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.

May 12, 2021

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on May 12, 2021 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology

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, 2021.

Abstrakt

V několika větách shrňte obsah a přínos této práce v českém jazyce.

Klíčová slova Gauss-Jordan, GPU, CUDA, Carcenacův prokládací algoritmus.

Abstract

Summarize the contents and contribution of your work in a few sentences in English language.

Keywords Gauss-Jordan, GPU, CUDA, Carcenac's striping algorithm.

Contents

List of Figures

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.

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.

Two implementations of Gauss-Jordan method will be implemented: one on CPU and another on GPU. The performance of implementation on CPU will be compared with performance of implementation on GPU.

The implementation on GPU, presented in this thesis, will be based on Carcenac's striping algorithm. But, to update matrix entries more efficiently, we will use custom CUDA kernels instead of CUBLAS library used in original Carcenac's algorithm.

CUDA streams will be covered and used for parallel stripe updating, so that each stripe is updated concurrently.

At the end, the presented implementation of Gauss-Jordan method on GPU will be compared to an already existing implementation of dense matrix solving on CUDA using *cuSOLVER* library using LU decomposition. Both implementations will be compared.

Terms and definitions

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

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

Could be converted to matrix with m rows and n columns, as illustrated below:

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

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 Section 20.

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 Section 21 for basic Gaussian elimination. See Section 22 for matrix in row echelon form.

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.

To bring matrix to row echelon form, multiplication of rows to constants with addition of rows, following with interchange of rows.

Gauss-Jordan algorithm brings matrix to its *reduced row echelon* form. See Section 23.

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.

Listing 1 illustrates a pseudocode for Gauss-Jordan method:

1. The matrix is traversed in top-to-down fashion, pivot is chosen and all the rows, below the pivot row, are updated(lines 1-7).
2. The matrix is traversed in down-to-top fashion, pivot is chosen and all the rows, above the pivot row, are updated(lines 8-15).

Listing 1: Gauss-Jordan algorithm

```
1  for(i = 0; i < matrix_size; i++) {
2    for(y = i + 1; y < matrix_size; y++) {
3        mult = matrix[i][i] / matrix[y][i];
4      for(x = i; x < matrix_size; x++)
5        matrix[y][x] = matrix[i][i] - mult * matrix[y][x];
6    }
7  }
8
9  for(i = matrix_size - 1; i > 0; i--) {
10   for(y = 0; y < i; y++) {
11       mult = matrix[i][i] / matrix[y][i];
12     for(x = 0; x < i + 1; x++)
13       matrix[y][x] = matrix[i][i] - mult * matrix[y][x];
14   }
15 }
```

Software aspects

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 are 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 Section 24.

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. To know more about computability, see Section 29.

All the code that is processed by CPU is a **host** code, whereas, code that is processed by GPU is called a **device** code.

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.

In code, kernel functions are written with **___global___** prefix.

Every kernel runs on separate CUDA core.

Listing 2: Kernel call

```
1 dim3 grid(10, 10), block(32, 32);
2 custom_kernel<<<grid, block>>>();
```

In Listing 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.

To learn more about CUDA kernels, see Section 25.

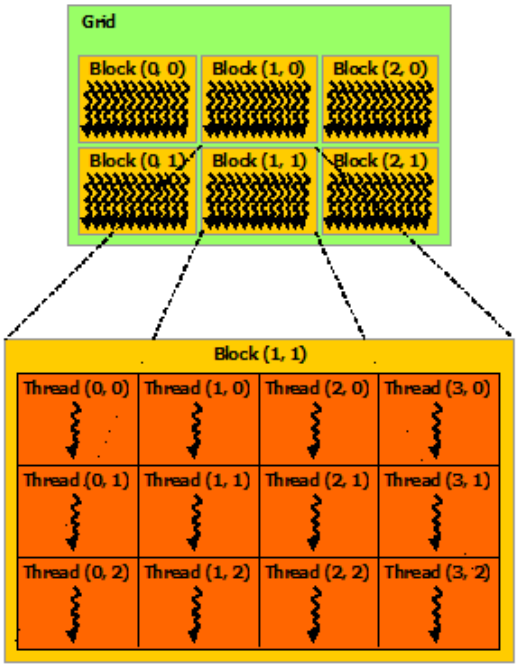

Figure 1: CUDA kernel

Figure 1 illustrates CUDA kernel. Image source: https://docs.nvidia.com/cuda/cuda-c-programming-guide/graphics/grid-of-thread-blocks.png.

1.2 Memory

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

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.

To learn more about global memory, see Section 27.

1.2.2 Shared memory

Every CUDA core has L1 cache. Memory could be allocated, referenced, and freed. That part of L1 cache, that is allocated by developer for use is known as **shared memory**. As, every core has its own shared memory, each kernel, given executed on one core, gets one shared memory address space, which is split between its blocks evenly. The GPU, used in this thesis(RTX 2080 Ti) has 64 KB of shared memory.

Listing 3: Shared Memory size declaration

```
1 int shared_memory_size = 2 * sizeof(double);
2 custom_kernel<<<grid, block, shared_memory_size>>>();
```

In Listing 3, shared memory of 16 B is allocated for each block of this kernel. Which means, that 64 KB of shared memory will be split to 100 address spaces(as this kernel has 100 blocks), which leads to $100 \cdot 16 = 1600$ B of memory being allocated in L1 cache as a shared memory.

Shared memory is roughly 100 times faster than global memory.

To learn more about global memory, see Section 28.

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.

1.3 Streams

All CUDA calls are synchronous, which means that host could call device code and not wait till the end of its execution, but each device call will be processed synchronously on GPU.

To make asynchronous device calls, those calls should be initiated on different CUDA streams.

Each stream is processed concurrently on device.

Each CUDA call that is processed on one stream will be executed synchronously.

By default, device code runs on stream 0, until other streams are created.

In each kernel call, 4-th argument is the stream, we want this kernel to execute on.

Listing 4: Creation usage and destruction of cuda streams

```
1 cudaStream_t stream;
2 cudaStreamCreate(&stream);
3 custom_kernel<<<grid, block, shared_memory_size, stream>>>();
4 ...
5 cudaStreamDestroy(stream);
```

Listing 4 shows how to create a stream, run kernel on it and destroy it.

1.3.1 Streams synchronization

To synchronize all streams, before host can move further, *cudaDeviceSynchrinize()* is used. To block host code until particular stream has finished all calls, *cudaStreamSynchronize(stream)* is used.

1.4 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()*.

2 Basic Gauss-Jordan algorithm implementation on CPU

2.1 Overview of the algorithm

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.

2.2 Algorithm

Listing 5: Implementation of Gauss-Jordan algorithm main

```
1  int matrix_size;
2  double * m_values = new double[matrix_size * matrix_size];
3
4  // matrix size is declared and matrix entries are populated here
5
6  // bring matrix to row echelon form
7  for(int i = 0; i < matrix_size; i++) {
8    for(int y = 0; y < matrix_size - i - 1; y++) {
9      double row_pivot = m_values[(y + i + 1) * matrix_size + i];
10     for(int x = 0; x < matrix_size - i; x++)
11       calc(m_values, row_pivot, x, y, i, matrix_size);
12   }
13 }
14
15 // bring matrix to reduced row echelon form
16 for(int i = matrix_size - 1; i > 0; i--) {
17   for(int y = 0; y < i; y++) {
18     double row_pivot = m_values[y * matrix_size + i];
19     for(int x = 0; x < i + 1; x++)
20       calc_reduced(m_values, row_pivot, x, y, i, matrix_size);
21   }
22 }
```

Listing 5 illustrates a code that brings matrix to row echelon form:

1. First loop, traverses through each row of the matrix, thus indicating on the current pivot row(lines 9-15).

2. Inner loop, traverses through all rows, lower than current pivot row and stores value of pivot column of this row in *row_pivot* variable(lines 10-14).

3. Finally, the entries of each row under pivot row are changed by traversing through their columns in the third loop(lines 12-13). Function *calc()*, Listing 6 (line 13), calculates new value for this entry.

4. Second loop, brings matrix to reduced row echelon form(lines 23-29). It traverses matrix from down-to-top and updates matrix entries in the same manner, as the first loop(line 9-15).

Listing 6: Implementation of Gauss-Jordan algorithm functions

```
1  void calc(double * matrix_A, double row_pivot, int thread_x, int
       thread_y, int pivot_id, int matrix_size) {
2    int x = thread_x + pivot_id, y = thread_y + pivot_id + 1;
3
4    if (!is_equal(row_pivot, 0)) { // do not compute, if it is already 0
5          double mult = matrix_A[pivot_id * matrix_size + pivot_id] /
               row_pivot;
6          if(x == pivot_id)
7               matrix_A[y * matrix_size + x] = 0;
8          else
9               matrix_A[y * matrix_size + x] = matrix_A[pivot_id *
                    matrix_size + x] - matrix_A[y * matrix_size + x] *
                    mult;
10     }
11 }
12
13 void calc_reduced(double * matrix_A, double row_pivot, int
       thread_x, int thread_y, int pivot_id, int matrix_size) {
14     int x = thread_x, y = thread_y;
15     if (!is_equal(row_pivot, 0)) { // do not compute, if it is already 0
16          double mult = matrix_A[pivot_id * matrix_size + pivot_id] /
               row_pivot;
17          if(x == pivot_id)
18               matrix_A[y * matrix_size + x] = 0;
19          else
20               matrix_A[y * matrix_size + x] = matrix_A[pivot_id *
                    matrix_size + x] - matrix_A[y * matrix_size + x] *
                    mult;
21     }
22 }
```

Function *calc()* in Listing 6 illustrates how particular matrix entry is updated:

1. Takes array of matrix values, current column's pivot value, x and y position of entry that needs to be changed, pivot row's id, and size of the whole matrix as arguments.
2. Calculate the ratio of pivot value to the pivot column of this row and store the result in *mult* variable.
3. If it is the pivot column of this row, assign value 0 to it.
4. Otherwise, subtract the current entry with corresponding column entry of the pivot row and write the results to the current entry.
5. Function *is_equal()* compares two double values with precision of 10^{-6}.

Logic of function *calc_reduced()* is the same, but it is used to update matrix in down-to-top function. See its usage in Listing 5 (line 22).

2.3 Complexity analysis

The program consists of 2 main loops with 2 nested loops per each.

For matrix with dimension N, each of these main loops goes through all the rows of the matrix. Given current pivot row i first and second inner loops go through $(N - i)$ entries each.

Complexity of both main loops with inner loops is $2 \cdot N \cdot (N - i) \cdot (N - i)$, which makes $O(N^3)$ time complexity.

Implementation on GPU

3 Concurrency of Gauss-Jordan algorithm and possible implementation

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 each row, could also be updated in parallel to other entries of the matrix. GPU could help us achieve high parallelization.

For $N \cdot N$ matrix, if we wish to achieve parallel processing of all entries, in theory, it could be possible to write a custom kernel, which would consist of one block of $N \cdot N$ threads.

As there are only 1024 threads per block available in GPU we are using for testing, it leaves us with a matrix of at most $N = 32$, which is very limited and is not worth of overhead of writing custom kernel and copy matrix entries from host to device.

To achieve parallelization, multiple blocks should be used that could run concurrently.

4 Carcenac's striping algorithm

The main focus of Carcenac's algorithm is in processing matrix by stripes of particular size and further usage of CUBLAS library. See Section 30.

This algorithm brings matrix to row echelon form.

The CUBLAS Library provides a GPU-accelerated implementation of the basic linear algebra subroutines (BLAS). See Section 31.

The main function that is used in this article is *cublasDgemm()*, which is used to multiply 2 matrices.

4.1 Motivation

Listing 7: Carcenac's motivation for using striping algorithm

```
1  // size of the matrix
2  int n;
3  // matrix A and B
4  double ** A, * B;
5  // matrix of multiplicatives
6  double * R;
7
8  /*
9  * population of matrices A and B here
10 */
11
12 for(int i = 0; i < n - 1; i++) {
13     for(int j = i; j < n - 1; j++) {
14         R[j] = A[j][i] / A[i][i];
15         B[j] = B[j] - R[j] * B[i];
16     }
17     for(int j = i; j < n - 1; j++) {
18         for(int k = i; k < n - 1; k++)
19             A[j][k] = A[j][k] - R[j] * A[i][k];
20     }
21 }
```

As in Listing 7, Carcenac divides matrix triangulation into 2 stages:

1. Find matrix of multiplicatives R that contains ratio of pivot entry to the pivot column of each row that is below pivot row: $R[j] = A[j][i]/A[i][i]$.

2. Change values of each column of row below pivot row. That is achieved by subtraction of corresponding entry of the multiplicatives matrix multiplied by corresponding entry of the pivot row from entry of the matrix that is meant to be updated: $A[j][k] = A[j][k] - R[j] \cdot A[i][k]$.

The motivation for using CUBLAS is line 19: $R[j] \cdot A[i][k]$, where multiplication of submatrix of A and matrix R could be efficiently solved by *cublasDgemm()*.

As submatrix of A could be very large, up-to $(N-1) \cdot (N-1)$ entries, for *cublasDgemm()* to efficiently calculate the product of matrices, submatrix should be divided into smaller matrices, thus decreasing the size of the matrix that would be processed by *cublasDgemm()*. That is achieved via striping.

4.2 Striping

Each stripe contains predetermined amount of rows.

Number of stripes depends on matrix size and number of rows per stripe: $N = n/s$, if s divides n, else $N = n/s + 1$, where N is the number of stripes, n is the number of rows in matrix, and s is the number of rows in the stripe.

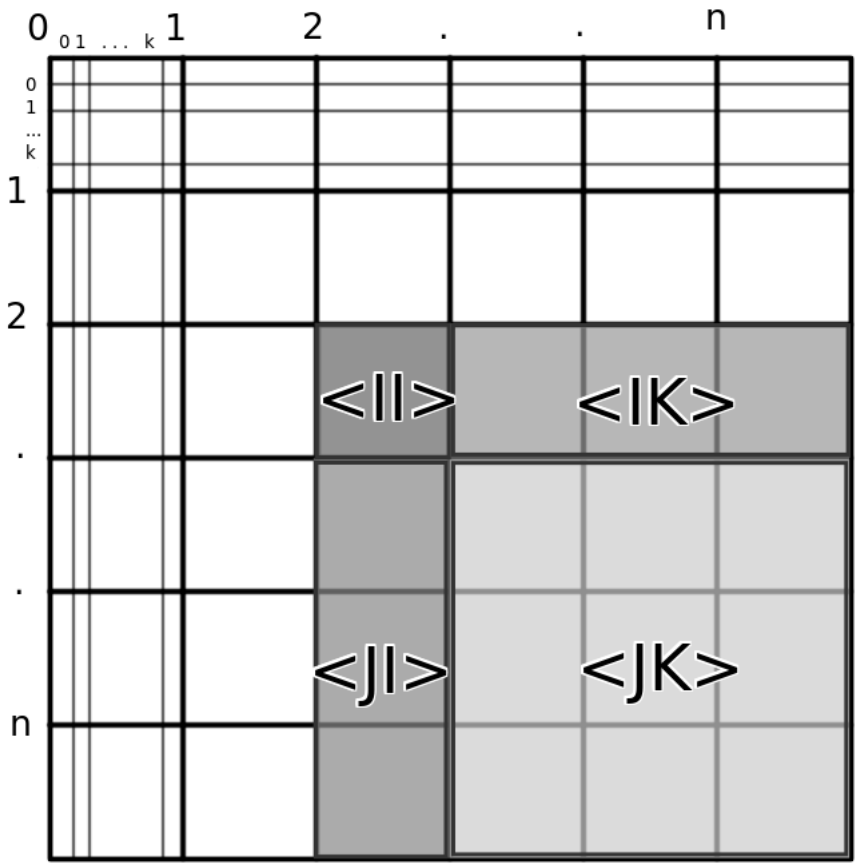

Figure 2: Stripes

Figure 2 illustrates created stripes that are split in tiles that will be processed by *cublasDgemm()*. Matrix is divided into n stripes with k rows in each.

There are 4 essential types of tiles in striping method:

- II is a pivot tile of the pivot stripe. It will have non-zero values in diagonal and all zeros below diagonal.
- IK is a leftover from the pivot stripe.
- JI is a grid of tiles from the pivot column.
- JK is a grid of tiles of leftover of JI stripes.

4.3 Processing of stripes

In order to process matrix correctly, stripes should be processed in particular order:

1. Tile II should be processed first, as it is the pivot tile and its values should be processed before going any further with other stripes.
2. Tiles IK should be processed right after, because lower stripes' values, depends on pivot stripe.
3. All the stripes, below the pivot stripe, are processed after pivot stripe.

4.4 Viability of using CUBLAS

There is a heavy reliance on CUBLAS library in Carcenac's artile. See Section 30. There are 2 main problems with using *cublasDgemm()* to bring matrix to row echelon for with striping method:

1. In all stripes, *cublasDgemm()* is used to multiply matrix R with the tile of the stripe that is being processed. See Section 4.1 for explanation. Every time, two matrices should be multipled, we would need to allocate them on device's global memory and, when processed, deallocate them, which is a very big overhead, as there N/s stripes(where N is the number of rows of the original matrix and s is the number of rows per stripe) and each stripe has 2 tiles:

   a) Pivot tile II or JI.

   b) The rest of the stripe IK or JK.

   We will get $N/s \cdot 2$ extra allocations and deallocations, which creates a very big overhead.

2. *cublasDgemm()* expects to take matrices, as parameters, in a column-major order, which means that a matrix must have column values instead of row values of the original matrix. See Section 32. So, before sending matrix to *cublasDgemm()*, it should be changed from row-major order to column-major, which is $n \cdot m$ complexity, where n is the number of rows and m is the number of columns of the matrix.

These two overheads make usage of *cublasDgemm()* non-viable, so writing custom kernels in CUDA, should be considered, instead of using CUBLAS.

5 Modification of striping algorithm

To get matrix in reduced row echelon form, striping algorithm should be applied twice:

1. From top-to-down to get matrix in row echelon form.
2. From down-to-top to get matrix in reduced row echelon form.

5.1 Differences between Carcenac's implementation and custom implementation

There are 2 main differences between Carcenac'c implementation of striping algorithm and an implementation, presented in this thesis:

1. In the modified striping algorithm, we are going to use custom kernels, instead of *cublas* library.
2. Streams are going to be used to parallelize the calculations between different stripes.

5.2 Adding CUDA streams

Each stripe from JI could be processed in parallel, as all stripes below pivot stripe could be processed concurrently, as they are only dependent on pivot stripe values.

All stripes from JI could be processed on separate CUDA streams.

Figure 3 illustrates stripes that are processed on separate CUDA streams.

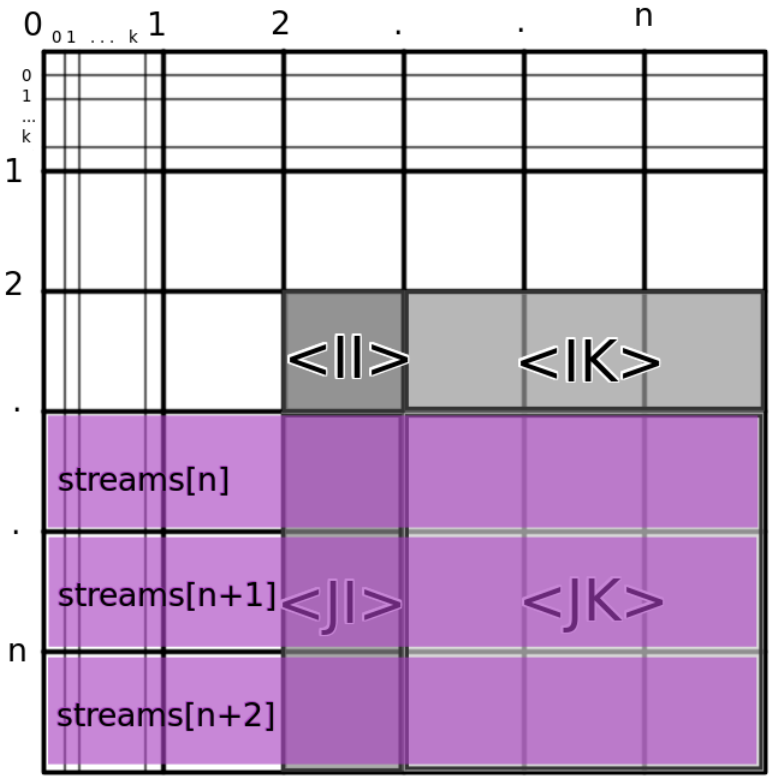

Figure 3: Streams

6 Types of blocks

As we have 1024 threads per block in CUDA kernel, that leaves us with the block with dimensions $32 \cdot 32$.

As last stripe and last block of each stripe could be truncated(number of rows in the matrix is not divisible by number of rows in one stripe, thus $n/32$ is not an integer, where n is the number of rows of this matrix), the stripes are divided into different types, which will be processed differently in code.

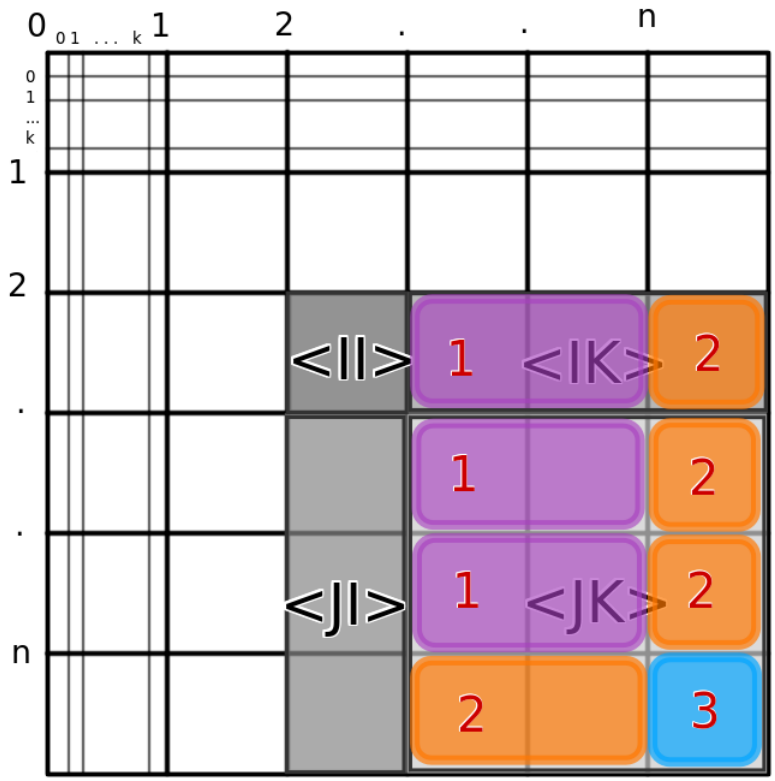

Figure 4: Types of blocks

Figure 4 illustrates types of blocks in stripes:

1. Blocks with complete dimensions: $32 \cdot 32$ entries.
2. Blocks with incomplete columns or rows sizes and are truncated from one side: less than 32 entries.

3. Blocks with incomplete columns and rows sizes and are truncated from both sides: both dimensions have less than 32 entries.

7 Implementation of modified algorithm

7.1 Data structures

Listing 8: Class that will store matrix

```
1  class Matrix {
2  public:
3      Matrix(int height, int width) {
4          this->m_height = height;
5          this->m_width = width;
6          this->m_values = new double[this->m_width * this->
               m_height];
7      }
8      Matrix(int height) { // one vector
9          this->m_height = height;
10         this->m_width = 1;
11         this->m_values = new double[this->m_width * this->
               m_height];
12     }
13     ~Matrix() {
14         delete []  this->m_values;
15     }
16 public: // members
17     int m_height, m_width;
18     double * m_values;
19 };
```

Listing 8 illustrates a class that stores matrix, as an array of doubles and its dimensions.

Listing 9: Multiplicatives array

```
1  struct Multiplicatives {
2      double * d_mult;
3  };
```

Listing 9 illustrates a class that will be used to store array of multiplicatives: entry values of matrix in pivot column.

7.2 Algorithm

7.2.1 Declaration of variables

Listing 10: Variables declaration

```
1 int matrix_size;
2
3 // declare matrix_size here
4
5 int max_stripe_size = 32;
6 Matrix matrix_A(matrix_size, matrix_size), matrix_A_cpu(
      matrix_size, matrix_size);
7 Matrix matrix_B(matrix_size), matrix_B_cpu(matrix_size);
8 Matrix matrix_out(matrix_size);
```

Listing 10 illustrates declaration of variables:

- *matrix_size* variable contains the size of the matrix. Matrix will be of dimensions $matrix\_size \cdot matrix\_size$.
- *max_stripe_size* variable contains the number of rows in one stripe. One tile of the stripe will be of the size $max\_stripe\_size \cdot max\_stripe\_size$.
- *matrix_A* is a matrix of coefficients with dimensions $matrix\_size \cdot matrix\_size$.
- *matrix_B* is a vector of linear equation results with size *matrix_size*.
- *matrix_out* is a vector of linear equations variables values with size *matrix_size*.

7.2.2 Main logic

First, we should come up with correct number of stripes.

As we already know that we will have stripes of size *max_stripe_size*, the number of stripes could be easily calculated by dividing size of the matrix by size of the stripe. If matrix can't be evenly divided by stripes, the last stripe will be of incomplete size, as it is illustrated in Listing 11.

Listing 11: Stripes number

```
1  // number of stripes
2  int stripes_count;
3  if(matrix_size % max_stripe_size == 0)
4      stripes_count = matrix_size / max_stripe_size;
5  else
6      stripes_count = matrix_size / max_stripe_size + 1;
7
8  // decide, if last stripe is truncated or not
9  int last_stripe_size = max_stripe_size;
10 if(stripes_count * max_stripe_size > matrix_size)
11     last_stripe_size = matrix_size - (stripes_count - 1) *
           max_stripe_size;
```

At this stage, it is important to create streams for stripe under pivot stripe and multiplicatives array that will be needed to store the values of the entries below the pivot entry(see Figure 5).

For each stripe, CUDA stream is created at this stage via *cudaStreamCreate()*.

Multiplicatives array for each stripe is allocated at this point to be reused throughout the code. To see the usage of multiplicatives array, see Figure 5.

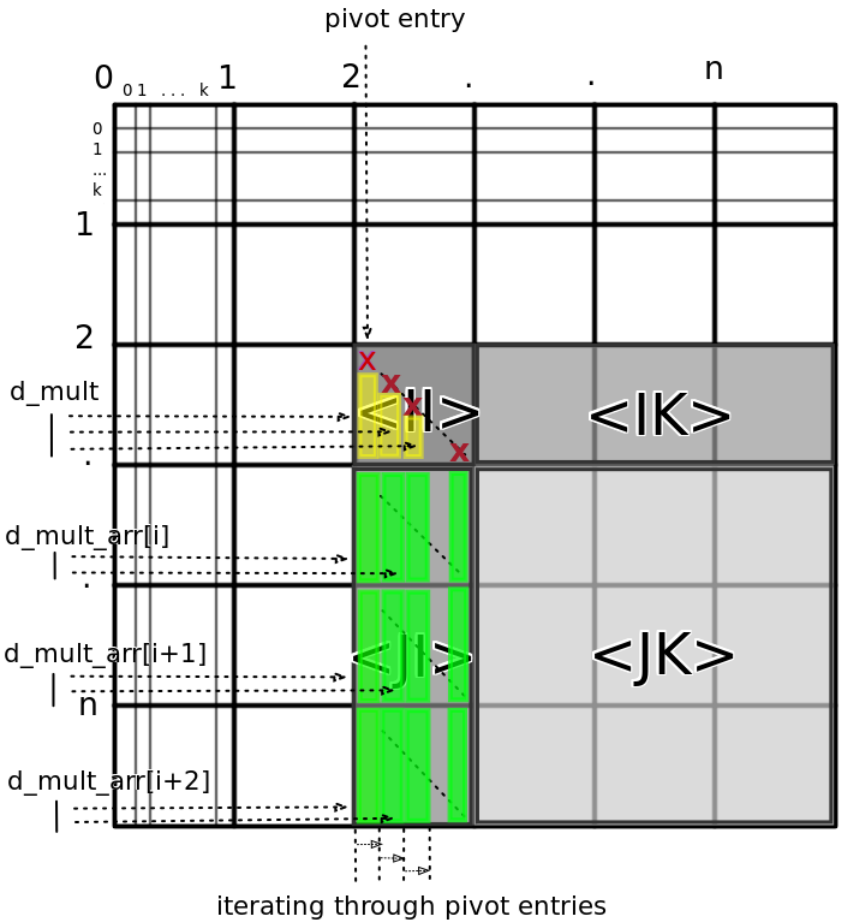

Figure 5: Multiplicatives array usage

Figure 5 illustrates the usage of multiplicatives arrays. Blocks II and JI are processed before remaining stripe blocks, so, current values of matrix entries in pivot column should be stored, because further, they will get updated. The entries in pivot column are stored in multiplicatives array.

After, we should bring matrix to row echelon form by going through each stripe, choosing it as a pivot stripe and modifying all lower stripes concurrently.

Listing 12: Traversing through stripes from top-to-down

```
1  for(int stripe_id = 0; stripe_id < stripes_count; stripe_id++) {
2
3      // iterate over rows of the pivot stripe here
4
5      // wait for the pivot stripe to be processed
6      cudaDeviceSynchronize();
7
8      // don't process other stripes, if it was last pivot stripe
9      if(stripe_id == stripes_count - 1) continue;
10
11     // iterate through lower stripes and update their rows here
12
13     // wait for streams synchronization
14     cudaDeviceSynchronize();
15 }
```

Listing 12 illustrates traversal through stripes.

Listing 13: Getting stripe size

```
1  // size of the current stripe
2  int block_pivot_dim = max_stripe_size;
3  // if it is the last stripe, the block dimensions could be truncated
4  if(stripe_id == stripes_count - 1)
5      block_pivot_dim = last_stripe_size;
```

In each iteration through the stripe, its size should be determined, as it could be the last stripe and its size could be truncated(number of rows in the stripe is less than 32). See Listing 13.

Listing 14: Iterating through rows of the stripe

```
1  for(int row_id = 0; row_id < block_pivot_dim - 1; row_id++) {
2      dim3 block_pivot(block_pivot_dim - row_id, block_pivot_dim -
           row_id - 1);
3      int shared_memory_size = (block_pivot_dim - row_id - 1) *
           sizeof(double);
4
5      /* kernel launch for processing II grid */
6
7      if(stripe_id == stripes_count - 1) continue;
8
9      int num_left_blocks = stripes_count - stripe_id - 2;
10     int column_stripe_offset = stripe_id + 1;
11
12     if(num_left_blocks > 0) {
13         dim3 grid_left(num_left_blocks, 1), block_left(
               block_pivot_dim, block_pivot_dim - row_id - 1);
14         /* kernel launch for processing IK grid */
15     }
16
17     column_stripe_offset += num_left_blocks;
18
19     dim3 block_last(last_stripe_size, block_pivot_dim - row_id - 1);
20     /* kernel launch for processing IK grid */
21 }
```

Iterate through each row of the pivot stripe and update the rows below the pivot row accordingly(see Listing 14):

1. Each update of the remaining rows, under pivot row, is executed in kernel with grid that consists of one block and a block with the dimensions of the sub-matrix under pivot row(line 2).

2. Shared memory size is determined, at this stage for the block that will be processed. It will be of the size of the rows number under pivot row(line 3).

3. Kernel *triangularize_normal_II* is called to process II tile(line 5). For the code of the kernel launch, see Listing 23.

4. If it was the last stripe, there are no IK blocks, in that case, we don't proceed with them(line 7). See Listing 16.

5. To process stripe IK, we should first calculate normal blocks of type 1. See Figure 4. Column offset for this grid is calculated(line 10), see Figure 6. Kernel *triangularize_normal_IK* will process the complete

blocks as one grid(lines 12-15). See Listing 17. For the code of the kernel launch, see Listing 24.

6. Column offset value is changed to the index of the last block of the pivot stripe and last block is processed separately, as it could be of truncated size(lines 17-20). See type 2 grid on Figure 4. For the code of the kernel launch, see Listing 25.

While the rows of the pivot stripe were processed, we should synchronize the host to wait until kernels finish their job. Done via *cudaDeviceSynchronize()*. See line 8 on Listing 12.

If it was the last stripe, no need to proceed further, as there won't be any lower stripes. Otherwise, proceed. See line 11 on Listing 12.

Listing 15: Traversing through lower stripes

```
1  for(int stripe_J_id = stripe_id + 1, stream_id = 0; stripe_J_id <
       stripes_count; stripe_J_id++, stream_id++) {
2      int block_height = max_stripe_size;
3      if(stripe_J_id == stripes_count - 1)
4          block_height = last_stripe_size;
5
6      for(int column_id = 0; column_id < max_stripe_size; column_id
           ++) {
7          dim3 block_pivot(max_stripe_size - column_id, block_height);
8          int shared_memory_size = block_height * sizeof(double);
9
10         /* kernel launch for proceesing JI grid */
11
12         int num_left_blocks = stripes_count - stripe_id - 2;
13         int column_stripe_offset = stripe_id + 1;
14
15         if(num_left_blocks > 0) {
16             dim3 grid_left(num_left_blocks, 1), block_left(
                   max_stripe_size, block_height);
17             /* kernel launch for processing JK grid */
18         }
19
20         column_stripe_offset += num_left_blocks;
21
22         dim3 block_last(last_stripe_size, block_height);
23         /* kernel launch for processing JK grid */
24     }
25 }
```

After the pivot rows were processed, rows of the lower stripes should be processed too. Listing 15 shows the iteration over the lower stripes and their update:

1. The size of the block is taken into account, as the last stripe could be of truncated size(lines 2-4). See type 2 grid on Figure 4.

2. Each pivot block JI of individual stripe will be iterated by columns, successfully iterating by pivot columns(lines 6-24).

For the code of the kernel launch JI, see Listing 26. For the code of the kernel launch to process complete blocks of JK, see Listing 27. For the code of the kernel launch to process the last block of JK, see Listing 28.

Before proceeding to the next pivot stripe, we should block host, until all the streams finish their job. Done via *cudaDeviceSynchronize()*. See line 19 on Listing 12.

Listing 16: Kernel to triangulize block II

```
1  __global__ void triangularize_normal_II(double * matrix_A,
       double * matrix_B, double * d_mult, int pivot_id, int
       matrix_size) {
2      int x = threadIdx.x + pivot_id, y = threadIdx.y + pivot_id + 1;
3
4      extern __shared__ double row_pivot[];
5      if(threadIdx.x == 0) {
6          row_pivot[threadIdx.y] = matrix_A[y * matrix_size + pivot_id
               ];
7          d_mult[threadIdx.y] = row_pivot[threadIdx.y];
8      }
9      __syncthreads();
10
11     if(!cuda_is_equal(row_pivot[threadIdx.y], 0)) {
12         double mult = matrix_A[pivot_id * matrix_size + pivot_id] /
               row_pivot[threadIdx.y];
13         if(x == pivot_id)
14             matrix_A[y * matrix_size + x] = 0;
15         else
16             matrix_A[y * matrix_size + x] = matrix_A[pivot_id *
                   matrix_size + x] - matrix_A[y * matrix_size + x] *
                   mult;
17
18         if(threadIdx.x == blockDim.x - 1)
19             matrix_B[y] = matrix_B[pivot_id] - matrix_B[y] * mult;
20     }
21 }
```

Listing 16 illustrates a kernel for block II:

1. It takes matrices that should be updated, id of the pivot entry in the matrix, matrix size, and an allocated array of multiplicatives, where the pivot column values will be stored.
2. Position of the current entry is calculated as thread indices + pivot's index in matrix(line 2).
3. To store the pivot column values in an array, each thread, before doing any computations, should wait until pivot column values are fetched into the shared memory and written into multiplicatives array and synchronized(lines 4-9).
4. The rest of the kernel is very straightforward and reminds Basic Gauss-Jordan algorithm implementation on CPU. See *calc()* function in section Section 2.2.

Listing 17: Kernel to triangulize block IK

```
1  __global__ void triangularize_normal_IK(double * matrix_A,
       double * matrix_B, double * d_mult, int pivot_id, int
       column_offset, int matrix_size) {
2      int x = blockDim.x * blockIdx.x + threadIdx.x + column_offset, y
           = threadIdx.y + pivot_id + 1;
3
4      extern __shared__ double row_pivot[];
5      if(threadIdx.x == 0)
6          row_pivot[threadIdx.y] = d_mult[threadIdx.y];
7      __syncthreads();
8
9      if (!cuda_is_equal(row_pivot[threadIdx.y], 0)) { // do not compute,
           if it is already 0
10         double mult = matrix_A[pivot_id * matrix_size + pivot_id] /
               row_pivot[threadIdx.y];
11         matrix_A[y * matrix_size + x] = matrix_A[pivot_id *
               matrix_size + x] - matrix_A[y * matrix_size + x] * mult;
12     }
13 }
```

Listing 17 illustrates kernel for block IK:

1. It takes matrices that should be updated, multiplicatives array, column offset of the processed grid, and matrix size.
2. Multiplicatives array is stored into shared memory and all threads are synchronized(lines 4-7).

3. The rest of the kernel is very straightforward and reminds Basic Gauss-Jordan algorithm implementation on CPU. See *calc()* function in section Section 2.2.

Listing 18: Kernel to triangulize block JI

```
1  __global__ void triangularize_normal_JI(double * matrix_A,
       double * matrix_B, double * d_mult, int pivot_id, int row_offset
       , int matrix_size) {
2      int x = threadIdx.x + pivot_id, y = threadIdx.y + row_offset;
3
4      extern __shared__ double row_pivot[];
5      if(threadIdx.x == 0) { // write once to not create racing
           conditions
6          row_pivot[threadIdx.y] = matrix_A[y * matrix_size + pivot_id
               ];
7          d_mult[threadIdx.y] = row_pivot[threadIdx.y];
8      }
9      __syncthreads();
10
11     if(!cuda_is_equal(row_pivot[threadIdx.y], 0)) { // do not compute,
           if it is already 0
12         double mult = matrix_A[pivot_id * matrix_size + pivot_id] /
               row_pivot[threadIdx.y];
13         if(x == pivot_id)
14             matrix_A[y * matrix_size + x] = 0;
15         else
16             matrix_A[y * matrix_size + x] = matrix_A[pivot_id *
                   matrix_size + x] - matrix_A[y * matrix_size + x] *
                   mult;
17
18         if(threadIdx.x == blockDim.x - 1)
19             matrix_B[y] = matrix_B[pivot_id] - matrix_B[y] * mult;
20     }
21 }
```

Listing 18 illustrates kernel for block JI, which is very close to the kernel for II(see Listing 16), with the only difference that, we need to know the row offset of current stripe, as it is below pivot stripe.

Listing 19: Kernel to triangulize block JK

```
1  __global__ void triangularize_normal_JK(double * matrix_A,
       double * matrix_B, double * d_mult, int pivot_id, int row_offset
       , int column_offset, int matrix_size) {
2      int x = blockDim.x * blockIdx.x + threadIdx.x + column_offset, y
           = threadIdx.y + row_offset;
3
4      extern __shared__ double row_pivot[];
5      if(threadIdx.x == 0)
6          row_pivot[threadIdx.y] = d_mult[threadIdx.y];
7      __syncthreads();
8
9      if(!cuda_is_equal(row_pivot[threadIdx.y], 0)) { // do not compute,
           if it is already 0
10         double mult = matrix_A[pivot_id * matrix_size + pivot_id] /
               row_pivot[threadIdx.y];
11         matrix_A[y * matrix_size + x] = matrix_A[pivot_id *
               matrix_size + x] - matrix_A[y * matrix_size + x] * mult;
12     }
13 }
```

Listing 19 illustrates kernel for block JK, which is very close to the kernel for IK(see Listing 17), with the difference that we need to know the row offset of this stripe(from which row it begins in the original matrix) and column offset of this grid(as this kernel will be processed as a grid of multiple blocks, the positions of blocks and threads inside the block we know and only need to know in advance the column of the original matrix, from which, this grid begins).

Function *cuda_is_equal()* compares two doubles with precision 10^{-6}.

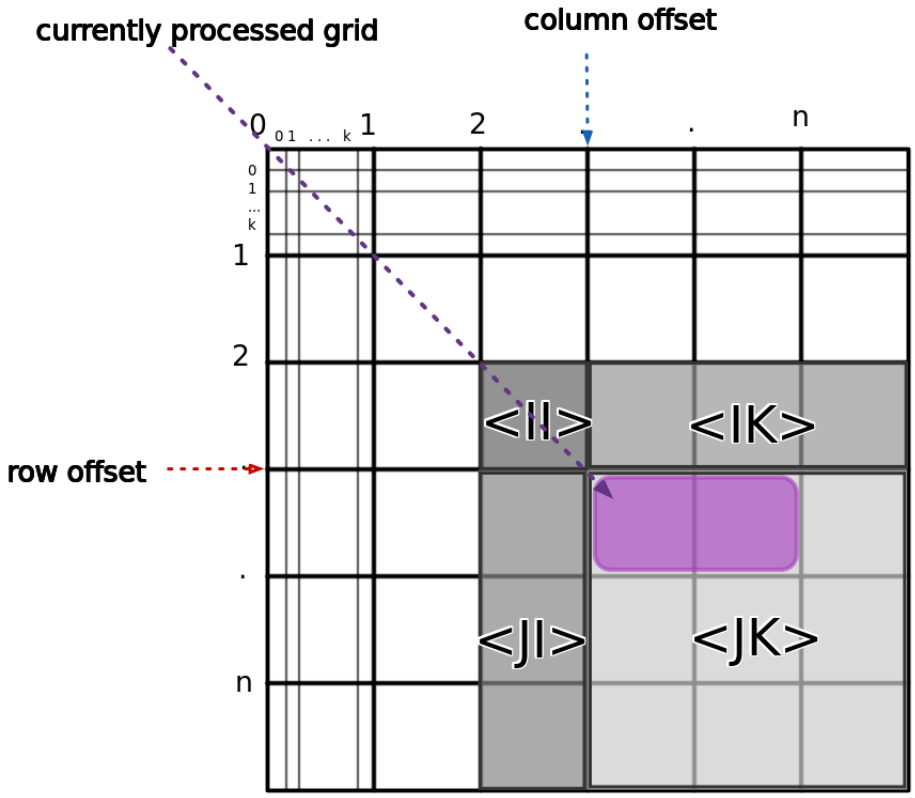

Figure 6: Stripe offsets

After bringing matrix to row echelon form, the same algorithm should be applied to matrix one more time, but in down-to-top fashion to bring matrix to reduced row echelon form. See Figure 7.

Listing 20: Iterating through stripes in down-to-top fashion

```
1  for(int stripe_id = stripes_count - 1; stripe_id >= 0; stripe_id--) {
2
3      // declare pivot blocks size
4      int block_pivot_dim = max_stripe_size;
5      // if it is the last stripe, the block dimensions could be
           truncated
6      if(stripe_id == 0)
7          block_pivot_dim = last_stripe_size;
8
9      // iterate over rows of the pivot stripe here
10
11     // wait for the pivot stripe to be processed
12     cudaDeviceSynchronize();
13
14     // don't process other stripes, if it was last pivot stripe
15     if(stripe_id == stripes_count - 1) continue;
16
17     // iterate through lower stripes and update their rows here
18
19     // wait for streams synchronization
20     cudaDeviceSynchronize();
21 }
```

Listing 20 illustrates traversal through stripes in down-to-top fashion:

1. Size of the pivot block is chosen: if it the last stripe, size could be truncated(lines 4-7).
2. Synchronize the host, so that it waits, until pivot stripe is processed(line 12).
3. Don't process other stripes, if pivot was the last stripe(line 15).
4. Synchronize the host, so that it waits, until all the streams that process non-pivot stripes finish their job(line 20).

Listing 21: Iteration through the rows of the pivot stripe

```
1  for(int row_id = block_pivot_dim - 1; row_id > 0; row_id--) {
2      dim3 block_pivot(row_id + 1, row_id);
3      int shared_memory_size = row_id * sizeof(double);
4
5      int block_offset = 0;
6      if(stripe_id > 0)
7          block_offset = matrix_size - (stripes_count - stripe_id) *
               max_stripe_size;
8
9      /* kernel launch for processing II grid */
10
11     if(stripe_id == 0) continue;
12
13     int num_left_blocks = stripe_id - 1;
14     int column_offset = matrix_size - (stripes_count - stripe_id +
           num_left_blocks) * max_stripe_size;
15
16     if(num_left_blocks > 0) {
17         dim3 grid_left(num_left_blocks, 1), block_left(
               block_pivot_dim, row_id);
18         /* kernel launch for processing IK grid */
19     }
20
21     column_offset = 0;
22
23     dim3 block_last(last_stripe_size, row_id);
24     /* kernel launch for processing IK grid */
25 }
```

Listing 21 illustrates traversal through rows of the pivot stripe through in down-to-top fashion. Logic is very close to the processing of pivot rows in top-to-down fashion, when matrix was brought to the row echelon form. See Listing 14.

For the code of the kernel launch II, see Listing 29. For the code of the kernel launch to process complete blocks of IK, see Listing 30. For the code of the kernel launch to process the last block of IK, see Listing 31.

Listing 22: Iterating through the stripes above the pivot stripe

```
1  for(int stripe_J_id = stripe_id − 1, stream_id = 0; stripe_J_id >=
       0; stripe_J_id−−, stream_id++) {
2      int block_height = max_stripe_size;
3      if(stripe_J_id == 0)
4          block_height = last_stripe_size;
5
6      for(int column_id = max_stripe_size − 1; column_id >= 0;
           column_id−−) {
7          dim3 block_pivot(column_id + 1, block_height);
8          int shared_mem_size = block_height ∗ sizeof(double);
9          int block_offset = 0;
10         if(stripe_id > 0)
11             block_offset = matrix_size − (stripes_count − stripe_id) ∗
                   max_stripe_size;
12         int row_offset = 0;
13         if(stripe_J_id > 0)
14             row_offset = matrix_size − (stripes_count − stripe_J_id)
                   ∗ max_stripe_size;
15
16         /∗ kernel launch for processing JI grid ∗/
17
18         int num_left_blocks = stripe_id − 1;
19         int column_offset = matrix_size − (stripes_count − stripe_id
               + num_left_blocks) ∗ max_stripe_size;
20         if(num_left_blocks > 0) {
21             dim3 grid_left(num_left_blocks, 1), block_left(
                   max_stripe_size, block_height);
22             /∗ kernel launch for processing JK grid ∗/
23         }
24         column_offset = 0;
25         dim3 block_last(last_stripe_size, block_height);
26         /∗ kernel launch for processing JK grid ∗/
27     }
28 }
```

After pivot stripe was processed, stripes above should be processed in down-to-top fashion. Listing 22 illustrates the iteration through stripes that are above the pivot stripes. The logic is the very close to the logic of the iteration through lower stripes in top-to-down fashion, when matrix was brought to the row echelon form. See Listing 15. The only difference is that, now, for each kernel that we start, we need to know the column and row offset of the grid that we will process in kernel, as all the stripes that are processed on separate streams are above the pivot stripe, we can't orient by pivot stripe any

more, as it was, when we traversed matrix in top-to-down fashion, so offsets should be used. See Figure 6.

Thus, kernels will be very similar to the ones we used to update matrix in top-to-down fashion. See Listing 35, Listing 36, Listing 37, Listing 38 for kernels that process grids II, IK, JK, JI correspondingly.

For the code of the kernel launch JI, see Listing 32. For the code of the kernel launch to process complete blocks of JK, see Listing 33. For the code of the kernel launch to process the last block of JK, see Listing 34.

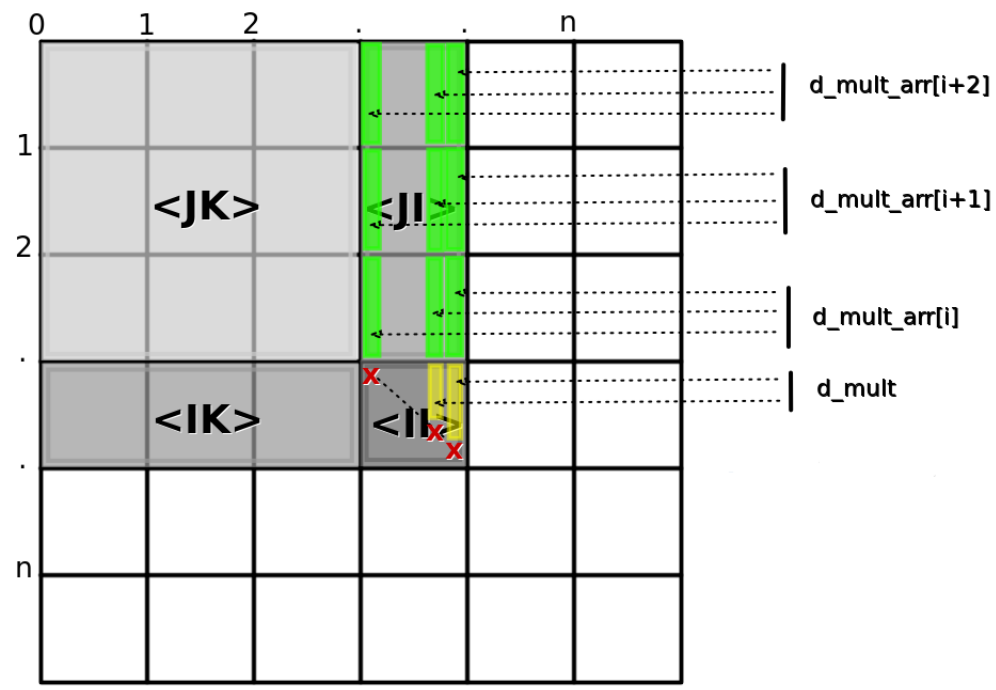

Figure 7: Processing matrix from down-to-top

Testing and evaluation of results

Values for testing are random *double* values in range from 1-1000 to not create an overflow during calculations, as the main focus of the thesis is speed of solving system of linear equations on GPU.

Execution time is calculated by C++ std::chrono library.

To ensure correctness of the calculations, the values of the *matrix_A* of the Modified striping algorithm(see section Section 2.2) are compared with the results of the calculations on CPU(see section Section 5).

As each stripe of the matrix will be chosen as a pivot stripe only once and as there are $N/32$ stripes, that makes $N/32$ iterations, where N is the number of rows of the matrix. So, overall complexity for now is $N/32$.

Each column, in the stripe, below the pivot stripe, will be processed roughly N^2 times, which makes an overall complexity $(N/32) \cdot N^2$ for now.

Update of the rows as a grid of blocks is done via CUDA kernel calls. There are 2 types of blocks for each stripe I, where pivot entry is located, and K–the rest of the stripe, see Figure 2. I is processed via a grid of one block, whereas K is processed as two grids:

1. The grid that contains all the leftover blocks, except the last block of this stripe, as it could be truncated. See type 1 grid on Figure 4.

2. The last block of this stripe, that is calculated as a grid of one block. See type 2 grid on Figure 4.

As every kernel does its job instantly, taking into account the threads that do separate from each other job in each kernel, it leaves us with 3 kernel calls per row, that makes overall complexity for now $(N/32) \cdot N^2 \cdot 3$

As the same algorithm will run again to bring matrix to reduced row echelon form, overall complexity will be $2 \cdot (N/32) \cdot N^2 \cdot 3$.

Overall complexity of $2 \cdot (N/32) \cdot N^2 \cdot 3$ makes $O(N^3)$ time complexity of the modified striping algorithm on GPU.

8 Comparison of execution time on GPU and CPU of Gauss-Jordan method

Figure 8 illustrates a graph with 2 curves: one for CPU execution time and another for GPU execution time with correspondence to matrix size. As it can be seen from the graph, GPU is very efficient with running modified striping algorithm to get a matrix in reduced row echelon form by Gauss-Jordan method. Matrix consists of *double* values.

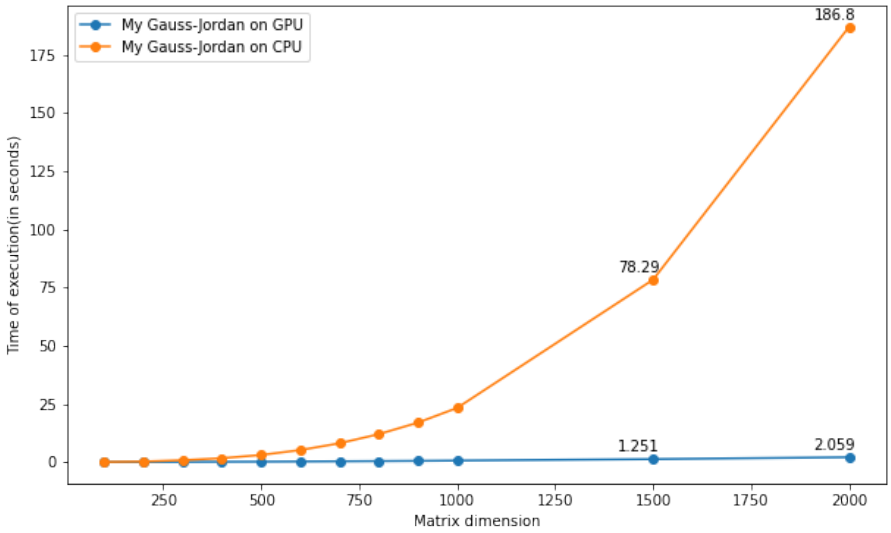

Figure 8: Execution time difference between GPU and CPU with double values

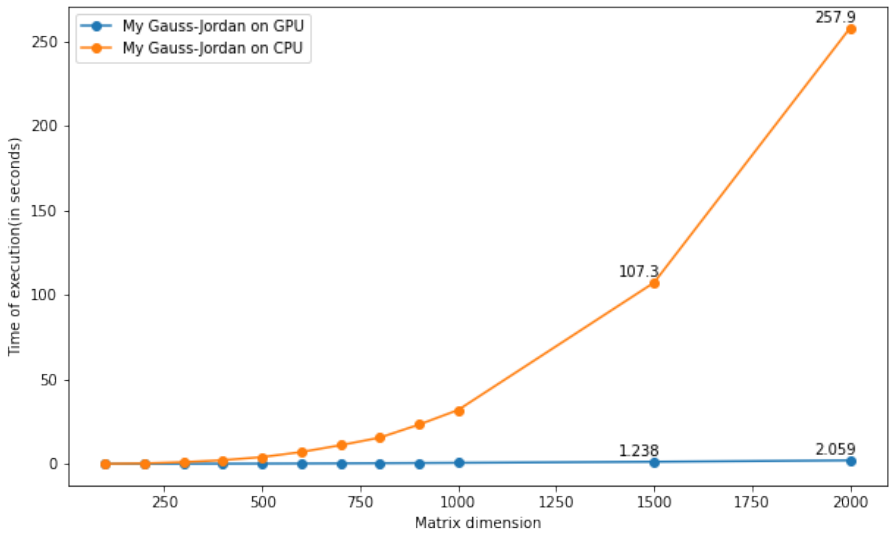

Figure 9: Execution time difference between GPU and CPU with float values

Figure 9 illustrates difference between execution time of GPU and CPU for a matrix with *float* values. As *float* is of single precision, it can be calculated faster, but the precision will be lower.

9 Comparison with other solution

So far we discussed how SLE could be solved based on Gauss-Jordan method. In the process of solving with this method, we get matrix in reduced row echelon form. Also, we compared the same Gauss-Jordan method on CPU and GPU and got the results in the favor of GPU.

Now, however, we will compare Gauss-Jordan method implementation on GPU using modified striping algorithm with custom kernels, with the already existing method provided by CUDA to solve dense matrices via LU decomposition with (cuSOLVER) library. See Section 33 for explanation of this method. See Section 34.

9.1 cuSolveDn

We are going to use *cuSolveDn* functions for solving dense matrices.

Test data consists of *double* values within range of 1-1000, so no zero's will be in the matrix, so, we need to use solver for dense matrices.

To compute LU factorization of the matrix *cusolverDnDgetrf()* function is used. See Section 33.

To solve the SLE, *cusolverDnDgetrs()* function is used. It takes in parameters a matrix that was LU-factored by *cusolverDnDgetrf()*. See Section 34.

9.2 Correctness of calculations

To ensure the correctness of this implmentation, the coefficients matrix should be multiplied with the resulting array of this *cuSolve* implementation and the resulting vector should be equal to the reference results vector.

The multiplication is done via *cublasDgemm()* function from *cublas* library.

Should be noted that before processing matrix via *cublasDgemm()* and *cuSolveDn()* functions, it should be transformed to column-major form, as CUDA libraries work with matrices in column-major form. See Section 32.

9.3 Comparison of Gauss-Jordan method with cuSolveDn implementation

Implementation of Gauss-Jordan method on GPU has a lot of overheads with launching kernels, copying data from global memory to shared memory, referencing global memory in kernels, context switches between streams, context switches between host and device. Host, waiting for CUDA kernels to finish their job, via *cudaDeviceSynchronize()* funtion is also a quite significant overhead. The matrix is traversed twice: once in top-to-down fashion, then in down-to-top fashion, which also makes an algorithm slower.

Whereas *cuSolveDn* functions were designed to specifically solve dense matrices, so the implementation of it is very efficient, memory-wise and speed-wise. LU decomposition is done only once and there is no need to traverse and update matrix after traversal through each pivot entry, as it is done in Gauss-Jordan method.

Figure 10 illustrates difference between execution time of *cuSolveDn* with modified striping algorithm(see Section 5) for matrix with *double* values.

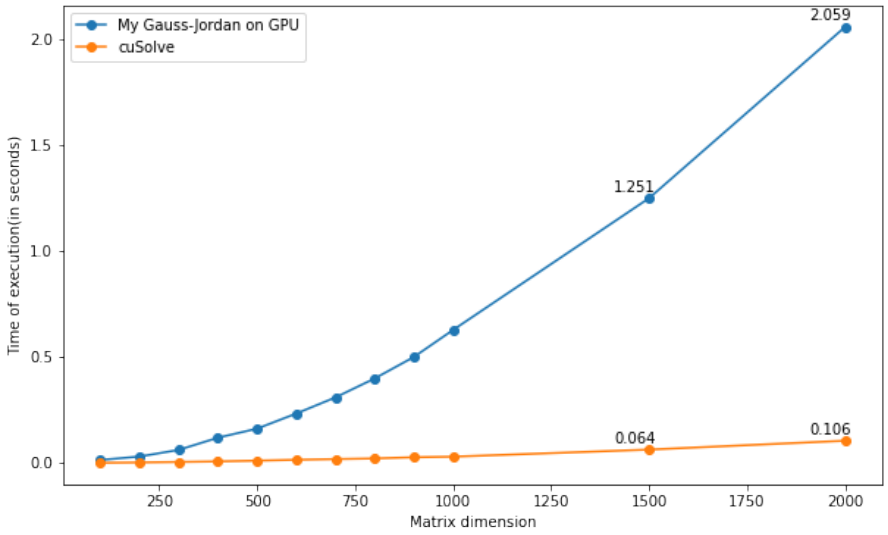

Figure 10: Execution time difference between Gauss-Jordan implementation and cuSolveDn implementation with double values

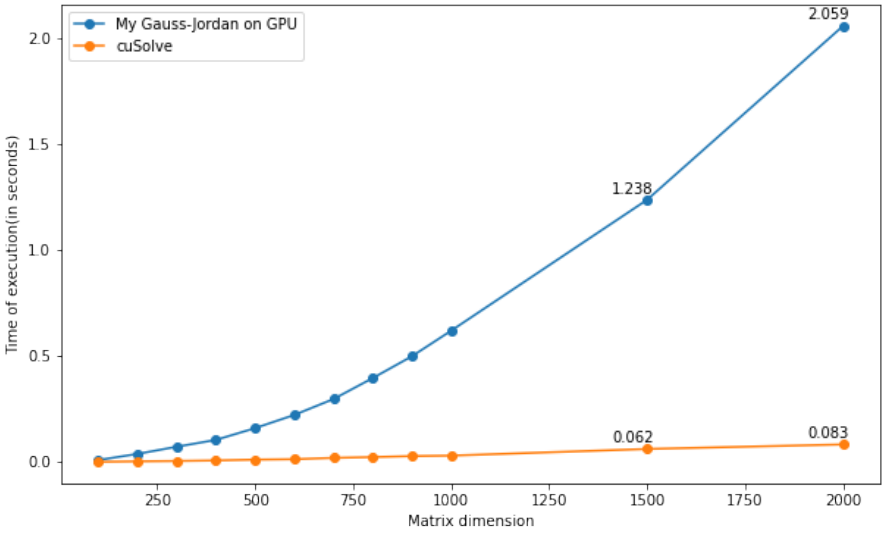

Figure 11: Execution time difference between Gauss-Jordan implementation and cuSolveDn implementation with float values

Figure 11 illustrates difference between execution time of *cuSolveDn* and modified striping algorithm(see Section 5) for a matrix with *float* values. As *float* is of single precision, it can be calculated faster, but the precision will be lower.

Conclusion

10 Overview

As we saw in this thesis, matrices can be solved via a highly parallel code using CUDA technology on GPU.

Two algorithms were implemented on CPU and GPU and, as we saw on the comparison graph(see section Section 8).

The results of calculations on GPU were compared with results of calculations on CPU to ensure correctness of calculations.

To implement Gauss-Jordan algorithm on GPU, striping technique was used, as number of threads per block in kernel call is limited, matrix should have been split into stripes and stripes into tiles(square sub-matrices) to process each of the tile separately as one grid of only one block(in case it is the pivot tile or a tile under the pivot tile) and a grid of multiple blocks, where blocks could be processed in parallel on device. The corner cases were also taken into account(last stripe and last tile of each stripe) and processed as a non-square sub-matrix in a separate kernel call.

Beyond simple parallelism of updating sub-matrix in the original matrix via a kernel call, we included streams to make kernel calls on separate stripes update the whole matrix concurrently. This concurrency of processing stripes also gives some advantage regarding performance, although it is not guaranteed that all of the streams will be processed at once on the device.

We also compared the presented modified striping algorithm for the implementation of Gauss-Jordan method for solving SLE with the already provided by CUDA, implementation of LU decomposition of matrix via *cuSOLVER* library. From the comparison, we showed that *cuSOLVER* library is much more efficient than custom implementation of Gauss-Jordan method, as it was meant to be very efficient with dense matrices. Although, if we need to get a matrix in reduced row echelon form, the modified striping algorithm, presented in this thesis, would be pretty efficient for such purpose.

11 Possible improvements of the modified striping algorithm

Variable of type *double*, lose precision during computations with numbers of very high range. As each CUDA kernel in this thesis contains a division of *double*s(this line: *double mult = matrix_A[pivot_id * matrix_size + pivot_id] / row_pivot[threadIdx.y];*), the precision is going to suffer a lot.

To prevent precision loss, special pivoting algorithms could be used, but it would make algorithm more complex. To read more about special pivoting algorithms, see Section 35.

Bibliography

20 System of linear equations

https://en.wikipedia.org/wiki/System_of_linear_equations

21 Basic Gaussian elimination

https://en.wikipedia.org/wiki/Gaussian_elimination

22 Matrix in row echelon form

https://www.geeksforgeeks.org/row-echelon-form/

23 Gauss-Jordan elimination

https://brilliant.org/wiki/gaussian-elimination/

24 CUDA

https://en.wikipedia.org/wiki/CUDA

25 CUDA kernels

https://medium.com/analytics-vidhya/cuda-compute-unified-device-architecture-part-2-f3841c25375e

26 CUDA kernel image

https://docs.nvidia.com/cuda/cuda-c-programming-guide/graphics/grid-of-thread-blocks.png

27 CUDA global memory

https://stackoverflow.com/questions/11178506/cuda-global-memory-where-is-it

28 CUDA shared memory

https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/

29 GPU computability

https://developer.nvidia.com/cuda-gpus

30 Carcenac's striping algorithm

https://link.springer.com/article/10.1007/s11227-013-1043-3

31 CUBLAS library

https://docs.nvidia.com/cuda/cublas/index.html

32 Matrix in column-major order

https://en.wikipedia.org/wiki/Row-_and_column-major_order

33 LU decomposition

https://en.wikipedia.org/wiki/LU_decomposition

34 cuSOLVER library

https://docs.nvidia.com/cuda/pdf/CUSOLVER_Library.pdf

35 Partial pivoting

https://www.sciencedirect.com/topics/mathematics/partial-pivoting

Appendix

A Kernel launches to bring matrix to row echelon form

A.1 Kernel launch for processing of II grid

Listing 23: Kernel launch II

```
1 triangularize_normal_II<<<1, block_pivot, shared_memory_size>>>(
      d_values_A, d_values_B, (d_mult_arr[0]).d_mult, stripe_id *
      max_stripe_size + row_id, matrix_size);
```

A.2 Kernel launches for processing of IK grid

Listing 24: Kernel launch IK for processing complete blocks

```
1 triangularize_normal_IK<<<grid_left, block_left,
      shared_memory_size>>>(d_values_A, d_values_B, (d_mult_arr
      [0]).d_mult, stripe_id * max_stripe_size + row_id,
      column_stripe_offset * max_stripe_size, matrix_size);
```

Listing 25: Kernel launch IK for processing the last block

```
1 triangularize_normal_IK<<<1, block_last, shared_memory_size>>>(
      d_values_A, d_values_B, (d_mult_arr[0]).d_mult, stripe_id *
      max_stripe_size + row_id, column_stripe_offset *
      max_stripe_size, matrix_size);
```

A.3 Kernel launch for processing of JI grid

Listing 26: Kernel launch JI

```
1 triangularize_normal_JI<<<1, block_pivot, shared_memory_size,
      streams[stream_id]>>>(d_values_A, d_values_B, (d_mult_arr[
      stream_id + 1]).d_mult, stripe_id * max_stripe_size + column_id,
       stripe_J_id * max_stripe_size, matrix_size);
```

A.4 Kernel launch for processing of JK grid

Listing 27: Kernel launch JK for processing complete blocks

```
1 triangularize_normal_JK<<<grid_left, block_left,
      shared_memory_size, streams[stream_id]>>>(d_values_A,
      d_values_B, (d_mult_arr[stream_id + 1]).d_mult, stripe_id *
      max_stripe_size + column_id, stripe_J_id * max_stripe_size,
      column_stripe_offset * max_stripe_size, matrix_size);
```

Listing 28: Kernel launch JK for the last block

```
1 triangularize_normal_JK<<<1, block_last, shared_memory_size,
      streams[stream_id]>>>(d_values_A, d_values_B, (d_mult_arr[
      stream_id + 1]).d_mult, stripe_id * max_stripe_size + column_id,
       stripe_J_id * max_stripe_size, column_stripe_offset *
      max_stripe_size, matrix_size);
```

B Kernel launches to bring matrix to reduced row echelon form

B.1 Kernel launch for processing of II grid

Listing 29: Kernel launch II

```
1 triangularize_reduced_II<<<1, block_pivot, shared_memory_size
      >>>(d_values_A, d_values_B, d_values_out, (d_mult_arr[0]).
      d_mult, block_offset + row_id, block_offset, matrix_size);
```

B.2 Kernel launch for processing of IK grid

Listing 30: Kernel launch IK for processing complete blocks

```
1 triangularize_reduced_IK<<<grid_left, block_left,
      shared_memory_size>>>(d_values_A, d_values_B, (d_mult_arr
      [0]).d_mult, block_offset + row_id, column_offset, block_offset,
      matrix_size);
```

Listing 31: Kernel launch IK for the last block

```
1 triangularize_reduced_IK<<<1, block_last, shared_memory_size>>>(
      d_values_A, d_values_B, (d_mult_arr[0]).d_mult, block_offset +
      row_id, column_offset, block_offset, matrix_size);
```

B.3 Kernel launch for processing of JI grid

Listing 32: Kernel launch JI

```
1 triangularize_reduced_JI<<<1, block_pivot, shared_mem_size,
      streams[stream_id]>>>(d_values_A, d_values_B, (d_mult_arr[
      stream_id + 1]).d_mult, block_offset + column_id, block_offset,
      row_offset, matrix_size);
```

B.4 Kernel launch for processing of JK grid

Listing 33: Kernel launch JK for processing complete blocks

```
1 triangularize_reduced_JK<<<grid_left, block_left, shared_mem_size,
      streams[stream_id]>>>(d_values_A, d_values_B, (d_mult_arr[
      stream_id + 1]).d_mult, block_offset + column_id, column_offset,
      row_offset, matrix_size);
```

Listing 34: Kernel launch JK for the last block

```
1 triangularize_reduced_JK<<<1, block_last, shared_mem_size, streams
      [stream_id]>>>(d_values_A, d_values_B, (d_mult_arr[stream_id
       + 1]).d_mult, block_offset + column_id, column_offset, row_offset
      , matrix_size);
```

C Kernels to bring matrix to reduced row echelon form

C.1 Kernel for processing of II grid

Listing 35: Kernel to triangulize block II

```
1  __global__ void triangularize_reduced_II(double * matrix_A,
       double * matrix_B, double * matrix_out, double * d_mult, int
       pivot_id, int block_offset, int matrix_size) {
2      int x = block_offset + threadIdx.x, y = block_offset + threadIdx.y;
3
4      extern __shared__ double row_pivot[];
5      if(threadIdx.x == blockDim.x - 1) { // write once to not create
           racing conditions
6          row_pivot[threadIdx.y] = matrix_A[y * matrix_size + pivot_id
               ];
7          d_mult[threadIdx.y] = row_pivot[threadIdx.y];
8      }
9      __syncthreads();
10
11     if(!cuda_is_equal(row_pivot[threadIdx.y], 0)) { // do not compute,
           if it is already 0
12         double mult = matrix_A[pivot_id * matrix_size + pivot_id] /
               row_pivot[threadIdx.y];
13         if(x == pivot_id)
14             matrix_A[y * matrix_size + x] = 0;
15         else
16             matrix_A[y * matrix_size + x] = matrix_A[pivot_id *
                   matrix_size + x] - matrix_A[y * matrix_size + x] *
                   mult;
17
18         if(threadIdx.x == 0) {
19             matrix_B[y] = matrix_B[pivot_id] - matrix_B[y] * mult;
20             if(!cuda_is_equal(matrix_A[y * matrix_size + x], 0))
21                 matrix_out[y] = matrix_B[y] / matrix_A[y *
                       matrix_size + x];
22             else
23                 matrix_out[y] = 0;
24         }
25     }
26 }
```

C.2 Kernel for processing of IK grid

Listing 36: Kernel to triangulize block IK

```
1  __global__ void triangularize_reduced_IK(double * matrix_A,
       double * matrix_B, double * d_mult, int pivot_id, int
       column_offset, int row_offset, int matrix_size) {
2      int x = blockDim.x * blockIdx.x + threadIdx.x + column_offset, y
           = threadIdx.y + row_offset;
3
4      extern __shared__ double row_pivot[];
5      if(threadIdx.x == blockDim.x - 1)
6          row_pivot[threadIdx.y] = d_mult[threadIdx.y];
7      __syncthreads();
8
9      if(!cuda_is_equal(row_pivot[threadIdx.y], 0)) { // do not compute,
           if it is already 0
10         double mult = matrix_A[pivot_id * matrix_size + pivot_id] /
               row_pivot[threadIdx.y];
11         matrix_A[y * matrix_size + x] = matrix_A[pivot_id *
               matrix_size + x] - matrix_A[y * matrix_size + x] * mult;
12     }
13 }
```

C.3 Kernel for processing of JI grid

Listing 37: Kernel to triangulize block JI

```
1  __global__ void triangularize_reduced_JI(double * matrix_A,
       double * matrix_B, double * d_mult, int pivot_id, int
       column_offset, int row_offset, int matrix_size) {
2      int x = threadIdx.x + column_offset, y = threadIdx.y + row_offset
           ;
3
4      extern __shared__ double row_pivot[];
5      if(threadIdx.x == blockDim.x - 1) { // write once to not create
           racing conditions
6          row_pivot[threadIdx.y] = matrix_A[y * matrix_size + pivot_id
               ];
7          d_mult[threadIdx.y] = row_pivot[threadIdx.y];
8      }
9      __syncthreads();
10
11     if(!cuda_is_equal(row_pivot[threadIdx.y], 0)) { // do not compute,
           if it is already 0
12         double mult = matrix_A[pivot_id * matrix_size + pivot_id] /
               row_pivot[threadIdx.y];
13         if(x == pivot_id)
14             matrix_A[y * matrix_size + x] = 0;
15         else
16             matrix_A[y * matrix_size + x] = matrix_A[pivot_id *
                   matrix_size + x] - matrix_A[y * matrix_size + x] *
                   mult;
17
18         if(threadIdx.x == 0)
19             matrix_B[y] = matrix_B[pivot_id] - matrix_B[y] * mult;
20     }
21 }
```

C.4 Kernel for processing of JK grid

Listing 38: Kernel to triangulize block JK

```
1  __global__ void triangularize_reduced_JK(double * matrix_A,
       double * matrix_B, double * d_mult, int pivot_id, int
       column_offset, int row_offset, int matrix_size) {
2      int x = blockDim.x * blockIdx.x + threadIdx.x + column_offset, y
           = threadIdx.y + row_offset;
3
4      extern __shared__ double row_pivot[];
5      if(threadIdx.x == blockDim.x - 1)
6          row_pivot[threadIdx.y] = d_mult[threadIdx.y];
7      __syncthreads();
8
9      if(!cuda_is_equal(row_pivot[threadIdx.y], 0)) { // do not compute,
           if it is already 0
10         double mult = matrix_A[pivot_id * matrix_size + pivot_id] /
               row_pivot[threadIdx.y];
11         matrix_A[y * matrix_size + x] = matrix_A[pivot_id *
               matrix_size + x] - matrix_A[y * matrix_size + x] * mult;
12     }
13 }
```

Acronyms

API Application Programming Interface. 5

CPU Central Processing Unit. 1, 5, 13, 28, 29, 37–40, 43

CUDA Compute Unified Device Architecture. 1, 5–8, 17, 19, 22, 37, 40, 41, 43, 44

GPGPU General Processing on Graphics Processing Unit. 5

GPU Graphics Processing Unit. 1, 5–7, 13, 37–41, 43

SLE System of Linear Equations. 1, 3, 40, 43