Introduction to CUDA and OpenCL: Shared memory operations

Ilona Tomkowicz, Zofia Pieńkowska

February 15, 2020

Contents

1 Exercise goal

The goal of this exercise was to implement matrix multiplication algorithm and compare calculation times for single-threaded computations and multi-threaded computations using shared memory. CuBLAS toolkit could have not been imported due to lack of necessary dependencies.

2 Implementation

2.1 Shared memory approach

Shared memory is a particularly powerful CUDA feature which allows fast multi-threaded computations. This exceptional speed stems from the fact that shared memory is an on-chip memory, which means faster access and decreasing latencies for frequently used data. To declare memory as shared, we use the `__shared__` variable declaration specifier.

2.2 Single-threaded approach

A simple algorythm of matrix multiplication has been implemented. Its calculations have been significantly slower than those performed on multiple threads. The matrix size has been gradually increased and resulted in segmentation fault at size 850x850.

2.3 Tests and construction of logs

Two tests have been conducted during the execution. The multiplying calculations result in two matrices, one of which has been obtained by a single-threaded algorythm (let us call this matrix S) and the other by multi-threaded algorythm (let us call this matrix M). The first test checks whether all elements of matrix M have been calculated correctly, and if this test is passed, second test is conducted, which compares matrices S and M to each other. Test results have been logged in a log file.

Also, `time.h` library has been included to allow measuring the time elapsed. Time needed for single-threaded computations and for kernel operations has also been logged.

As a result, logs were obtained in following form:

```
Multiplicating matrix 32 x 32.
Kernel operations successful. Time elapsed: 0.000000 s.
Verification test PASSED, multi-threaded calculations are correct.
Comparision test PASSED, single-threaded calculations are correct.
Time elapsed for single-threaded calculations: 0.000000 s.
```

2.4 Data analysis

Following results have been obtained:

Number of elements	**32**	**64**	**128**	**256**	**512**	**600**	**800**
Single-threaded computation time [s]	0	0	0,01	0,12	0,84	0,90	2,29
Multi-threaded computation time [s]	0	0	0	0	0	0	0

For number of elements greater than 800, execution resulted in segmentation fault. For all the cases where execution could have been performed, time elapsed during multi-threaded computations was too small to be effectively measured. However, single-threaded computations time appeared to grow exponentially and has been fitted with function $a \cdot b^x$. The function has been fitted in Gnuplot using the least squares method. Fitted function coefficients are a=0,0703 and b=1,0043. Measured data and the fitted function have been illustrated on fig. 1.

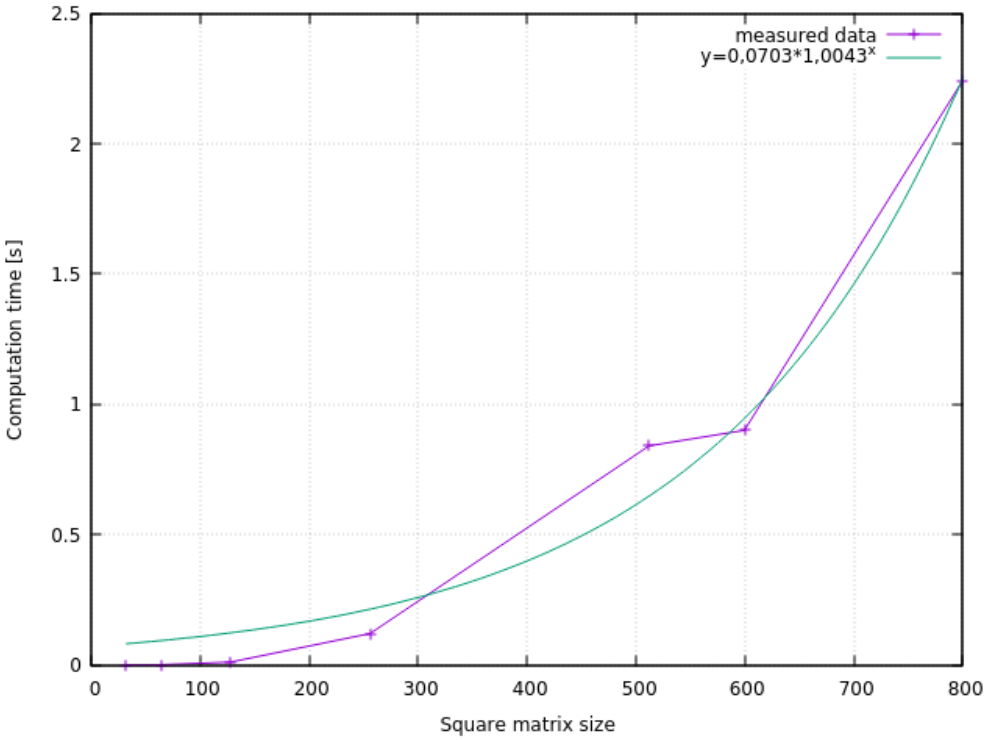

Figure 1: Time elapsed during matrix multiplication as a function of matrix size.

3 Conclusion

It can be seen that time needed for single-threaded computations increases exponentially with respect to matrix size. The use of shared memory allows multi-threaded computations to be even faster, because the memory used is located on-chip in contrast to being local or global. Its name corresponds to the fact that it is *shared* by all threads in a thread block, which allows them to cooperate. The threads still need to be synchronized before proceeding to other calculations to avoid race conditions leading to incorrect results.