Introduction to CUDA and OpenCL final project: Linear algebra toolkit

Mateusz Kaleta

February 2020

1 Introduction

The aim of this project was to create a toolkit application for linear algebra operations. Four operations were considered, namely: matrix addition/subtraction, multiplication and finding inverse using Gauss-Jordan elimination method. Each operation was implemented both for GPU and CPU in order to check the eventual speedup of GPU code. The application has an object oriented structure with different classes managing input/output and delagating tasks specified by the user.

2 Application structure

The application was implemented in C++ with the CUDA specific elements and libraries. It is build upon two classes.

2.1 Matrix class

This class provides the simple representation of a matrix and serves as an interface between user and the program. It stores matrix elements in one dimensional array, which is convenient for computation. It also provides methods to read matrices from text file, print out matrix elements to standard output etc.

2.2 Toolkit classes

Two toolkit classes: CPUToolkit and GPUToolkit, which derive from abstract class Toolkit were implemented. Each class provides its own implementation (CPU and or CPU-style, respectively) of the four considered matrix operations.

3 Launching application

After compilation (e.g. make), the application can be launched command line with the following command, specilising different configurations:

./Toolkit [gpu—cpu] [add—subtract—multiply—inverse] [normal—performance]
where:

- first parameter specifies if we want to use GPU or CPU code,
- second parameter specifies what operation to perform,
- last parameter tell if we want to use "normal" mode: i.e. read matrices from text file and compute the result, or to launch an application in 'performance evaluation mode', i.e. perform the same operation for varying matrix sizes, to see how it influences the computation time.

4 Operations

4.1 Matrix addition/subtraction

The most basic operation implemented is matrix addition/subtraction:

$$\mathbf{C} = \mathbf{A} + \mathbf{B}, \tag{1}$$

which can easily be implemented in usual CPU style. However this requires iterating over every element of the matrix, and performing one operation at a time. The equivalent GPU code delegates the tasks to many cores, so that one core is responsible for single matrix element.

4.2 Matrix multiplication

Considering matrix multiplication, every element C_{ij} in the resulting matrix C requires the computation of dot product of the corresponding row vector A_i and column vector B_j. Since every of this operation is independent, the parallel code can also be implemented. In this application the most standard CUDA-style matrix multiplication was implemented, where one core calculates one element of resulting matrix.

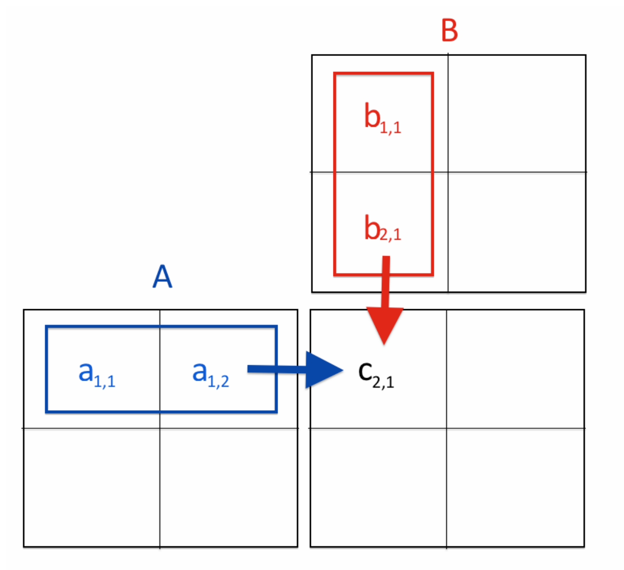

Figure 1: Visualisation of matrix multiplication with GPU.
Source: https://www.quantstart.com/articles/Matrix-Matrix-Multiplication-on-the-GPU-with-Nvidia-CUDA/

4.3 Matrix inversion: Gauss-Jordan elimination

To calculate the matrix inverse, the Gauss-Jordan elimination algorithm was implemented. Suppose we want to find an invese of matrix $\mathbf{A}_{nxn}$. We first create a coresponding identity matrix $\mathbf{1}_{nxn}$ of the same size. Then, by performing elementary operations on both $\mathbf{A}_{nxn}$ and $\mathbf{1}_{nxn}$ we aim to transform $\mathbf{A}_{nxn}$ into identity, which also turns $\mathbf{1}$ into $\mathbf{A}^{-1}$.:

$$\mathbf{A} \rightarrow \mathbf{1} = \mathbf{A}^{-1}\mathbf{A} \tag{2}$$

$$\mathbf{1} \rightarrow \mathbf{A}^{-1}\mathbf{1} = \mathbf{A}^{-1} \tag{3}$$

This operation consist of few steps. The corresponding pseudo-code is presented below.

For every row of $\mathbf{A}$ and $\mathbf{I}$:

- If diagonal element $A_{ii} = 0$, find row element row k, such that $A_{ki} \neq 0$ and add it to row i. Do the same for matrix **I**
- Divide elements in row i by diagonal element A_{ii}:
  $A_{ij} = A_{ij}/A_{ii}$, for $i \neq j$
  $I_{ij} = I_{ij}/A_{ii}$.
- performed row reduction, i.e, subtract the current row multiplied by a scalar from all other rows in **A** and **I**.

5 Results

5.1 Computation time vs. matrix size

The GPU speedup was tested on three operations: addition (since subtraction is computationally of the same complexity), multiplication and Gauss-Jordan elimination. The computation time vs. matrix size is presented on the figures below.

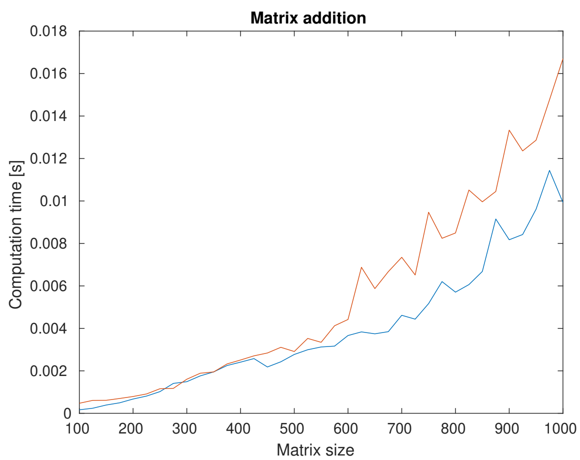

Figure 2: Computation time for matrix addition vs. matrix size with CPU (blue) and GPU(red).

For the matrix addition operation GPU implementation overall performance is comparable (or even worse) as CPU one. This is because additional resources are being spent for initializing memory tranfering data between host and device.

More complex algorithms, such as matrix multiplication and Gauss-Jordan elimination show different trends.

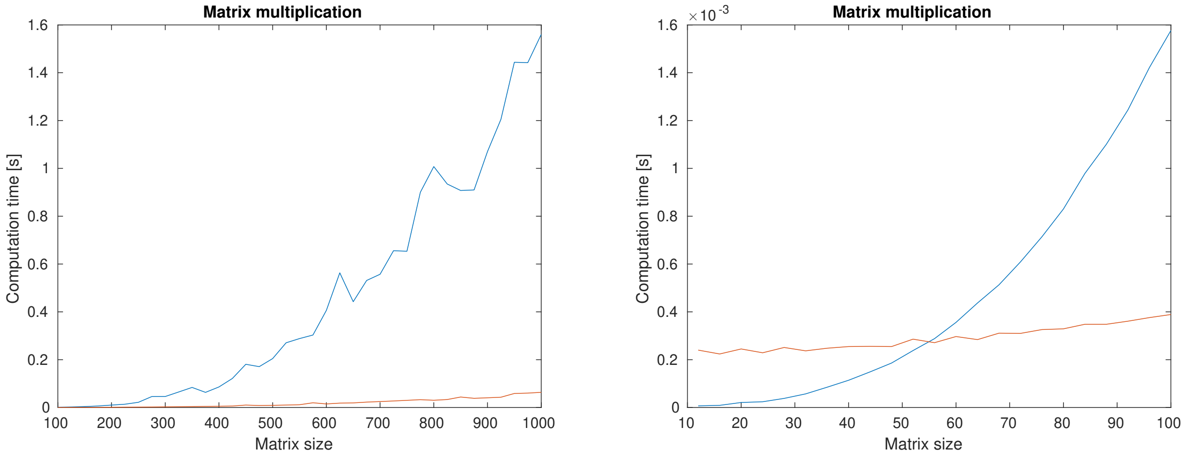

Figure 3: Computation time for matrix multiplication algorithm vs. matrix size with CPU (blue) and GPU(red). On the left panel: matrix size in range 100-1000, on the right: 10-100

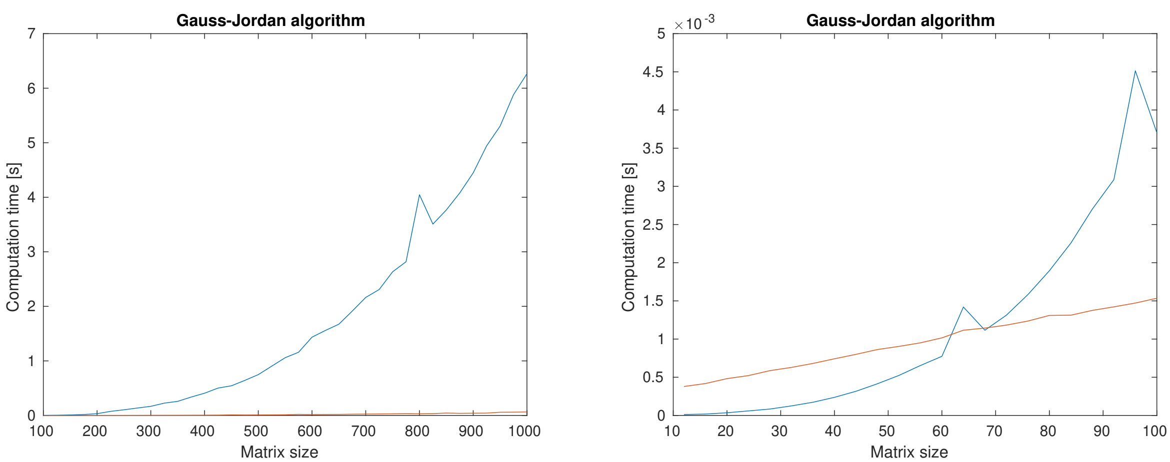

Figure 4: Computation time for Gauss-Jordan algorithm vs. matrix size with CPU (blue) and GPU(red). On the left panel: matrix size in range 100-1000, on the right: 10-100

5.2 Nvprofiler analysis

An additional analysis was performed with the use of nvprof. Below are listed GPU activities for two example operations: matrix addition and Gauss-Jordan algorithm.

```
==31580== Profiling application: ./Toolkit gpu add normal
==31580== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   50.99%  2.4640us         3     821ns     736ns     960ns  [CUDA memcpy HtoD]
                   32.45%  1.5680us         1  1.5680us  1.5680us  1.5680us  add_kernel(double*, double*, double*, int, int)
                   16.56%     800ns         1     800ns     800ns     800ns  [CUDA memcpy DtoH]
      API calls:   99.58%  210.49ms         3  70.164ms  8.7840us  210.47ms  cudaMalloc
                    0.14%  296.05us         1  296.05us  296.05us  296.05us  cuDeviceTotalMem
                    0.11%  230.82us        96  2.4040us     281ns  91.470us  cuDeviceGetAttribute
                    0.07%  143.55us         3  47.848us  10.162us  115.57us  cudaFree
                    0.04%  84.044us         4  21.011us  10.948us  35.892us  cudaMemcpy
                    0.04%  74.819us         1  74.819us  74.819us  74.819us  cudaLaunchKernel
                    0.02%  52.023us         1  52.023us  52.023us  52.023us  cuDeviceGetName
                    0.00%  4.9840us         1  4.9840us  4.9840us  4.9840us  cuDeviceGetPCIBusId
                    0.00%  3.0990us         3  1.0330us     317ns  2.2710us  cuDeviceGetCount
                    0.00%  1.3580us         2     679ns     327ns  1.0310us  cuDeviceGet
                    0.00%     575ns         1     575ns     575ns     575ns  cuDeviceGetUuid
[cuda-s02@lhcbgpu1 Default]$
```

Figure 5: Profiling result for matrix addition operation: For matrix size n=10

```
==29809== Profiling application: ./Toolkit gpu inverse normal
==29809== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   39.22%  23.105us        20  1.1550us  1.1200us  1.4720us  reduce_row(double*, double*, int, int, bool)
                   36.77%  21.664us        10  2.1660us  1.6320us  3.5200us  normalize_row(double*, double*, int, int)
                   18.41%  10.848us        10  1.0840us     992ns  1.7280us  find_pivots(double*, double*, int, int)
                    2.88%  1.6960us         2     848ns     736ns     960ns  [CUDA memcpy HtoD]
                    2.72%  1.6000us         2     800ns     800ns     800ns  [CUDA memcpy DtoH]
      API calls:   99.57%  226.25ms         2  113.13ms  7.6690us  226.24ms  cudaMalloc
                    0.14%  322.26us        40  8.0560us  6.1850us  60.026us  cudaLaunchKernel
                    0.12%  271.53us         1  271.53us  271.53us  271.53us  cuDeviceTotalMem
                    0.07%  164.92us        96  1.7170us     158ns  70.664us  cuDeviceGetAttribute
                    0.05%  105.95us         2  52.977us  13.227us  92.727us  cudaFree
                    0.03%  67.484us         4  16.871us  8.3650us  27.253us  cudaMemcpy
                    0.02%  39.975us         1  39.975us  39.975us  39.975us  cuDeviceGetName
                    0.00%  5.8750us         1  5.8750us  5.8750us  5.8750us  cuDeviceGetPCIBusId
                    0.00%  2.5980us         3     866ns     337ns  1.7540us  cuDeviceGetCount
                    0.00%  1.1030us         2     551ns     219ns     884ns  cuDeviceGet
                    0.00%     292ns         1     292ns     292ns     292ns  cuDeviceGetUuid
```

Figure 6: Profiling result for Gauss-Jordan algorithm: For matrix size n=10

It is visible that for the addition operation, the actual computation (by kernel) accounts for less than 50% of the GPU time, and the rest is spent to transfer data between host and device. For Gauss-Jordan elimination, on the other hand, most of the GPU time is spent by three kernels performing calculation, which means that we take more advantage of the parallel architecture.

6 Summary

A toolkit for linear algebra operations was implemented both on GPU using CUDA architecture, CPU code was also implemented as a reference point. The results (Fig. 2-4) show that GPU implementation provides significant speedup for algorithms such as matrix multiplication and Gauss-Jordan elimination, wheareas is not very efficient for operations such as matrix addition/subtraction.