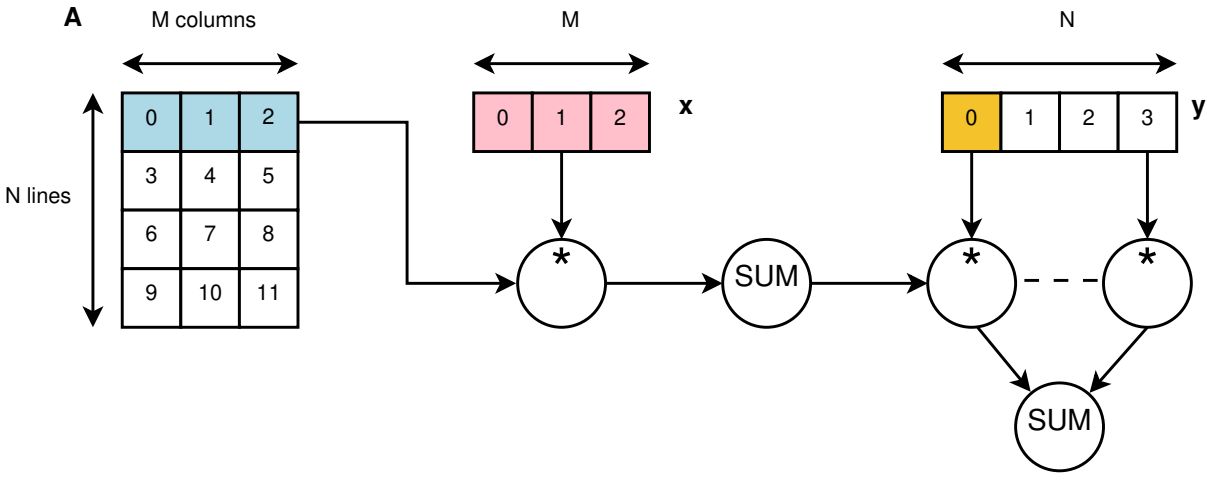

Figure 1: Workflow of the part 2 vector/matrix

Question 1.6 Draw figures to display performance

Question 1.7 Explain your performance

2 Matrix-Vector operation

The sequential code is available in the file `tp_openmp_part_2_vector.cpp`

The Figure 1 describes the algorithm (The algorithm is in comment within the code).

Question 2.1 Complete the code to have a working sequential code.

Question 2.2 Port the code to CUDA with 1 thread per block and 1 block per line (of the matrix A)

Question 2.3 Evaluate the performance for:

- N : 2, 4, 8, 10, 12, 14, 16
- M : 1, 3, 7, 9, 11, 13, 15

Question 2.4 Draw figures to display performance

Question 2.5 Explain your performance

Question 2.6 Add the support of multiple threads per block and atomics

Question 2.7 Evaluate the performance for:

- N : 2, 4, 8, 10, 12, 14, 16
- M : 1, 3, 7, 9, 11, 13, 15

Question 2.8 Draw figures to display performance

Question 2.9 Explain your performance

Question 2.10 Add the support of shared memory and atomics

Question 2.11 Evaluate the performance for:

- N : 2, 4, 8, 10, 12, 14, 16
- M : 1, 3, 7, 9, 11, 13, 15

Question 2.12 Draw figures to display performance

Question 2.13 Explain your performance

3 Matrix multiplication and floating point precision

The sequential code is available in the file `tp_openmp_part_4_matrix_mul.cpp`

Question 3.1 Port the code to CUDA (1 thread per block)

Question 3.2 Evaluate the performance for:

- N : 1000, 4000, 8000, 12000, 18000
- M : 1000, 4000, 8000, 12000, 18000

Question 3.3 Draw figures to display performance

Question 3.4 Explain your performance

Question 3.5 Improve your kernel to support multiple threads per block with shared memory and atomics

Question 3.6 Evaluate the performance for:

- N : 1000, 4000, 8000, 12000, 18000
- M : 1000, 4000, 8000, 12000, 18000

Question 3.7 Draw figures to display performance

Question 3.8 Explain your performance

Question 3.9 Modify the code to change the type of the matrix from double to float

Question 3.10 Evaluate the performance for:

- N : 1000, 4000, 8000, 12000, 18000
- M : 1000, 4000, 8000, 12000, 18000

Question 3.11 Draw figures to display performance

Question 3.12 Explain your performance

Question 3.13 Modify the code to change the type of the matrix from double to half. You should include `half.hpp` and the type is `half_float::half`

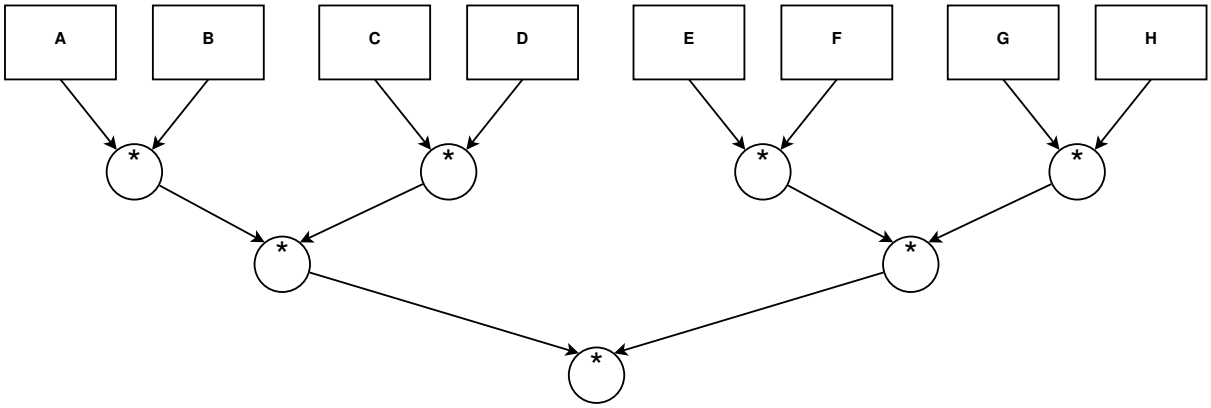

Figure 2: Workflow of the part 2 vector/matrix

Question 3.14 Evaluate the performance for:

- N : 1000, 4000, 8000, 12000, 18000
- M : 1000, 4000, 8000, 12000, 18000

Question 3.15 Draw figures to display performance

Question 3.16 Explain your performance

4 Chaining Matrix Multiplication

You can chain multiple matrix multiplication for example the Figure 2.

Question 4.1 Propose a parallel implementation of such workflow using the code from the previous part (using streams seems to be a good idea)

Question 4.2 Evaluate the performance for:

- N : 1000, 4000, 8000, 12000, 18000
- M : 1000, 4000, 8000, 12000, 18000

Question 4.3 Draw figures to display performance

Question 4.4 Explain your performance