AGH

Faculty of Physics and Applied Computer Science

Multicore Programming

Linear Algebra Toolkit

Artur Kucia

February 19, 2017

1 Introduction

The task for this project was to create an efficient application for solving sets of linear equations using the Nvidia CUDA technology. CUDA is an extension for the C and C++ language (and Fortran) that enables General-Purpose computing on GPUs applicable devices. It takes advantage of the the parallel multi-core architecture of the modern GPUs.

2 Algorithms

2.1 Gauss-Jordan elimination

For solving set of linear equations the Gauss-Jordan elimination algorithm performs sequence of elementary operations on rows: multiplying a row by a nonzero scalar and adding two rows together.
We can use the Gauss-Jordan algorithm to find the inverse matrix $\mathbf{A^{-1}}$ of the matrix $\mathbf{A}$ and then use it's properties to solve the system of equations:

$$\mathbf{Ax} = \mathbf{b} \tag{2.1}$$

$$\mathbf{A^{-1}Ax} = \mathbf{A^{-1}b} \tag{2.2}$$

$$\mathbf{x} = \mathbf{A^{-1}b} \tag{2.3}$$

To find the inverse matrix $\mathbf{A^-1}$ we first create the identity matrix $\mathbf{I}$ of the same size and by performing elementary operations on both matrices transform the matrix $\mathbf{A}$ into a new identity matrix. The algorithm looks as follows:
for every row j=1,2,...,N of matrix **A** and **I**:

1. if a_{jj} is equal to 0:
   find row k such that a_{kj} is not equal to 0 and add it to row j

2. divide the elements of row by the diagonal element a_{jj} for columns i=1,2,...,j, i≠j:

$$i_{ij} = i_{ij}/a_{jj} \tag{2.4}$$

$$a_{ij} = a_{ij}/a_{jj} \tag{2.5}$$

3. divide the diagonal elements of **I**:

$$i_{jj} = i_{jj}/a_{jj} \tag{2.6}$$

4. subtract current row from all other rows in matrix **A** and **I**

When the inverse of the matrix **A** is computed, multiply it by the vector **b** to solve the system of equations.

2.2 LU decomposition

An LU decomposition refers to the factorization of matrix A into two matrices: a lower triangular matrix L and an upper triangular matrix U that they satisfy the equation:

$$\mathbf{A} = \mathbf{LU} \tag{2.7}$$

For the case of A being a 3 by 3 matrix it's factorization would look like this:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = \begin{pmatrix} \alpha_{11} & 0 & 0 \\ \alpha_{21} & \alpha_{22} & 0 \\ \alpha_{31} & \alpha_{32} & \alpha_{33} \end{pmatrix} \begin{pmatrix} \beta_{11} & \beta_{12} & \beta_{13} \\ 0 & \beta_{22} & \beta_{23} \\ 0 & 0 & \beta_{33} \end{pmatrix} \tag{2.8}$$

To compute L and U matrices we apply the Crout's algorithm and update the matrix **A** in place:

for every column j=1,2,...,N of matrix **A**:

1. if a_{jj} is equal to 0:
   find row k such that a_{kj} is not equal to 0 and add it to row j

2. update elements of **L** for columns i=1,2,...,j :

$$a_{ij} = a_{ij} - \sum_{k=1}^{j-1} a_{ik}a_{kj} \tag{2.9}$$

3. update elements of **U** for rows i=j+1,j+2,...,N :

$$a_{ji} = \frac{1}{a_{jj}}[a_{ji} - \sum_{k=j+1}^{N} a_{jk}a_{kj}] \tag{2.10}$$

If we have L and U matrices we can transform the set of equations:

$$\mathbf{Ax} = \mathbf{b} \tag{2.11}$$

$$(\mathbf{LU})\mathbf{x} = \mathbf{b} \tag{2.12}$$

$$\mathbf{L}(\mathbf{Ux}) = \mathbf{b} \tag{2.13}$$

$$\tag{2.14}$$

solve the for the vector **y** first:

$$\mathbf{L}\mathbf{y} = \mathbf{b} \tag{2.15}$$

and then solve for the final solution **x**:

$$\mathbf{U}\mathbf{x} = \mathbf{y} \tag{2.16}$$

Solving the triangular sets of equations is easy. For 2.15 can be done as follows:

$$y_1 = \frac{b_1}{\alpha_{11}} \tag{2.17}$$

$$y_i = \frac{1}{\alpha_{ii}} \left[b_i - \sum_{j=1}^{i-1} \alpha_{ij} y_j \right], i = 2, 3, ..., N \tag{2.18}$$

And for 2.16:

$$x_N = \frac{y_N}{\beta_{NN}} \tag{2.19}$$

$$x_i = \frac{1}{\beta_{ii}} \left[y_i - \sum_{j=i+1}^{N} \beta_{ij} x_j \right], i = N-1, N-2, ..., 1 \tag{2.20}$$

3 Hardware

The computer used for the project was equipped with the following hardware:

- **CPU** Intel Core i5-5200U CPU, 2.20GHz
- **GPU** Nvidia GeForce 820M, 900 MHz, 96 CUDA cores, 964 Mb of memory, CUDA Compute Capability 2.1

4 Application Architecture

4.1 Matrix and Vector classes

Matrix class objects store matrices as a floating point 1 dimensional arrays and two variables corresponding to the dimensions of a stored matrix. It also implements useful interfaces for retrieving those variables, saving and loading matrices to text files and a matrix multiplication function.
The Vector class inherits from Matrix and stores the n by 1 dimensional matrix.

4.2 Solver classes

Application uses two classes CPUSolver and GPUSolver that both implement the same interfaces for solving set of equations: solveGJ and solveLUD.

5 Input and output

5.1 Command line interface

User of the application has the following commands available:

- -input pathToMatrix pathToVector input txt files
- -output pathToOutput output txt file
- -random N uses randomized matrix and vector of size N as an input
- -gpu uses gpu to perform calculations
- -lud uses LU decomposition algorithm for solving set of equations
- -gj uses Gauss-Jordan elimination algorithm for solving set of equations
- -scr prints results to screen

6 GPU Kernels

Two fully utilize the GPU each block requires about 256 (or its multiples) threads executed in parallel.

6.1 Gauss Jordan elimination

The algorithm which computes the inverse of the matrix was split into four kernels:

6.1.1 kernel jj

It uses 1D grid of 1D blocks to find diagonal element which is equal to 0 and add another row to it in parallel. (step 1)

6.1.2 kernel normalize row

Uses 1D grid of 1D blocks to divide every element in current row in matrix **A** and **I**. (step 2 and 3)

6.1.3 kernel reduce row i and kernel reduce row a

Use 2D grids of 2D blocks to subtract current row from every other rows in both matrices. (step 4)

The multiplication of $\mathbf{A^{-1}}$ is conducted on CPU.

6.2 LU decomposition

Three kernels were used to run the LUD on GPU. Each implements one of the steps of the decomposition algorithm. The task of solving the two triangular systems of equations was left for the CPU as they cannot be solved in parallel.

6.2.1 kernel jj

This kernel uses 1D grid of 1D block and looks for rows with diagonal elements that are equal to 0 and adds next suitable row to it. The addition is done in parallel. (step 1)

6.2.2 kernel update U

Uses 1D grid of 1D blocks. Each thread computes the sum and updates different element of the jth column of the U matrix (step 2 equation 2.10).

6.2.3 kernel update L

Uses 1D grid of 1D blocks. Each thread computes the sum and updates different element of the jth row of the L matrix (step 3 equation 2.9).

7 Benchmarking

7.1 Computation time

The performance of both algorithms was tested for the matrices of size 2 to 1000. The measured times are shown on the plots below.

7.1.1 Performance of the Gauss-Jordan elimination algorithm

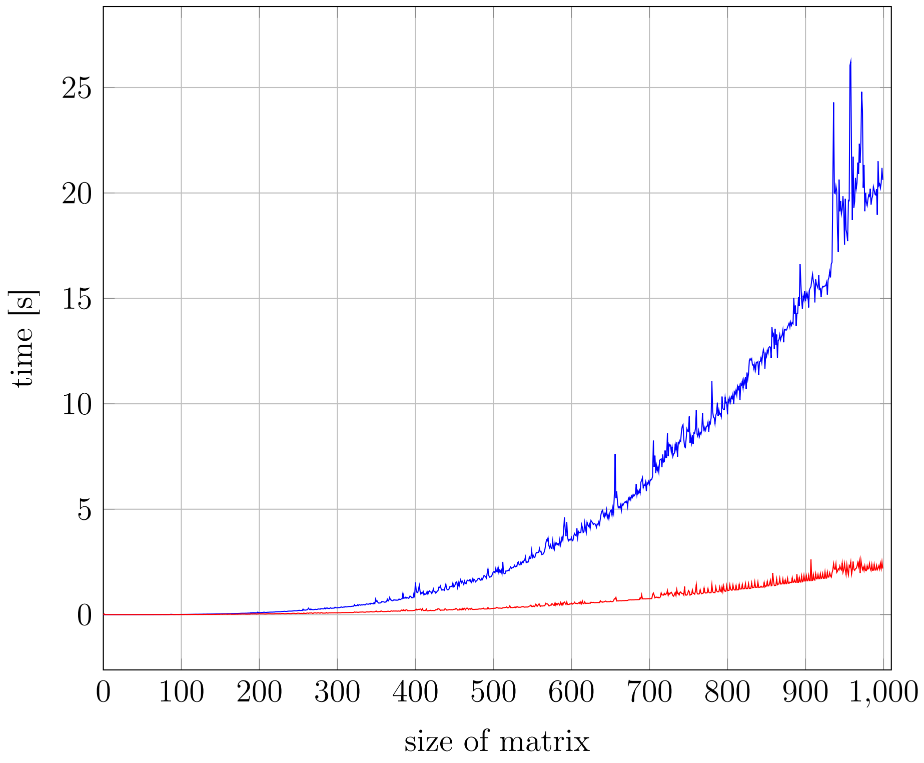

Figure 1: The performance of the CPU (blue) and GPU (red) versions of Gauss-Jordan elimination algorithm.

It is clearly visible that the GPU version of the algorithm outperforms the CPU for the matrices of size greater than about 250. The gain grows very rapidly with the increasing matrix size and for the matrices of size of about 1000 by 1000 it is almost 10 times faster.

7.1.2 Performance of the LU decomposition algorithm

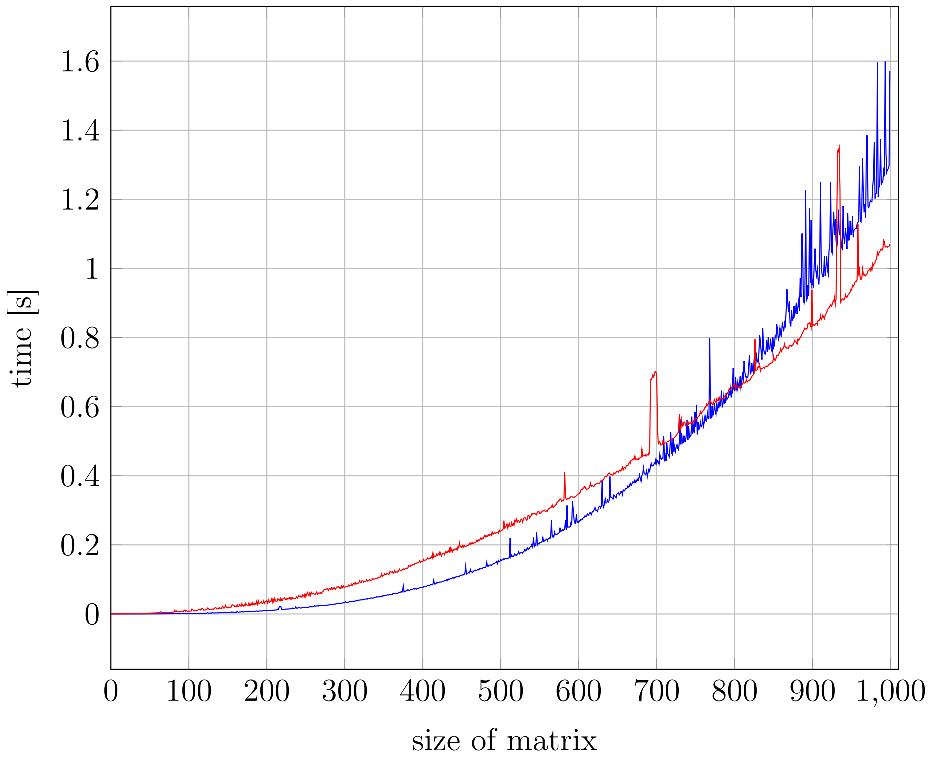

Figure 2: The performance of the CPU (blue) and GPU (red) versions of LU decomposition algorithm.

The performance gain while using the GPU to compute the LU decomposition is not as great as with the Gauss-Jordan algorithm. The GPU can run faster only for the matrices of size greater than 800. It is worth noting, that both version of the algorithm run many times faster than while using the Gauss-Jordan elimination.

7.2 Branch efficiency

Using the Nvidia profiler tool the branch efficiency of kernels for both algorithms was measured:

Table 1: Measured average branch efficiency for the Gauss-Jordan algorithm

	matrix size 10 by 10	matrix size 100 by 100	matrix size 1000 by 1000
kernel jj	96.15%	96.88%	99.98 %
kernel normalize row	94.45%	97.29%	99.56 %
kernel reduce row i	74.55%	96.25%	99.58 %
kernel reduce row a	73.97%	96.03%	99.55 %

Table 2: Measured average branch efficiency for the LU decomposition

	matrix size 10 by 10	matrix size 100 by 100	matrix size 1000 by 1000
kernel jj	95.95%	96.86%	99.39 %
kernel update U	96.54%	99.06%	99.97 %
kernel update L	96.53%	99.06%	99.97 %

The results show, that branch efficiency is on acceptable level.

8 Conclusions

The developed GPU application is considerably faster while solving very large sets of equations. The speedup achieved in Gauss-Jordan elimination algorithm was up do 10x for very large matrices. The improvement of the LU decomposition algorithm was very small and achieved only for matrices as big as 800 by 800 elements.