FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Gauss-Jordan Solver of Linear Equation Systems on GPU
Student:	Emil Eyvazov
Supervisor:	doc. Ing. Ivan Šimeček, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	until the end of summer semester 2022/2023

Instructions

1) Familiarize with the existing methods for Gauss-Jordan method on GPU [1,2].
2) Implement Gauss-Jordan method using CUDA
3) Measure the performance of your solver and compare it with existing solutions.
4) Discuss the results from 3).

[1] https://www.sciencedirect.com/science/article/pii/S2212017312000096
[2] https://link.springer.com/article/10.1007/s11227-013-1043-3

Electronically approved by doc. Ing. Jan Janoušek, Ph.D. on 17 February 2021 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Gauss-Jordan Solver of Linear Equation Systems on GPU

Emil Eyvazov

Department of . . . Computer Science
Supervisor: doc. Ing. Ivan Šimeček, Ph.D.

January 3, 2022

Declaration

In Prague on January 3, 2022 …………………

Citation of this thesis

Eyvazov, Emil. *Gauss-Jordan Solver of Linear Equation Systems on GPU.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.

Abstrakt

Tato práce se zabývá řešením soustav lineárních rovnic pomocí různých řešičů na GPU i na CPU. Řešitelé jsou mezi sebou testováni na čas provedení a správnost výpočtů.

Klíčová slova Gauss-Jordan, GPU, CUDA, částečné otáčení, paralelní redukce.

Abstract

This thesis is about solving systems of linear equation via different solvers on GPU as well as on CPU. Solvers are tested between each other in time of execution and correctness of calculations.

Keywords Gauss-Jordan, GPU, CUDA, Partial Pivoting, Parallel Reduction.

Contents

List of Figures

Chapter 1

Introduction

The goal of this project is to implement System of Linear Equations (SLE) with Gauss-Jordan elimination method on Graphics Processing Unit (GPU) using Compute Unified Device Architecture (CUDA) technology.

As SLEs are used widely used in many fields, an efficient algorithm that would solve them is very essential. SLE could be solved via different mathematical methods, one of the most traditional ones being Gaussian elimination [1] and its variation Gauss-Jordan elimination [2] methods. There is an implicit parallelism in solving SLE.

We will look at why GPU is very efficient in solving highly parallelized tasks and compare the solution on GPU with the solution on CPU. We will also cover CUDA technology that is available only on NVIDIA GPUs. Gauss-Jordan method will be implemented on GPU and on CPU for reference. Partial pivoting variations of Gaussian elimination will also be implemented and tested between each other. cuBLAS library will also be used to substitute Gauss-Jordan elimination with Gaussian elimination with partial pivoting strategies. Besides traditional Gaussian elimination with various pivoting strategies, LU decomposition of matrices, implemented via NVIDIA's cuSOLVE library, will also be covered.

All the previously mentioned methods will be compared with each other based on time of execution and correctness of calculations. Tests will be done with double and single precision variations of all solvers. Special algorithm will be implemented on CPU with extended double precision type of values to be a reference solver: all other solvers' results will be compared with reference solver's results.

Testing will consist of 2 parts: comparison of basic Gauss-Jordan algorithm implementations on CPU and GPU — this test will serve the purpose of showing difference in performance on CPU and GPU; comparison of all solvers' on GPU results with reference solution on CPU — this test will be used to evaluate precisions of different solvers on GPU and get the most precise algorithm.

CHAPTER 2

Terms and definitions

The System of Linear Equations (SLE) are used in many engineering and scientific problems. SLE with n unknowns and m equations:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + ... + a_{2n}x_n &= b_2 \\ &. \\ &. \\ &. \\ a_{m1}x_1 + a_{m2}x_2 + ... + a_{mn}x_n &= b_m \end{aligned}$$

Could be converted to matrix with m rows and n columns, as illustrated below, where matrix A is matrix of coefficients, b is vector of results of each equation, and x is vector of unknowns. To learn more about SLE, see [3]:

$$\mathrm{A} = \begin{bmatrix} a_{11} & a_{12} & . & . & . & a_{1n} \\ a_{21} & a_{22} & . & . & . & a_{2n} \\ . & . & . & & & . \\ . & . & & . & & . \\ . & . & & & . & . \\ a_{m1} & a_{m2} & . & . & . & a_{mn} \end{bmatrix}, \mathrm{x} = \begin{bmatrix} x_1 \\ x_2 \\ . \\ . \\ . \\ x_n \end{bmatrix}, \mathrm{b} = \begin{bmatrix} b_1 \\ b_2 \\ . \\ . \\ . \\ b_n \end{bmatrix}$$

Gaussian elimination is an algorithm for solving SLE. It consists of a sequence of operations performed on the corresponding matrix of coefficients, which bring matrix to its row echelon form. See [1] for basic Gaussian elimination. See [4] for matrix in row echelon form. To bring matrix to *reduced row echelon* form, Gauss-Jordan algorithm is used, see [2] for Gauss-Jordan algorithm.

Matrix is in row echelon form, if it has following properties:

- Any row consisting entirely of zeros occurs at the bottom of the matrix.

- For each row that does not contain entirely zeros, the first non-zero entry is 1 (called a leading 1).
- For two successive (non-zero) rows, the leading 1 in the higher row is further left than the leading one in the lower row.

Matrix is in reduced row echelon form, if it has following properties:

- All of the rows containing nonzero entries sit above any rows, whose entries are all zero.
- The first nonzero entry of any row, called the *leading entry* of that row, is positioned to the right of the leading entry of the row above it.

Basically, matrix in reduced row echelon form has nonzero values only in pivot entries.

2.1 Common notation of variables in algorithms

Some algorithms will have common variables:

- *matrix_size*: height and width of the matrix, e.g. $matrix\_size = 500$ means that it is matrix with 500×500 entries. In other words, SLE consists of 500 equations.
- *pivot_id*: index of the current pivot in the matrix.
- *m_A*: left-hand side of the SLE, i.e. matrix A — matrix of coefficients.
- *m_B*: right-hand side of the SLE, i.e. matrix B.
- *m_C*: SLE variables' values that will be calculated after matrix A and matrix B are calculated, i.e. matrix C.

2.2 Gauss-Jordan method pseudo-code

Algorithm 1 illustrates a pseudo-code for Gauss-Jordan method:

1. First, matrix will be processed in top-down fashion, as illustrated on the left matrix in Figure 1:
   a) Pivot is chosen(line 1).
   b) Row under the pivot that will be updated is chosen and ratio of pivot entry with entry in the pivot column for given row under the pivot is calculated and stored, as a multiplicative, in *mult* variable(lines 2-3).

c) Every column entry of the row chosen in line 2 is updated with regards to the multiplicative calculated in line 3(lines 4-6).

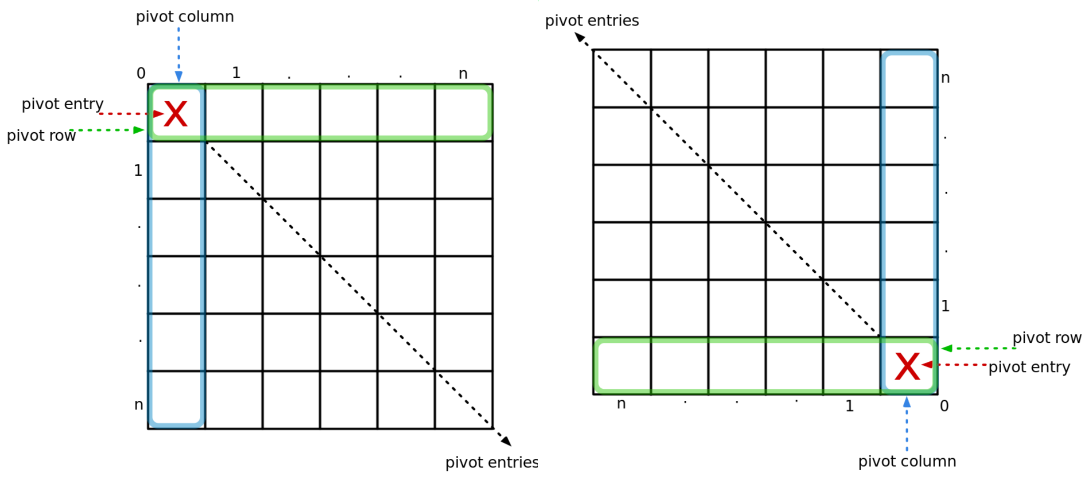
2. After bringing matrix to row echelon form, it will be brought to reduced row echelon form, as illustrated on the right matrix in Figure 1:

   a) The process is similar to the described above with only difference that matrix will be processed in down to top fashion(lines 9-16).

Algorithm 1 Gauss-Jordan algorithm

```
 1: for $i = 0$ to $matrix\_size - 1$ do
 2:     for $y = i + 1$ to $matrix\_size - 1$ do
 3:         $mult \leftarrow matrix[y][i]/matrix[i][i]$
 4:         for $x = i$ to $matrix\_size - 1$ do
 5:             $matrix[y][x] \leftarrow matrix[y][x] - mult \cdot matrix[i][x]$
 6:         end for
 7:     end for
 8: end for
 9: for $i = matrix\_size - 1$ downto $0$ do
10:     for $y = 0$ to $i - 1$ do
11:         $mult \leftarrow matrix[y][i]/matrix[i][i]$
12:         for $x = 0$ to $i$ do
13:             $matrix[y][x] \leftarrow matrix[y][x] - mult \cdot matrix[i][x]$
14:         end for
15:     end for
16: end for
```

Figure 1: Matrix that will be processed in top-down and down-top fashion



Chapter 3

Software aspects

3.1 GPGPU and CUDA

General Processing on Graphics Processing Unit (GPGPU) is used to implement non-graphical tasks on GPU. GPU has a lot more cores than CPU, but those cores do not have that much functionality that CPU cores have, which make GPU cores highly specialized for particular tasks and, as there are a lot of them, it is possible to implement highly parallelized code that will run on GPU.

Compute Unified Device Architecture (CUDA) is a parallel computing platform and API model created by NVIDIA. It allows developers to use CUDA-enabled cores for GPGPU. It is a software level that gives access to instruction set of GPU. CUDA code is mostly written in C/C++. Special extension **.cu** is used to recognize CUDA code.

Code that runs on GPU is called a *device* code, whereas all code, executed on CPU is called *host* code. When host recognizes device code, host launches device code on GPU and continues running leftover host code, so host is not blocked by device code. To learn more about CUDA, see [5].

Every NVIDIA GPU has **computability**, which determines the capability of GPU to run particular CUDA instructions. For the purpose of this article, NVIDIA GeForce RTX 2080 Ti was used with computability 7.5.

3.1.1 Kernel

Kernel is a parallel code that is launched and executed on a device by many threads at once. Consists of a grid that contains multiple blocks and is configurable by developer.

Each block is executed by multiple threads at once. There is a maximum number of threads that is possible to allocate for one block. For GPU, used in this thesis(RTX 2080 Ti), this maximum is 1024 threads per each block. To learn more about CUDA kernels, see [6]. In code, kernel functions are written with __**global**__ prefix.

In Algorithm 2, 10 x 10 = 100 blocks are created with 32 x 32 = 1024 threads for each block. Depending on the GPU, all blocks could run in parallel or be scheduled for latter execution on device.

Algorithm 2 Kernel call

```
1: $dim3\ grid(10, 10),\ block(32, 32)$
2: $custom\_kernel <<< grid, block >>> ()$
```

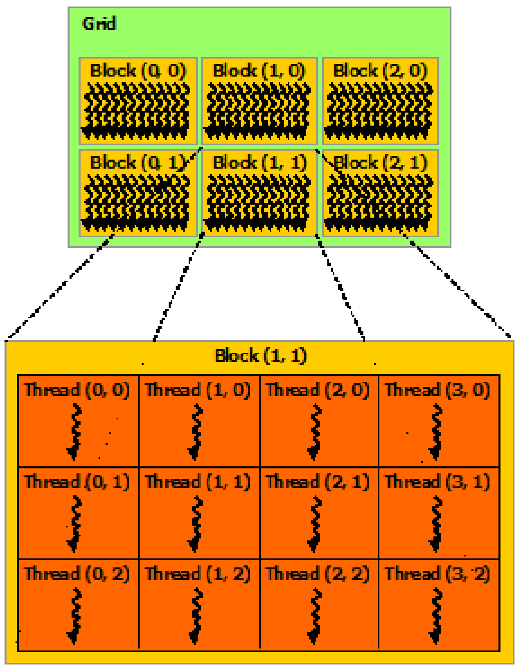


Figure 2: CUDA kernel

Figure 2 illustrates CUDA kernel. Image source: [7].

3.1.2 Memory

CUDA memory is divided into global memory, local memory, and shared memory.

3.1.2.1 Global memory

Global memory available for all cores of the GPU. Has the greatest memory space compared to any other type of memory in CUDA memory hierarchy, though it is the slowest.

The GPU, used in this thesis(RTX 2080 Ti), has 11 GB of global memory.

3.1.2.2 Shared memory

Shared memory is located on chip, which makes it much faster than global memory. Shared memory is roughly 100 times faster than global memory. During CUDA kernel call, the amount of shared memory(in bytes) could be defined. The GPU, used in this thesis(RTX 2080 Ti) has 49 KB of shared memory per block. To learn more about shared memory, see [8].

In Algorithm 3, shared memory of 16 B is allocated for each block of called kernel.

Algorithm 3 Shared Memory size declaration

```
1: $int\ shared\_memory\_size \leftarrow 2 \cdot sizeof(double)$
2: $custom\_kernel <<< grid,\ block,\ shared\_memory\_size >>> ()$
```

3.1.2.3 Local memory

Each thread of the kernel block gets its limited amount of local memory, which is accessible only by this particular thread. Very close to stack memory for each function in C language.

3.1.3 Device memory allocation in code

As host memory and device memory are separate, host memory can't be accessed in device code. Device memory should be allocated beforehand on host. It is achieved by *cudaMalloc()*.

Freeing is achieved by *cudaFree()*. Memory copying from device to host and vice versa is achieved via *cudaMemcpy()*.

3.2 Basic Gauss-Jordan algorithm implementation on CPU

Basic Gauss-Jordan algorithm consists of finding ratio of pivot column of the pivot row with every lower row's pivot column and further subtraction of pivot row with all lower rows multiplied by that ratio.

Doing it in top-to-down fashion, will bring matrix to row echelon form. Doing it again, but in down-to-top fashion, will bring it to reduced row echelon form.

Algorithm 4 illustrates bringing of matrix A(the matrix of coefficients) into row echelon form with modification of matrix B(right-hand side of SLE). In lines 7-13, matrix of coefficients is modified per one row. In line 14, the matrix B(right-hand side of SLE) is modified.

Algorithm 4 Top-down triangulation on CPU

```
 1: for $i = 0$ to $matrix\_size - 1$ do
 2:     $pivot \leftarrow matrix\_A[i][i]$
 3:     for $y = i + 1$ to $matrix\_size - 1$ do
 4:         $row\_pivot \leftarrow matrix\_A[y][i]$
 5:         $mult \leftarrow row\_pivot/pivot$
 6:         for $x = i$ to $matrix\_size - 1$ do
 7:             $m\_A[y][x] \leftarrow m\_A[y][x] - mult \cdot m\_A[i][x]$
 8:         end for
 9:         $m\_A[y][i] \leftarrow 0$
10:         $m\_B[y] \leftarrow m\_B[y] - mult \cdot m\_B[i]$
11:     end for
12: end for
```

After bringing the matrix A in row echelon form and modifying matrix B accordingly, matrix is brought in reduced row echelon form with according changes to matrix B. For bringing matrix to reduced row echelon form, see A.1.

At the end, the values of SLE variables are calculated and written into matrix C, as illustrated in Algorithm 5.

Algorithm 5 Getting results of SLE on CPU

```
1: for $i = 0$ to $matrix\_size - 1$ do
2:     $m\_C[i] \leftarrow m\_B[i]/m\_A[i][i]$
3: end for
```

3.2.1 Complexity analysis

The program consists of 3 loops: the out-most loop traverses through all rows of the matrix A consecutively choosing current pivot; second loop consecutively traverses through all rows under the currently chosen pivot; inner-most loop updates columns of the row chosen in the upper loop.

When the current pivot is the first entry in the matrix, the inner two loops, will run n times, when an outer loop, also runs n times, so, overall time complexity will be $O(n^3)$, where n is the number of variables of SLE.

Chapter 4

Algorithms on GPU

Basic Gauss-Jordan algorithm on CPU, processes each row, under pivot row, in successive manner, whereas, they could be updated concurrently with other rows. Each column of given row, could also be updated in parallel to other columns of that particular row. It leaves quite some room for parallelization and GPU could help achieving it.

4.1 CUDA kernel grid and blocks

Custom CUDA kernels could be implemented to run on GPU to modify all of the matrix rows and columns concurrently. To process matrix, 2 dimensional blocks will be used: with dimension X and Y. As, it is possible to allocate only 1024 threads per block, both dimensions X and Y of kernel block would be of size $\sqrt{1024} = 32$.

As CUDA kernel blocks are processed in a grid, for blocks to run concurrently, grid should be able to cover all of the matrix with its blocks, so, assuming the matrix height and width is N and as grid is 2 dimensional(X and Y), for each dimension, there will be $\frac{N}{32}$ blocks, if $N \bmod 32 = 0$ and $\frac{N}{32} + 1$ blocks, if $N \bmod 32 \neq 0$.

4.2 Basic Gauss-Jordan implementation on GPU

To process each row under pivot row, ratio of pivot entry with every entry below the pivot entry in pivot column must be calculated and stored in some array, such array will be called as array of multiplicatives. To get the values of ratios in parallel, custom CUDA kernel could be used. Algorithm 6 illustrates kernel for calculation of multiplicative values for given pivot, which will be called with dimension of grid and blocks as described in Section 4.1:

1. Position of entry of the pivot column under pivot entry is established(line 1).
2. All threads that got position of the entry that goes out of the matrix boundaries, will not continue calculation(lines 2-4).
3. Multiplicative values is calculated for given entry of the matrix in the pivot column under the pivot row(line 5).

Algorithm 6 Kernel for calculation of multiplicative values in top-down fashion

```
1: $pos\_y \leftarrow blockIdx.y \cdot blockDim.y + threadIdx.y + pivot\_id + 1$
2: if $pos\_y \geq matrix\_size$ then
3:     $return$
4: end if
5: $mult[pos\_y] \leftarrow matrix[pos\_y][pivot\_id]/matrix[pivot\_id][pivot\_id]$
```

For down-top version of Algorithm 6, see B.1.

Algorithm 7 Kernel for updating matrix rows under pivot row

```
1: $pos\_x \leftarrow blockIdx.x \cdot blockDim.x + threadIdx.x + pivot\_id$
2: $pos\_y \leftarrow blockIdx.y \cdot blockDim.y + threadIdx.y + pivot\_id + 1$
3: $mem\_id \leftarrow threadIdx.y$
4: if $pos\_y \geq matrix\_size \ \vee \ pos\_x \geq matrix\_size$ then
5:     $return$
6: end if
7: $extern\ \_\_shared\_\_\ arr[]$
8: if $threadIdx.x = 0$ then
9:     $arr[id] \leftarrow mult[pos\_y]$
10: end if
11: $\_\_syncthreads()$
12: if $pos\_x = pivot\_id$ then
13:     $m\_A[pos\_y][pos\_x] \leftarrow 0$
14:     $m\_B[pos\_y] \leftarrow m\_B[pos\_y] - arr[id] \cdot m\_B[pivot\_id]$
15: else
16:     $m\_A[pos\_y][pos\_x] \leftarrow m\_A[pos\_y][pos\_x] - arr[id] \cdot m\_A[pivot\_id][pos\_x]$
17: end if
```

After calculation of array of multiplicatives, rows, under the pivot row, must me updated. CUDA kernel for such purpose is illustrated in Algorithm 7 with the same dimensions of grid and blocks as described in Section 4.1:

1. Absolute positions of X and Y of current entry processed by the thread is calculated(lines 1-2).

2. Id of the shared memory array that will be used for accessing shared memory array(line 3).

3. Check, if current thread processes entry of the matrix that is out of bound of the matrix(lines 4-6).

4. Initialize shared memory array, copy to it values from multiplicatives array, as value of entry from multiplicative array will be used throughout the given row of the matrix that will be updated, and synchronize threads to wait(done via $\_\_syncthreads()$ function), until all threads finish initializing shared memory(lines 7-11).

5. Update the left-hand side of the SLE(lines 13, 16) and right-hand side of the SLE(line 14).

Down-top version of kernel in Algorithm 7 is different only with the fact that it contains calculation of SLE variables' values and absolute positions of X and Y of a matrix entry that will be processed via thread are different.

Algorithm 8 Calculation of SLE variables' values

```
1: $pos\_x \leftarrow blockIdx.x \cdot blockDim.x + threadIdx.x$
2: $pos\_y \leftarrow blockIdx.y \cdot blockDim.y + threadIdx.y$
3: ...
4: if $pos\_y = pivot\_id - 1 \ \wedge \ pos\_x = pivot\_id$ then
5:     $m\_C[pos\_y + 1] \leftarrow m\_B[pos\_y + 1]/m\_A[pos\_y + 1][pos\_x]$
6:     $m\_C[0] \leftarrow m\_B[0]/m\_A[0]$
7: end if
```

Algorithm 8 illustrates down-top processing of matrix:

1. Absolute positions of X and Y of matrix entry that must be processed via current thread is different that of a position in a kernel in top-down version, as in down-top version, rows, above the pivot row are updated(lines 1-2).

2. ... means copy of the lines 3-17 of Algorithm 7.

3. It is only possible to calculate value of the SLE variable which is at the position of the current pivot entry, as the current pivot row will not be changed, that is why, when current kernel thread, processes entry which is the next above the pivot entry in the pivot column, then value of the SLE variable associated with the currently chosen pivot entry, will be calculated.

   As the first row of the matrix will not be chosen as a pivot row, as it is the last row that is processed in the down-top version of the algorithm, SLE variable, associated with the pivot entry of the first row should also be calculated, whenever, any other SLE variable's value is calculated(lines 4-6).

The steps with finding array of multiplicatives and further updating of the matrix is done at each iteration when the pivot entry is chosen, both in top-down and down-top fashion. Algorithm 9 illustrates the whole pseudo-code for solving SLE on GPU.

Algorithm 9 Gauss-Jordan method on GPU

```
1: for $pivot\_id = 0$ to $matrix\_size - 2$ do
2:     $get\_mult\_array\_top\_down(m\_A,\ mult,\ pivot\_id,\ matrix\_size)$
3:     $update\_matrix\_top\_down(m\_A,\ m\_B,\ pivot\_id,\ matrix\_size)$
4: end for
5: for $pivot\_id = matrix\_size - 1$ downto $0$ do
6:     $get\_mult\_array\_down\_top(m\_A,\ mult,\ pivot\_id,\ matrix\_size)$
7:     $update\_matrix\_down\_top(m\_A,\ m\_B,\ m\_C,\ pivot\_id,\ matrix\_size)$
8: end for
```

4.2.1 Testing of basic Gauss-Jordan implementation on GPU

To test Gauss-Jordan method algorithm on GPU, algorithm on CPU will be used as a reference solution.

Testing environment:

- GPU name: NVIDIA GeForce RTX 2080 Ti.
- Size of total global memory (in bytes): 11554717696.
- Size of shared memory per block (in bytes): 49152.

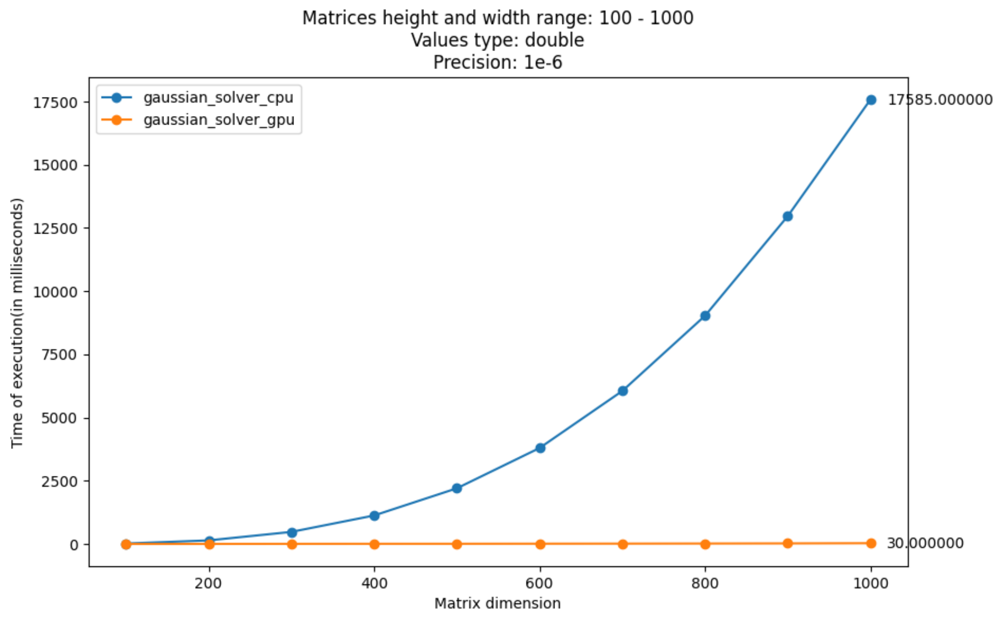
Tests will consist of comparison of time of execution of solvers, as well as, correctness of solved SLE variables, i.e. *matrix* C(see Section 2.1), compared to values of SLE variables solved via CPU solver: Gauss-Jordan solver on CPU will be considered 100% correct, as it is taken as a reference solution.

Testing will be made on different matrix dimensions and correctness of calculation will be calculated as an average correctness, in percentage, for a given solver throughout different matrix sizes ranging from 100-1000 with step 100, i.e. set of matrix sizes that consists of 10 matrices: $matrix\ size\ set = \{100, 200, 300, ..., 1000\}$:

$$average\_correctness(solver) = \frac{\sum_{j=100}^{1000} correctness(solver,\ matrix_j)}{|matrix\ size\ set|} \tag{1}$$

Both types of tests, i.e. the time of execution and average correctness of calculations, will be performed on double and single precision types.

As it is illustrated in Figure 3 with both, double and single precision types, Gauss-Jordan implementation on GPU is much faster than Gauss-Jordan implementation on CPU.



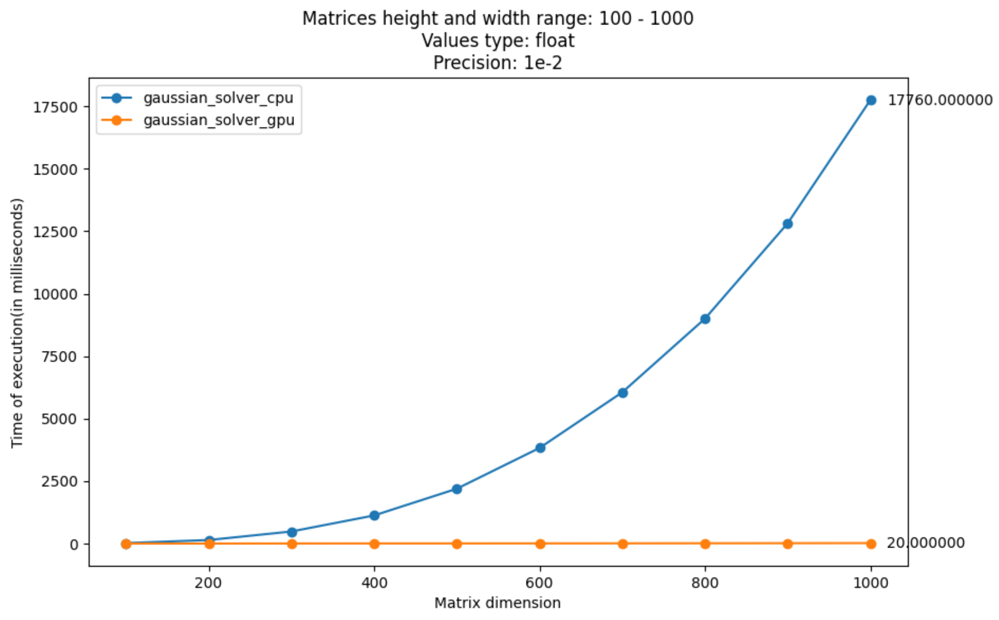


Figure 3: Time of execution with double and single precision values

As of correctness of calculations, illustrated in Figure 4, Gauss-Jordan implementation on GPU gives roughly the same results as a Gauss-Jordan implementation on CPU with double precision, although, with single precision, algorithm on GPU sometimes gives different results compared to algorithm on CPU and average correctness of calculations drops.

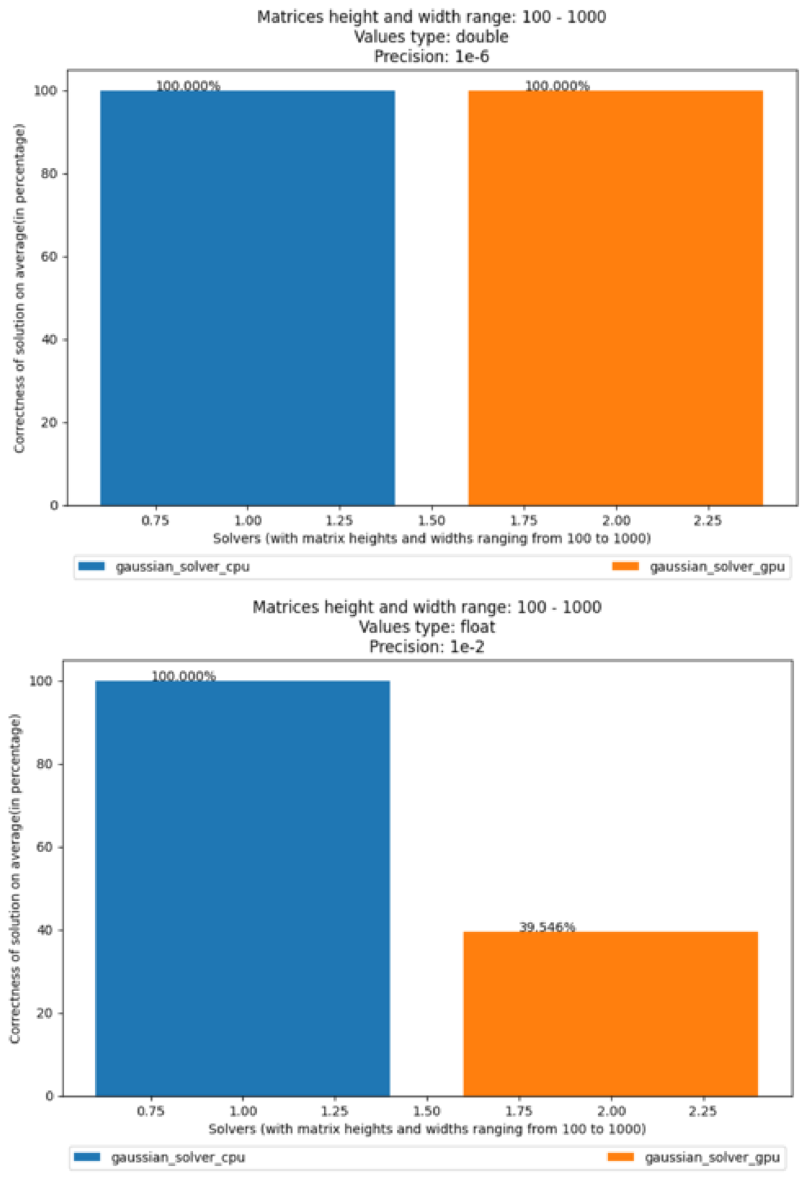


Figure 4: Average correctness of solvers with double and single precision values

4.2.1.1 Matrix entry values generation

Values of matrix entries are generated via mt19937 random number generator, used in C++, see [9] with uniform distribution of numbers between -0.5 and 0.5. Listing 4.1 illustrates code in C++ for generating n random number and saving them in an array to use as matrix values.

```
std::random_device random_dev;
std::mt19937 generator(random_dev());
std::uniform_real_distribution<double> dist(-0.5, 0.5);

for(int i = 0; i < n; i++)
    array[i] = (double) dist(generator);
```

Listing 4.1: Generation of random numbers via uniform distribution

4.2.1.2 Comparison of values

Comparison is made on calculated values of SLE variables, i.e. *matrix* C(see Section 2.1) of given solver, with the results(*matrix* C) obtained by reference solution.

Algorithm 10 illustrates a function of comparison of two values that will be used to compare results of different solvers with a reference solution of SLE that will be computed via reference solver:

- z_1 is a number from the *matrix* C of the reference solver and z_2 is a number from the *matrix* C of the solver that is tested. See Section 2.1 for explanation of *matrix* C.
- *precision* is chosen as 10^{-6} for double precision values and 10^{-2} for single precision values.
- Function $abs()$ return absolute value of a number and $min()$ returns minimum between 2 given numbers.

Algorithm 10 Comparison of two values

```
1: if $abs(z_1 - z_2) \leq precision \cdot abs(min(z_1, z_2))$ then
2:     $return\ true$
3: end if
4: $return\ false$
```

4.2.2 Problem with basic Gaussian elimination

Problem with Gaussian elimination consists of round-off errors during calculations, see [10].

As in Gaussian elimination algorithm, updating a given entry of the matrix, for example, entry in row y and column x with given current pivot row at index i, consists of 2 parts:

1. Calculation of ratio: $mult \leftarrow matrix[y][i]/matrix[i][i]$, as illustrated in line 3 and 11 of Algorithm 1.

2. Updating given entry with regards to obtained ratio: $matrix[y][x] \leftarrow matrix[y][x] - mult \cdot matrix[i][x]$ as illustrated in lines 5 and 13 of Algorithm 1.

When 2nd step of updating a given entry is done, the new value of entry $matrix[y][x]$ depends on the ratio $mult$, as if that ratio is evaluated in a large value, the multiplication $mult \cdot matrix[i][x]$ will be quite large too, in that case the whole subtraction $matrix[y][x] - mult \cdot matrix[i][x]$ will evaluate in a small number, in some cases, depending on a ratio $mult$, that subtraction operation could evaluate in $\epsilon : \epsilon \to 0$, i.e. to a very small number and given particular precision used in a computer, calculation of such a small number could lead to a truncature effect or a round-off error, that could even, potentially, lead to 0. This round-off error would lead to numerical instability caused by standard Gaussian elimination algorithm.

4.3 Partial pivoting

To overcome round-off errors, described in Section 4.2.2, partial pivoting strategy could be used.

As value of the ratio $mult$, could get large given a current pivot that consists of a small value, that would lead to a round-off error, another pivot, with maximum value in the pivot column, under pivot row, must be chosen, so that the pivot, used in the ratio, would be of a large value, that would lead to a ratio $mult$ to be of a small value that could solve a resulting round-off error.

When a value in the pivot column under pivot row is chosen as a new pivot, current pivot row and row of the new chosen pivot with larger value, must be swapped. To read more about partial pivoting, see [11].

Naive way of finding entry in pivot column with maximum value(to be chosen as a pivot), would be of $O(n)$ complexity: traverse the whole pivot column below pivot row and iteratively compare values with each other. That can be achieved via one loop. As in Partial Pivoting strategy, an entry, in the pivot column, with maximum is chosen at each iteration over the rows of Gauss-Jordan algorithm, this added complexity will be an overhead. A better approach than a naive iteration over the pivot column must be chosen to find an entry with maximum value.

4.4 Parallel Reduction

Parallel reduction algorithms are such algorithms that given an array of elements, produce a single result. As examples of such algorithms could serve: finding sum of elements in an array, finding minimum/maximum element in an array. See [12] for detailed explanation of parallel reduction algorithms.

We are interested in finding a maximum element in an array, where pivot column, could be referred as an array, maximum element of which, will be found. With GPU and CUDA kernels, finding maximum via parallel reduction algorithm could be achieved in $O(log(n))$ time complexity:

1. Start with an array of pivot column entries below and including pivot row, as we should also take into account current pivot, in case if it is an entry with maximum value in the pivot column.

   An array with pivot column values will start from current pivot row and will be of size $N = matrix\_size - pivot\_id$, where $matrix\_size$ is the size height and width of the matrix and $pivot\_id$ is an index of current pivot row.

   Besides an array with pivot column entries, an additional array of indices will also be needed, as each entry of that additional array of indices, will contain index of an entry with maximum value, compared to another entry, as a result of $max()$ function, illustrated in Figure 5.

   For the first run of the algorithm, indices of entries from pivot column will be copied to array of indices and in further steps of an algorithm, only an array of indices will be used to refer to the pivot column entries.

2. Parallel reduction kernel should be called that will find an entry with maximum value between 2 entries, i.e. for each kernel call, every thread of that kernel, will compare 2 entries, that is why, number of threads needed for comparison is half the number of elements in the array, as illustrated in Equation 3, where T is number of threads:

$$T = \begin{cases} \frac{N}{2}, & N \bmod 2 = 0 \\ \frac{N}{2} + 1, & N \bmod 2 \neq 0 \end{cases} \tag{3}$$

3. Each thread will find a maximum entry between 2 entries, so, for each given entry, its mirror entry is found and only then, thread compares given entry with its mirror entry.

   To calculate mirror entry, first, starting position of threads, must be calculated, i.e. the top entry of the green entries illustrated in Figure 5. Equation 4 illustrates the calculation of starting position of threads.

$$start\_pos = \begin{cases} matrix\_size - \frac{N}{2}, & N \bmod 2 = 0 \\ matrix\_size - \frac{N}{2} - 1, & N \bmod 2 \neq 0 \end{cases} \tag{4}$$

After starting position of threads is calculated, current thread, can calculate an id of mirroring entry given current entry of an array. Equation 5 illustrates calculation of an index of mirroring entry, given and index of current entry($entry\_id$).

$$mirror\_id = \begin{cases} start\_pos - (entry\_id - start\_pos) - 1, & N \bmod 2 = 0 \\ start\_pos - (entry\_id - start\_pos), & N \bmod 2 \neq 0 \end{cases} \tag{5}$$

After mirroring entry to the given entry is calculated, values under given entry and its mirror entry are compared and an index of an entry with greater value is written into array of indices into the position of a current entry, as illustrated in Equation 6, where *array* is an array with values of pivot column entries that will be compared between each other.

$$indices[entry\_id] = \begin{cases} mirror\_id, & array[indices[mirror\_id]] > array[indices[entry\_id]] \\ entry\_id, & array[indices[mirror\_id]] \leq array[indices[entry\_id]] \end{cases} \tag{6}$$

4. N(number of elements to be compared) will be divided by 2, i.e. $N = \frac{N}{2}$, as each thread compares 2 entries at once, and, if $N > 1$, steps 1-3 are repeated.

As, in step 4 of parallel reduction algorithm, number of compared values is divided by 2 during each iteration, algorithm will run $\lceil log(n) \rceil$ times, where n is an initial number of elements in the pivot column below pivot row including the current pivot itself.

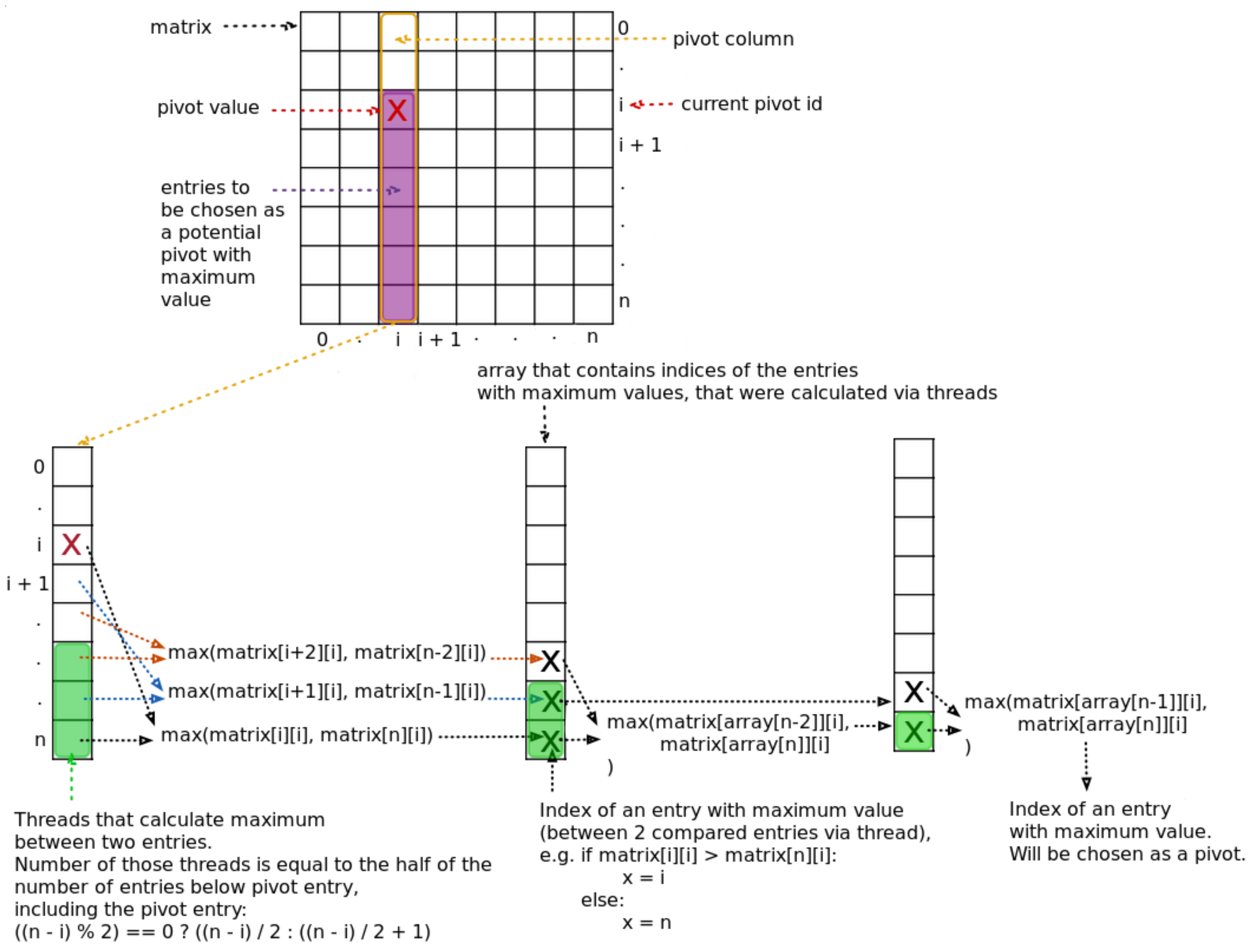

matrix
pivot column
pivot value
current pivot id
entries to be chosen as a potential pivot with maximum value
0
.
i
i + 1
.
.
.
n
0 . i i + 1 . . . n
array that contains indices of the entries with maximum values, that were calculated via threads
max(matrix[i+2][i], matrix[n-2][i])
max(matrix[i+1][i], matrix[n-1][i])
max(matrix[i][i], matrix[n][i])
max(matrix[array[n-2]][i], matrix[array[n]][i])
max(matrix[array[n-1]][i], matrix[array[n]][i])
Threads that calculate maximum between two entries.
Number of those threads is equal to the half of the number of entries below pivot entry, including the pivot entry:
((n - i) % 2) == 0 ? ((n - i) / 2 : ((n - i) / 2 + 1)
Index of an entry with maximum value (between 2 compared entries via thread),
e.g. if matrix[i][i] > matrix[n][i]:
x = i
else:
x = n
Index of an entry with maximum value.
Will be chosen as a pivot.

Figure 5: Parallel reduction algorithm

4.5 Full Partial Pivoting

In full partial pivoting strategy, the maximum pivot for given pivot entry is searched in an entire pivot column under pivot row. If an entry in the pivot column, below pivot row, with a value greater than current pivot, is found, an entire row of newly found pivot, with maximum value, compared to other pivot candidates, is swapped with current pivot entry. After rows have been swapped, algorithm should continue the same as in the Basic Gauss-Jordan algorithm on GPU:

1. Array of multiplicatives is calculated.
2. Matrix is updated.

The process of finding maximum pivot and processing matrix(finding array of multiplicatives and updating matrix entries below pivot row) must be done at every iteration over rows to bring matrix in row echelon form.

When matrix is being brought in reduced row echelon form, it is not possible to choose maximum pivot in a pivot column. The reason being that all entries, below the main diagonal, consist of 0 values.

$$a_j = \begin{cases} 0, & j < i \lor j > i \\ x, & x \neq 0 \land j = i \end{cases} \tag{7}$$

Equation 7 illustrates pivot row entry values with the pivot at the index i. If this pivot row gets swapped with another row at the index k, as this row is above pivot row(when bringing matrix to reduced row echelon form, algorithm runs in bottom-up fashion, so, pivot row is always below the rows that will be updated), i.e. $i > k$, pivot entry at the row k will be 0 and calculations won't succeed, as to calculate values of SLE variables, right-hand side gets divided by pivot entries of the right hand side and division by 0 will be encountered, as illustrated in Equation 8.

$$\frac{m\_B[k]}{m\_A[k][k]} = \varnothing : m\_B[k] \neq 0 \land m\_A[k][k] = 0 \tag{8}$$

For the reason, indicated above, pivot with maximum value will be chosen only when matrix is brought in row echelon form, i.e. the first part of the Gauss-Jordan algorithm, whereas, the second part, when matrix is brought into reduced row echelon form, will be left the same, as in Basic Gauss-Jordan algorithm on GPU.

At each iteration over matrix rows, to choose a pivot with greatest value in the pivot column, parallel reduction algorithm will be used.

4.5.1 Parallel Reduction Kernel

As explained in Section 4.4, parallel reduction algorithm runs only $\lceil log(n) \rceil$ times, where n is the number of elements in a pivot column to be compared with each other. Algorithm 11 illustrated the preamble of calling parallel reduction kernel:

1. Number of entries of the pivot column below the pivot row including the pivot entry is calculated(line 1).
2. Array of indices is instantiated(line 2): indices of pivot column entries below pivot row including pivot row is copied into array of indices. Arguments are:
    - *array_indices* — array, where indices of pivot column entries will be copied to.
    - *matrix_size* — height and width of the matrix.
    - *pivot_id* — current pivot index.
3. Parallel reduction kernel is executed with arguments(line 4):
    - *m_A* — left-hand side matrix.
    - *array_indices*
    - *partial_pivot_id* — index of an entry with the maximum value(result of parallel reduction algorithm).
    - *elements_count*
    - *pivot_id*
    - *matrix_size*
4. Elements count is divided by 2 and iteration continues, if elements count is bigger than 1(lines 5-9).

Algorithm 11 Preamble for parallel reduction kernel

```
 1: $elements\_count \leftarrow matrix\_size - pivot\_id$
 2: $create\_array\_of\_indices(...)$
 3: while $elements\_count > 1$ do
 4:     $parallel\_reduction\_kernel(...)$
 5:     if $elements\_count \bmod 2 = 0$ then
 6:         $elements\_count \leftarrow elements\_count/2$
 7:     else
 8:         $elements\_count \leftarrow elements\_count/2 + 1$
 9:     end if
10: end while
```

Algorithm 12 illustrates a kernel for copying indices of pivot column entries into an array of indices:

1. Absolute Y position of a matrix entry that will be processed by current thread is calculated(line 1).
2. Check if calculated position of an entry is out of bounds of a matrix(lines 2-4).
3. Copy an index of current entry that into array of indices(line 5).

Algorithm 12 Kernel for copying pivot column entry's indices into array of indices

```
1: $pos\_y \leftarrow blockIdx.y \cdot blockDim.y + threadIdx.y + pivot\_id$
2: if $pos\_y \geq matrix\_size$ then
3:     $return$
4: end if
5: $array\_indices[pos\_y] \leftarrow pos\_y$
```

Algorithm 13 illustrates parallel reduction kernel itself:

1. Starting position of threads is calculated(lines 1-5). As the number of needed threads is half of the number of elements in an array, threads will execute on second half of an array of indices(lower part of the pivot column, see green entries, that indicate entries where threads run, illustrated in Figure 5).
2. Absolute Y position of current entry that current thread processes is calculated(line 6).
3. If current thread processes a pivot column entry that is out of boundaries of a matrix, thread stops execution(lines 7-9).
4. Index of mirror entry to the current entry, processed by current thread, is calculated(lines 10-14). Current entry's value will be compared with mirror entry's value.
5. In case if there are 2 entries to be compared(lines 15-19), compare their values and write an index of an entry with greatest value into *partial_pivot_id* that is a variable that stores result of parallel reduction algorithm — newly chosen pivot with greatest value that will be swapped with current pivot in the matrix(line 16). Comparison of values is done via function *abs_max_id*() that return an index of the greatest from two compared values.

6. If number of elements in the array of indices, that must be compared, is greater than 2, compare current entry's value with mirror entry's value and write an index of an entry with maximum value into an array of indices(lines 17-21).

Algorithm 13 Parallel reduction kernel for full pivoting

```
1: if $elements\_count \bmod 2 = 1$ then
2:     $threads\_start\_pos \leftarrow matrix\_size - elements\_count/2 - 1$
3: else
4:     $threads\_start\_pos \leftarrow matrix\_size - elements\_count/2$
5: end if
6: $pos\_y \leftarrow blockIdx.y \cdot blockDim.y + threadIdx.y + threads\_start\_pos$
7: if $pos\_y \geq matrix\_size$ then
8:     $return$
9: end if
10: if $elements\_count \bmod 2 = 0$ then
11:     $pos\_y\_mirror \leftarrow start\_pos - (pos\_y - threads\_start\_pos) - 1$
12: else
13:     $pos\_y\_mirror \leftarrow start\_pos - (pos\_y - threads\_start\_pos)$
14: end if
15: if $elements\_count = 2$ then
16:     $partial\_pivot\_id \leftarrow abs\_max\_id($
            $m\_A[array\_indices[pos\_y]][pivot\_id], array\_indices[pos\_y],$
            $m\_A[array\_indices[pos\_y\_mirror]][pivot\_id],$
            $array\_indices[pos\_y\_mirror]$
        $)$
17: else
18:     if $elements\_count \bmod 2 \neq 1 \vee pos\_y \neq threads\_start\_pos$ then
19:         $array\_indices[pos\_y] \leftarrow abs\_max\_id($
                $m\_A[array\_indices[pos\_y]][pivot\_id], array_indices[pos\_y],$
                $m\_A[array\_indices[pos\_y\_mirror]][pivot\_id],$
                $array_indices[pos\_y\_mirror]$
            $)$
20:     end if
21: end if
```

4.6 Local Partial Pivoting

Local partial pivoting works the same way as full partial pivoting with only difference that, instead of finding an entry in the pivot column with the greatest value to swap with current(default) pivot, only limited number of entries in pivot column below pivot row will be processed. This will limit the number of entries that must be processed therefore will speed up calculations.

The whole algorithm is the same as in the full partial pivoting with the only difference being in parallel reduction algorithm:

1. Number of entries in line 1 of Algorithm 11 will be a predefined value: limited number of rows, below pivot row that will be processed.
2. If statement condition in line 2 of Algorithm 12 will be substituted with $pos\_y \geq matrix\_size \vee pos\_y \geq pivot\_id + elements\_count$.
3. Starting position of threads will be different in lines 1-5 of Algorithm 13, these lines will be substituted with $threads\_start\_pos \leftarrow pivot\_id + elements\_count/2$.
4. Absolute position Y of current thread to be processed will also be different, so, line 6 in Algorithm 13 will be substituted with $pos\_y \leftarrow blockIdx.y \cdot blockDim.y + threadIdx.y + threads\_start\_pos$.
5. If statement condition in line 7 of Algorithm 13 will be substituted with $pos\_y \geq matrix\_size \vee pos\_y \geq pivot\_id{+}elements\_count$, as any thread should also ensure that it doesn't process an entry out of boundaries of given limit on rows to be processed.

4.7 cuBLAS

cuBLAS is a library, written by NVIDIA developers(see [13]) that contains function for solving matrix in triangular form(row echelon form) with multiple right-hand sides, namely *cublasDtrsm*() for double precision type and *cublasStrsm*() for single precision type.

All cuBLAS functions work with matrices in column major order, see [14], so, before solving matrix with *cublasDtrsm*() or *cublasStrsm*(), matrix of coefficients, in row echelon form, should be converted to column major order.

Conversion of matrix in row major order to column major order could be implemented via custom CUDA kernel, as illustrated in Algorithm 14:

1. Absolute X and Y positions of current processed entry via current thread are calculated(lines 1-2).
2. Current thread stops execution, if currently processed entry is out of boundaries of the matrix or if it is a pivot entry, as pivot entries stays in its default entry after transposition(lines 3-5).

3. Positions X and Y of a transposed entries are calculated(lines 6-7).
4. Only threads that are executed above the matrix's main diagonal is needed for transposition. All the threads that process entries below the main diagonal will stop their execution(lines 8-10).
5. Thread swaps current entry with its transposed entry(lines 11-13).

Algorithm 14 Conversion of matrix to column major order

```
 1: $pos\_x \leftarrow blockIdx.x \cdot blockDim.x + threadIdx.x$
 2: $pos\_y \leftarrow blockIdx.y \cdot blockDim.y + threadIdx.y$
 3: if $pos\_x \geq matrix\_size \vee pos\_y \geq matrix\_size \vee pos\_x = pos\_y$ then
 4:     $return$
 5: end if
 6: $pos\_x\_transposed \leftarrow pos\_y$
 7: $pos\_y\_transposed \leftarrow pos\_x$
 8: if $pos\_x\_transposed < pos\_x$ then
 9:     $return$
10: end if
11: $tmp\_val \leftarrow matrix[pos\_y][pos\_x]$
12: $matrix[pos\_y][pos\_x] \leftarrow matrix[pos\_y\_transposed][pos\_x\_transposed]$
13: $matrix[pos\_y\_transposed][pos\_x\_transposed] \leftarrow tmp\_val$
```

Algorithm with usage of cuBLAS consists of bringing matrix in row echelon form(basic Gaussian elimination) when algorithm runs in top-down fashion, but, instead of down-top part of Gauss-Jordan algorithm, matrix will be converted to column major order and processed via *cublasDtrsm*() or *cublasStrsm*() for double or single precision types correspondingly.

4.8 cuSOLVE

cuSOLVE is a library, written by NVIDIA developers(see [15]), which contains functions for LU decomposition(see [16]) of dense matrices.

As an additional solver, cuSOLVE solver will. Namely, the LU decomposition for dense matrices implemented via cuSolverDN functions will be used as a cuSOLVE solver.

As LU decomposition from cuSolverDN functions is implemented via NVIDIA developers, it is quite efficient in time consuming for solving SLEs.

Chapter 5

Testing and evaluation of results

For testing, all the discussed solvers on GPU will be used, with the total of 7 solvers:

1. cuSOLVE solver.
2. Basic Gauss-Jordan algorithm.
3. Basic Gaussian algorithm with cuBLAS.
4. Full Partial Pivoting Gauss-Jordan algorithm.
5. Full Partial Pivoting Gaussian algorithm with cuBLAS.
6. Local Partial Pivoting Gauss-Jordan algorithm.
7. Local Partial Pivoting Gaussian algorithm with cuBLAS.

5.1 Reference solution

Full partial pivoting implementation of Gauss-Jordan algorithm on CPU with long double type of values will be used as a reference solver, as it is implemented with long double(extended double precision) type that makes it more precise than other algorithms with double(double precision) and float(single precision) types. Results of all solvers on GPU will be compared with reference solver on CPU for establishing correctness of calculations of those solvers on GPU.

5.2 GPU for testing

GPU, used for testing, is the same as it was for testing of Basic Gauss-Jordan algorithm in Section 4.2.1.

5.3 Types of tests

As in Section 4.2.1, 2 types of tests will be done: time of execution of each solver for given matrix size and average correctness of calculations. Tests will be done on values with double and single precision types.

Matrix sizes will range from 250–4000 with step 250, i.e. set of matrix sizes that consists of 16 matrices: $matrix\ size\ set = \{250, 500, 750, ..., 4000\}$.

Values, used for matrices, are generated randomly, as described in Section 4.2.1.1.

5.4 Local Partial Pivoting

For algorithms with local partial pivoting, 10 rows will be used for parallel reduction algorithm to choose an entry in the pivot column with largest value: 1 for the current pivot row and 9 more rows under pivot row.

5.5 Double precision

5.5.1 Time of execution

As it could be observed in Figure 6, Gauss-Jordan algorithms with partial pivoting are only slightly slower than basic Gauss-Jordan algorithm without partial pivoting.

As expected, Full Partial Pivoting is slower than Local Partial Pivoting, but, as parallel reduction algorithm, used in partial pivoting approaches, is only $O(log(n))$ time complexity, the difference is very subtle.

cuBLAS variations of partial pivoting strategies of Gaussian algorithm proved to be much faster than standard Gauss-Jordan algorithm with partial pivoting.

Reference solver on CPU graph is not included into the plot, as CPU is very slow compared to GPU for SLE solving, so reference solver would shift plot high enough and all GPU solver's graphs would collide with each other.

As to solve a SLE via Gaussian elimination algorithms, CUDA kernels are called for every pivot rows from host code, there will be a lot of host to device context switches which is a overhead. Because of abundance of context switches of Gaussian elimination algorithms, cuSOLVE is much faster than Gaussian elimination algorithms. One more reason for cuSOLVE solver to be much faster than Gauss-Jordan solvers is that LU decomposition is faster than Gauss-Jordan method.

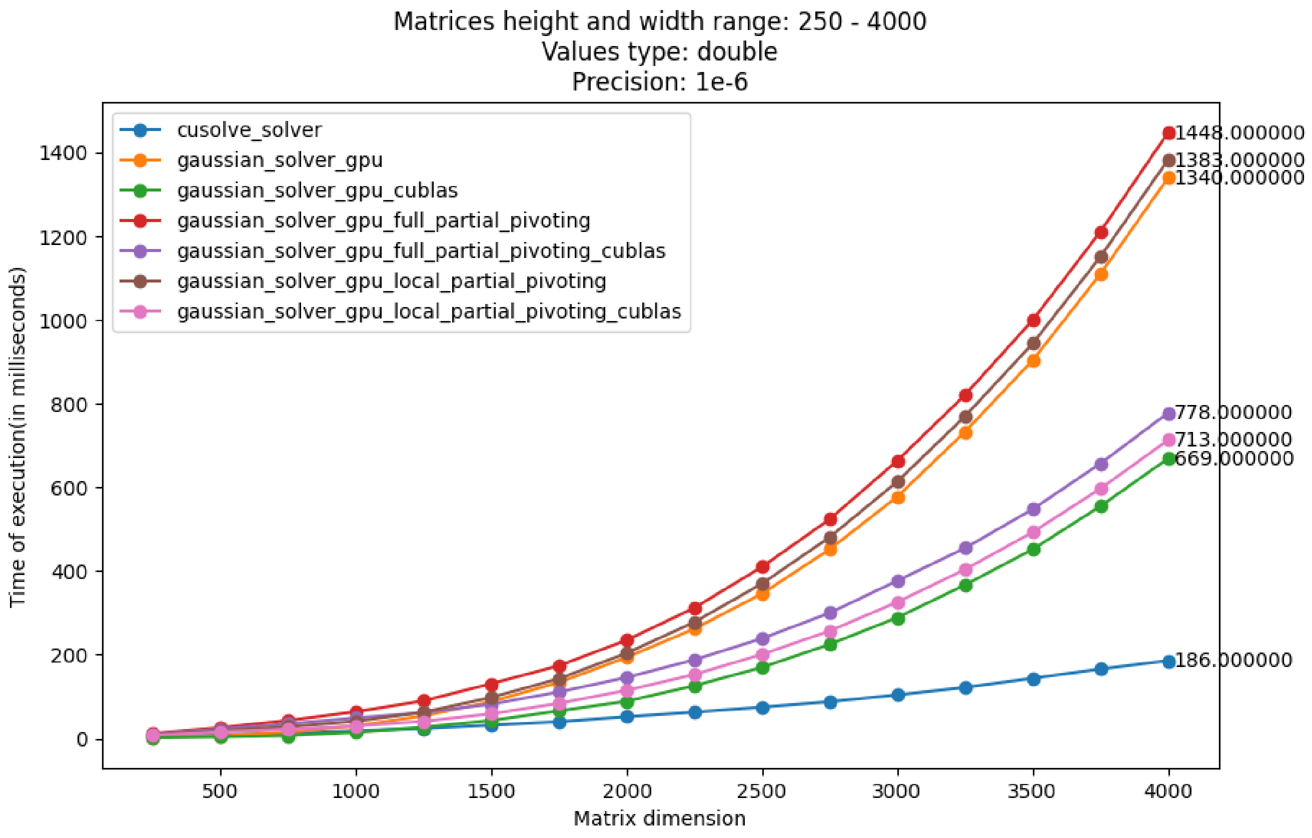


Figure 6: Time of execution with values of double precision type

5.5.2 Correctness of calculations

For establishing correctness of calculations, the same Algorithm 10 that was implemented in Section 4.2.1.2 is used, but with different precisions: 10^{-6}, 10^{-8}, 10^{-10}, 10^{-12}.

As it could be observed in Figure 7 and Figure 8, partial pivoting approaches are much more accurate than basic Gauss-Jordan implementation.

cuBLAS implementations of Gaussian algorithms of those partial pivoting strategies produce accuracy close to implementations of Gauss-Jordan algorithm with partial pivoting strategies.

With precision increasing from 10^{-6} to 10^{-12}, the difference between full partial pivoting strategy and cuSOLVE gets bigger in favor of full partial pivoting strategy. With the increase of precision, correctness of local partial pivoting gets close to basic Gauss-Jordan algorithm without any pivoting strategy, whereas correctness of full partial pivoting strategy remains high.

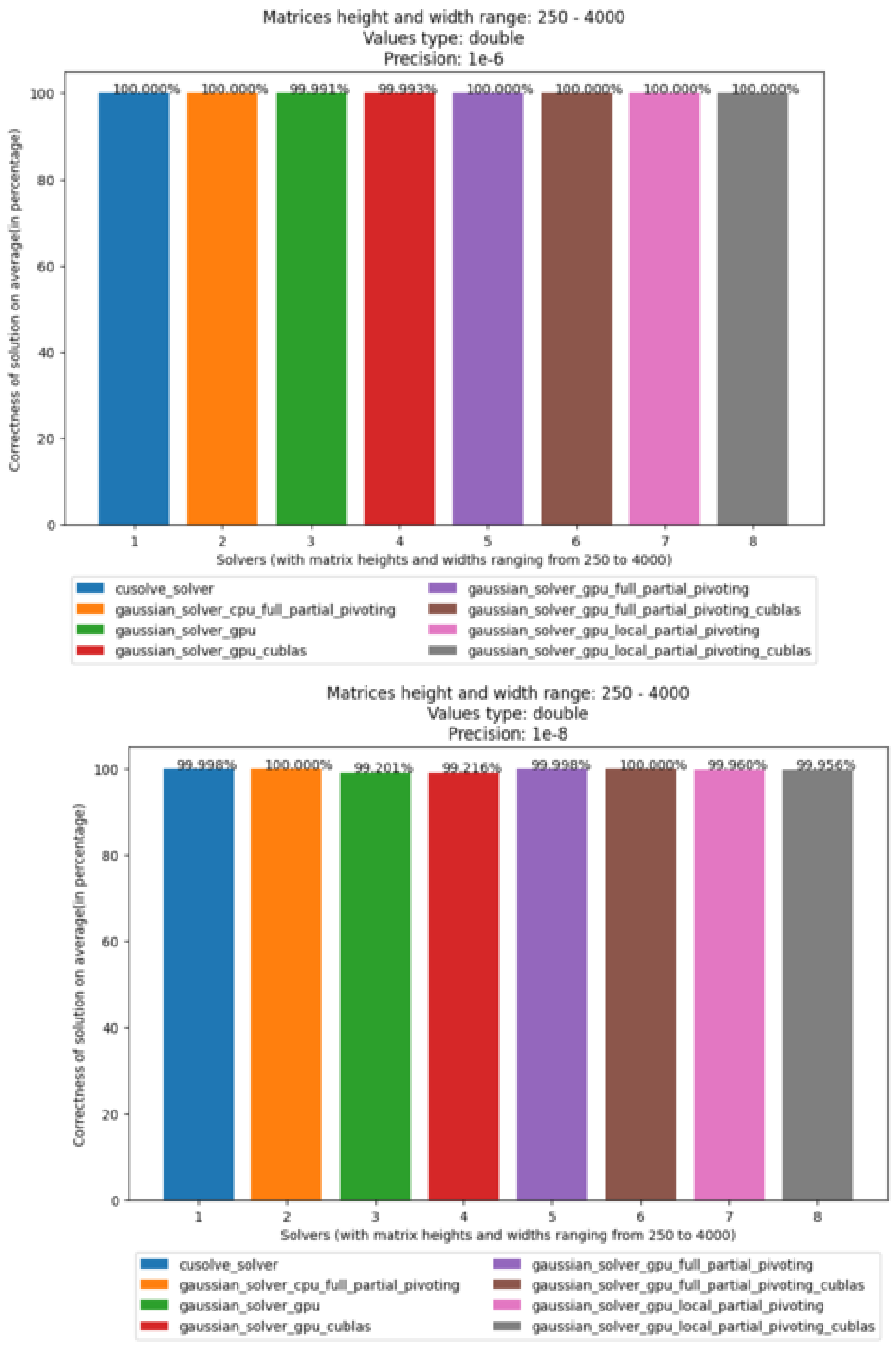


Figure 7: Average correctness of calculations with precisions 10^{-6} and 10^{-8}

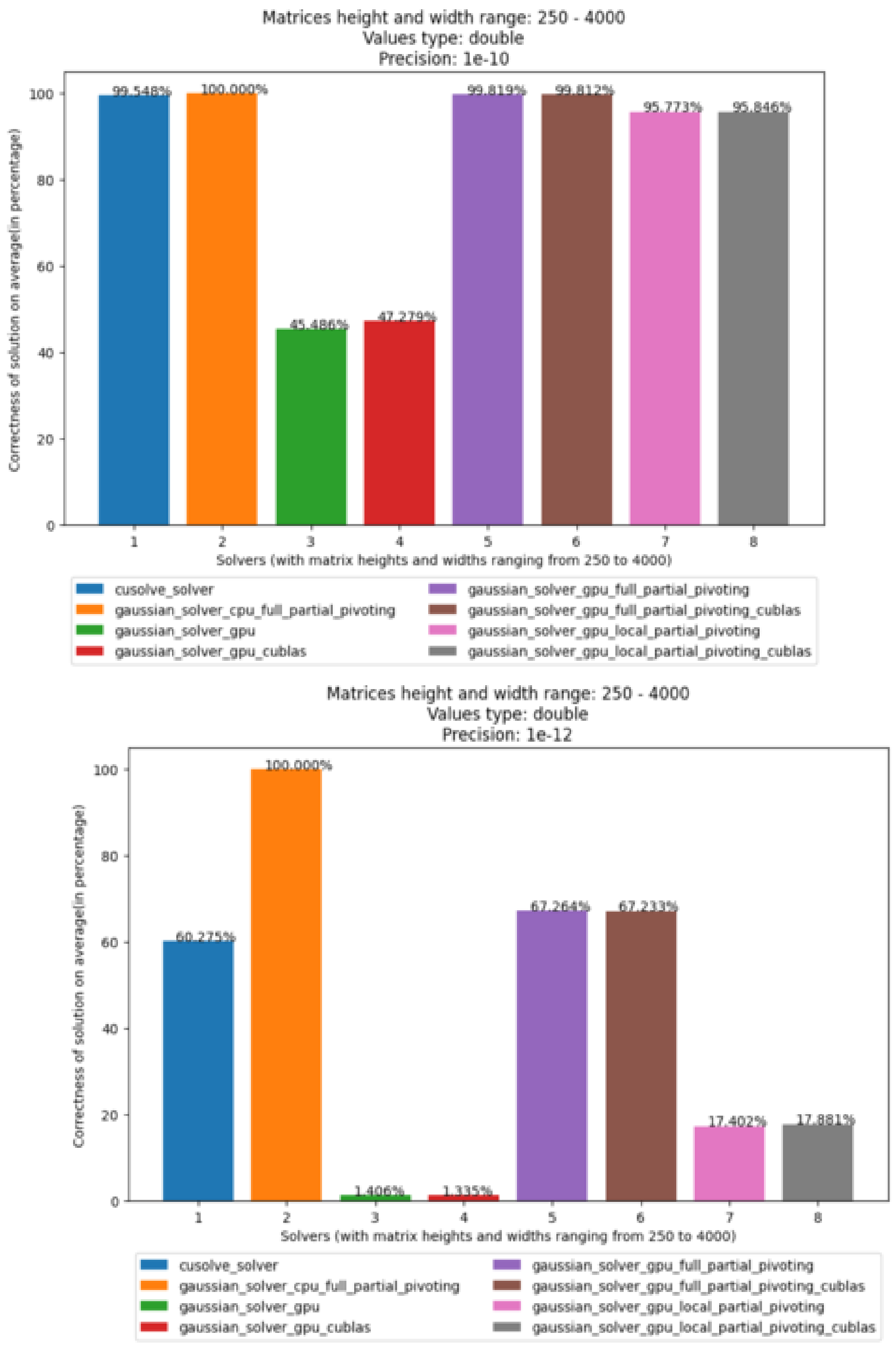


Figure 8: Average correctness of calculations with precisions 10^{-10} and 10^{-12}

5.6 Single precision

5.6.1 Time of execution

As illustrated in Figure 9, calculations with values of single precision type are much faster, but overall picture remains the same as it was for values of double precision type.

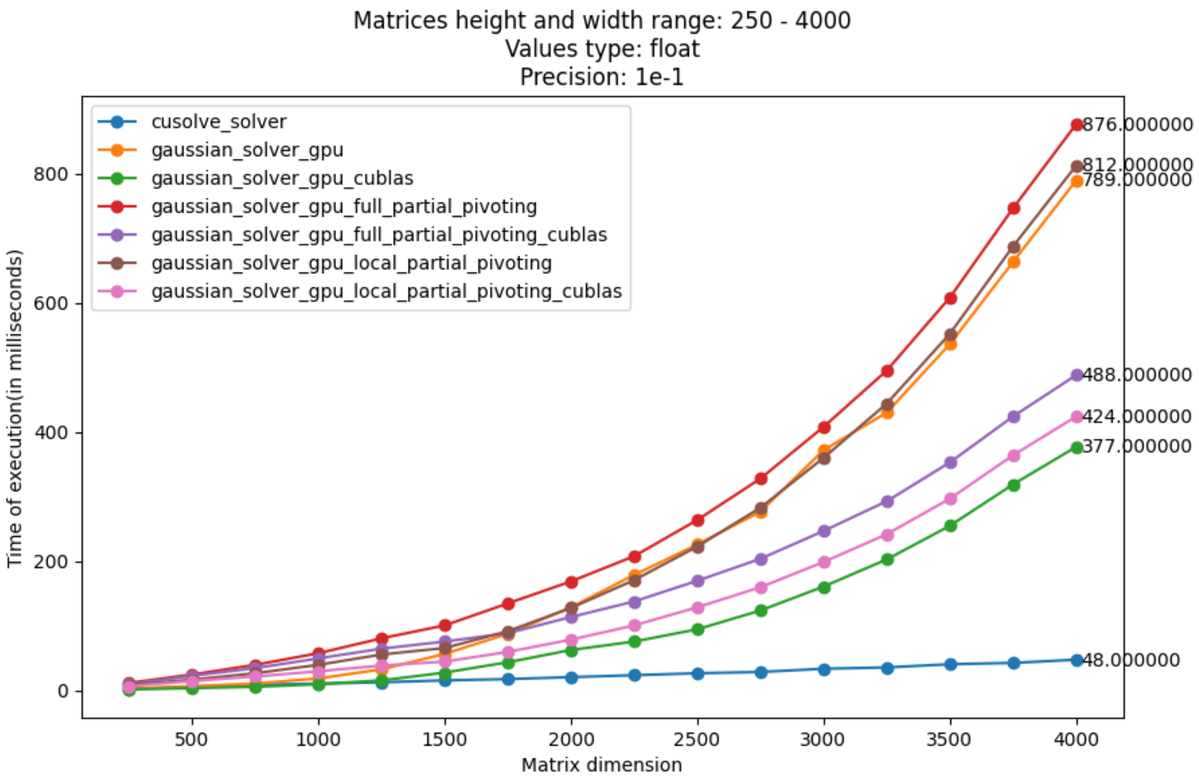
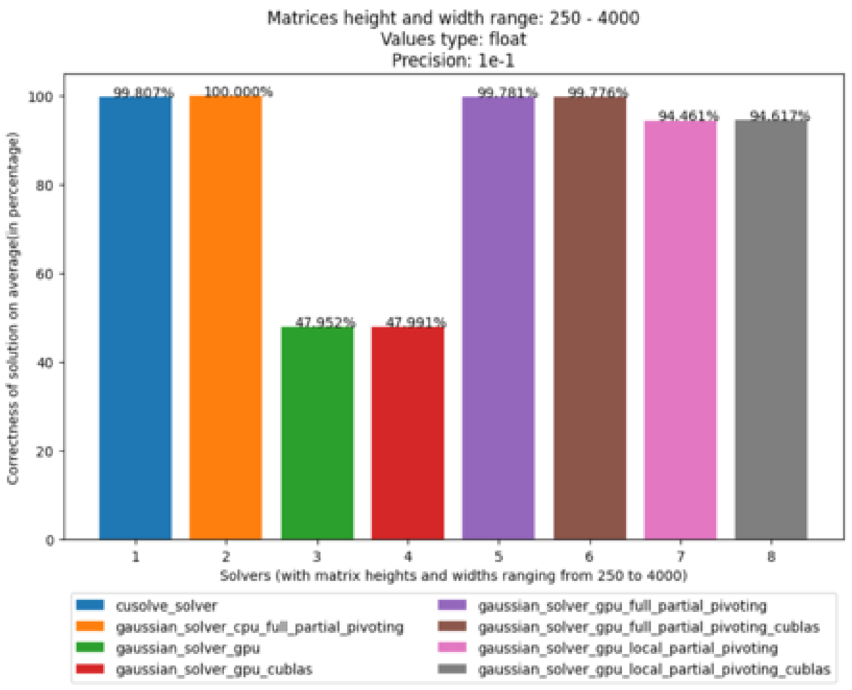


Figure 9: Time of execution with values of single precision type

5.6.2 Correctness of calculations

For values of single precision type Algorithm 10 from Section 4.2.1.2 is used with precisions: 10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}.

Figure 10 and Figure 11 illustrate that overall picture stays the same as it was for testing with double precision: partial pivoting algorithms are still much more precise than basic Gauss-Jordan algorithm, but cuSOLVE and full partial pivoting strategy have quite similar correctness of calculations, although with precision 10^{-4} the overall correctness is quite low for all algorithms.



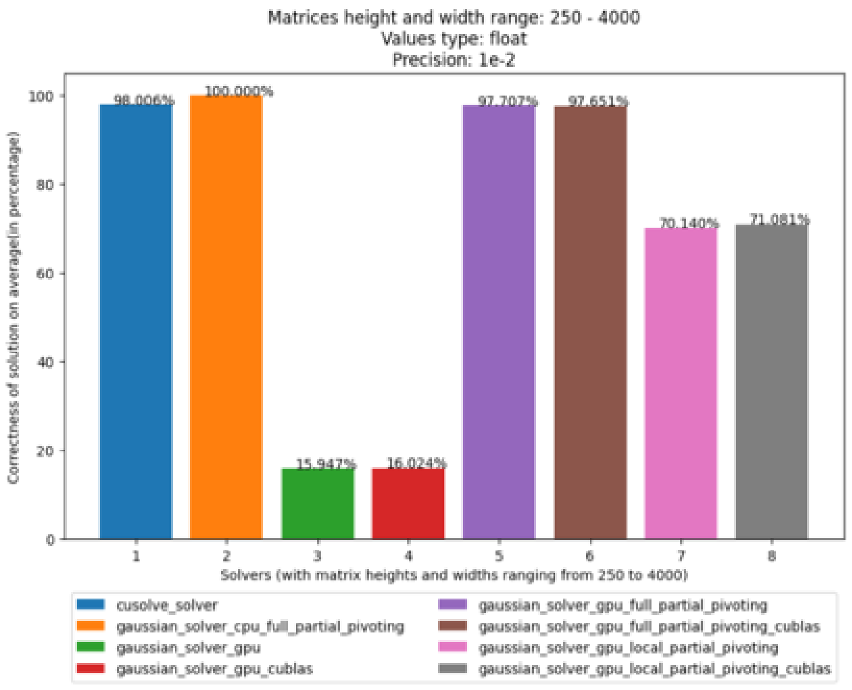
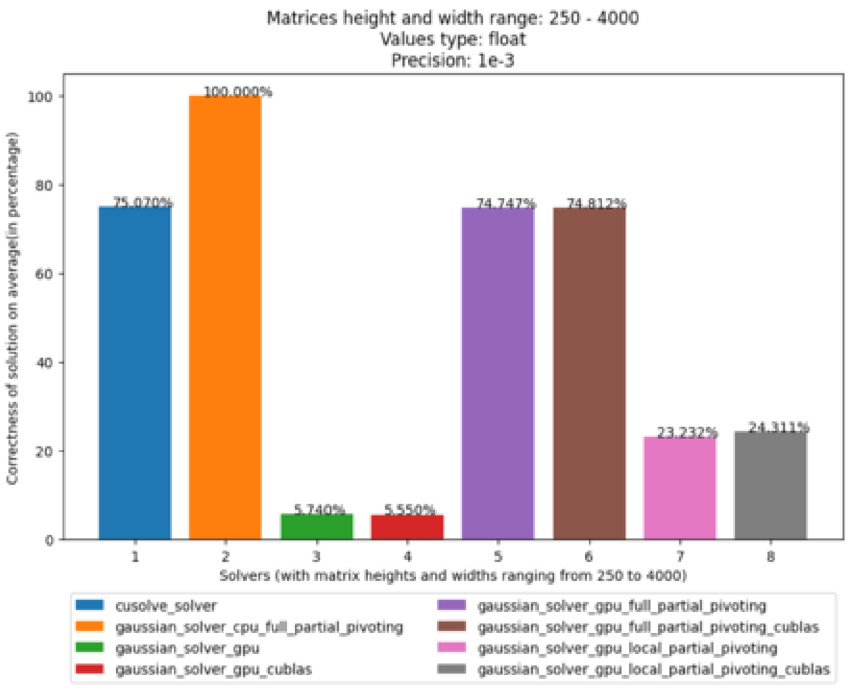


Figure 10: Average correctness of calculations with precisions 10^{-1} and 10^{-2}



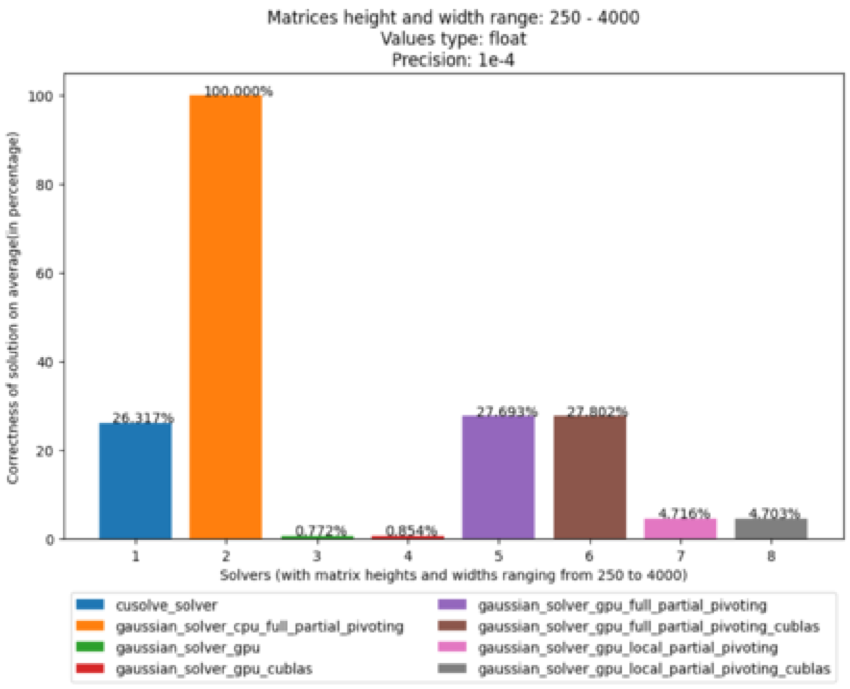


Figure 11: Average correctness of calculations with precisions 10^{-3} and 10^{-4}

Chapter 6

Conclusion

The goal of this work was to implement SLE solvers on GPU via various Gaussian elimination strategies, compare Gauss-Jordan implementation on GPU with CPU, study the difference of time and correctness of execution of different solvers.

NVIDIA's CUDA technology on GPUs was used to implement CUDA kernels that would run on GPU to achieve parallelism.

Basic Gauss-Jordan algorithm on GPU and CPU were implemented and results of GPU algorithm are compared with CPU algorithm. Tests showed that, implementation of Gauss-Jordan algorithm on GPU was much faster than on CPU that proved the use of high parallel environment for efficient solving of SLEs.

To prevent loss of precision(round off errors) during calculations, partial pivoting strategies were implemented, namely full partial pivoting and local partial pivoting. To find an entry in the pivot column with the largest value, parallel reduction algorithm for finding maximum in a given pivot column of the matrix was implemented.

cuBLAS library with $cublasDtrsm()$ function for double precision type and $cublasStrsm()$ function for single precision type were used to substitute down-top part of Gauss-Jordan algorithm. cuBLAS variation of Gaussian elimination algorithm was implemented on full and local partial pivoting algorithms.

During testing, it was established that due to parallel reduction algorithm being of $O(log(n))$ time complexity, differences in time of execution were quite subtle between basic algorithm implementation on GPU and partial pivoting variations of it, although correctness of calculation was much higher on partial pivoting strategies.

cuBLAS variation of Gaussian elimination algorithm with partial pivoting strategies proved to be close to 2 times faster than standard Gauss-Jordan algorithm with partial pivoting strategies and correctness of calculations with algorithms with cuBLAS with partial pivoting strategies was approximately the same as it was for partial pivoting algorithms without cuBLAS.

When it comes to correctness of calculations, full partial pivoting algorithms(both with cuBLAS and without it) proved to be a bit more precise than cuSOLVE algorithm. As testing was done with different precisions, with greater precision, difference in correctness of calculations between full partial pivoting strategies and cuSOLVE is bigger in favor of full partial pivoting strategy. So, given drawback in time of execution of full partial pivoting algorithms compared to cuSOLVE algorithm, full partial pivoting algorithms are more precise than cuSOLVE algorithm and as cuBLAS variation of full partial pivoting algorithm is nearly 2 times faster than an algorithm without cuBLAS, it makes full partial pivoting algorithm with cuBLAS the best solver on GPU when it comes to correctness of calculations.

Chapter 7

Possible improvements

As an improvement of partial pivoting strategy, complete pivoting strategy could be implemented, when not only rows get swapped, but also columns: first, maximum value throughout all columns of a given row is found, then it is swapped with pivot column entry of this row and only then maximum value in the pivot column is found for swapping with current pivot row. For more information about complete pivoting, see [17].

Bibliography

[1] Golub Gene, V. L. *Matrix Computations*. Johns Hopkins, third edition, ISBN 978-0-8018-5414-9.

[2] Henry Maltby, J. K., Shaurya Singh. Gauss-Jordan elimination. *Brilliant.org*, November 2021, [visited on 2021-11-19]. Available from: `https://brilliant.org/wiki/gaussian-elimination/`

[3] Howard Anton, C. R. *Elementary linear algebra : applications version*. John Wiley & Sons, third edition, ISBN 978-1-118-43441-3.

[4] Leon, S. *Linear Algebra with Applications*. Pearson, 8th edition, ISBN 978-0136009290.

[5] NVIDIA. CUDA C++ Programming Guide. October 2021, [visited on 2021-11-19]. Available from: `https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`

[6] Gupta, P. CUDA kernels. *NVIDIA Developer*, June 2020, [visited on 2021-11-19]. Available from: `https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/`

[7] NVIDIA. CUDA kernel image. *NVIDIA Developer Zone*, October 2021, [visited on 2021-11-19]. Available from: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/graphics/grid-of-thread-blocks.png`

[8] Harris, M. CUDA shared memory. *NVIDIA Developer Blog*, January 2013, [visited on 2021-11-19]. Available from: `https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/`

[9] Random numbers generator. [visited on 2021-11-22]. Available from: `https://www.cplusplus.com/reference/random/mt19937/`

[10] Weisstein, E. W. Roundoff Error. *MathWorld–A Wolfram Web Resource*, November 2021, [visited on 2021-11-19]. Available from: `https://mathworld.wolfram.com/RoundoffError.html`

[11] B.V., E. Partial pivoting. *ScienceDirect*, 2021, [visited on 2021-11-19]. Available from: `https://www.sciencedirect.com/topics/mathematics/partial-pivoting`

[12] Karikalan, S. Parallel Reduction with CUDA. *Medium.com*, December 2020. Available from: `https://shreeraman-ak.medium.com/parallel-reduction-with-cuda-d0ae10c1ae2c`

[13] NVIDIA. CUBLAS library. *NVIDIA Developer Zone*, October 2021, [visited on 2021-11-19]. Available from: `https://docs.nvidia.com/cuda/cublas/index.html`

[14] Knuth, D. E. *The Art of Computer Programming*. Addison-Wesley, first edition, ISBN 0-201-03801-3.

[15] NVIDIA. cuSOLVE library. October 2021, [visited on 2021-11-19]. Available from: `https://docs.nvidia.com/cuda/pdf/CUSOLVER_Library.pdf`

[16] West, M. LU Decomposition for Solving Linear Equations. February 2018, [visited on 2021-11-29]. Available from: `https://courses.physics.illinois.edu/cs357/sp2020/notes/ref-9-linsys.html`

[17] B.V., E. Complete pivoting. *ScienceDirect*, 2021, [visited on 2021-11-19]. Available from: `https://www.sciencedirect.com/topics/mathematics/complete-pivoting`

Appendix

A Gauss-Jordan method on CPU

A.1 Bringing matrix in reduced row echelon form

Algorithm 15 Down-top triangulation on CPU

```
1: for $i = matrix\_size - 1$ downto 0 do
2:     $pivot \leftarrow matrix\_A[i][i]$
3:     for $y = 0$ to $i - 1$ do
4:         $row\_pivot \leftarrow matrix\_A[y][i]$
5:         $mult \leftarrow row\_pivot/pivot$
6:         for $x = 0$ to $i$ do
7:             $m\_A[y][x] \leftarrow m\_A[y][x] - mult \cdot m\_A[i][x]$
8:         end for
9:         $m\_A[y][i] \leftarrow 0$
10:        $m\_B[y] \leftarrow m\_B[y] - mult \cdot m\_B[i]$
11:    end for
12: end for
```

B Gauss-Jordan implementation on GPU

B.1 Kernel for calculation of multiplicative values in top-down fashion

Algorithm 16 Kernel for calculation of multiplicative values in down-top fashion

```
1: $pos\_y \leftarrow blockIdx.y \cdot blockDim.y + threadIdx.y$
2: **if** $pos\_y \geq pivot\_id$ **then**
3:     $return$
4: **end if**
5: $mult[pos\_y] \leftarrow matrix[pos\_y][pivot\_id]/matrix[pivot\_id][pivot\_id]$
```

Acronyms

API Application Programming Interface. 7

CPU Central Processing Unit. 1, 7, 13, 16–18, 31, 32, 39

CUDA Compute Unified Device Architecture. 1, 7–9, 13, 14, 21, 28, 32, 39

GPGPU General Processing on Graphics Processing Unit. 7

GPU Graphics Processing Unit. 1, 7–9, 13, 16–18, 21, 24, 31, 32, 39, 40

SLE System of Linear Equations. 1, 3, 4, 10, 11, 15, 16, 19, 24, 29, 32, 39

Chapter 8

Contents of enclosed CD

```
 Thesis ................ the directory with latex source file and thesis pdf
 |__ thesis.pdf .................................. thesis in PDF format
 |__ thesis.tex .................................. latex file with thesis
 |__ Resources ....................... directory with images from thesis
|__ src ...................................... the directory of source codes
 |__ Doc ...................... doxygen documentation of the source code
 |__ Execution_Time ......................................... csv files
 |__ Graphs ............................ graphs generated from csv files
 |__ Main .................................... implementation of solvers
 |__ Makefile ............................................... makefile
 |__ README.md ........ readme with exlanation of makefile and directories
 |__ Scripts ...................... scripts used for generation of graphs
 |__ Test ......................... implementation of unit and main tests
```