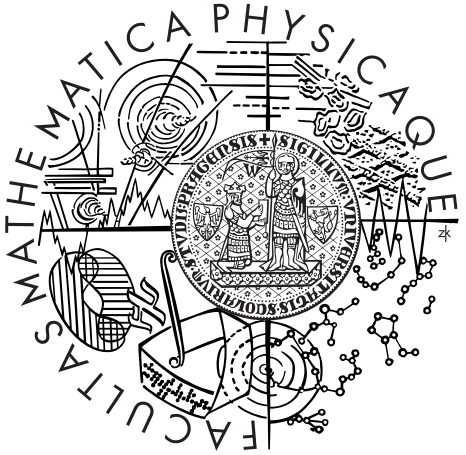
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Jan Dědeček

Implementing incomplete inverse decomposition on graphical processing units

Department of Applied Mathematics

Supervisor of the master thesis: prof. Ing. Miroslav Tůma, CSc.

Study programme: Informatics

Specialization: Discrete models and algorithms

Prague 2013

I would like to express my sincere gratitude to my advisor prof. Ing. Miroslav Tůma, CSc. for the continuous support of my master study and research, for his patience, motivation, enthusiasm, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. Also I would like to thank my family for supporting me throughout my life.

I declare that I carried out this master Thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... signature of the author

Název práce: Implementace neúplného inverzního rozkladu na grafických kartách

Autor: Jan Dědeček

Katedra: Katedra Aplikované Matematiky

Vedoucí diplomové práce: prof. Ing. Miroslav Tůma, CSc. , Ústav informatiky AV ČR, v. v. i.

Abstrakt: Cílem této práce bylo vyhodnocení možnosti řešit soustavy lineárních algebraických rovnic na grafických akcelerátorech. Zatímco řešiče obecně hustých soustav na těchto procesorech jsou víceméně součástí standardních uživatelských knihoven, tato práce se zaměřuje na soustavy řídké, kde tomu tak zdaleka není. Konkrétně se tato práce zaměřuje na jeden specifický algoritmus přibližného inverzního rozkladu symetrických a pozitivně definitních matic, který je kombinován s příslušnou krylovovskou metodou, metodou sdružených gradientů. Důležitou součástí této práce je inovativní paralelní implementace. Předkládané výsledky experimentů se systémy různých velikostí i struktur řídkosti ukazují, že nový přístup je slibný a v jeho vývoji by se mělo pokračovat. Sumárně, práce ukazuje, že přepodmiňování řídkých soustav přibližnými inverzemi na grafických akcelerátorech je jeden z možných efektivních postupů řešení.

Klíčová slova: Přibližná inverze, neúplná faktorizace, grafický procesor

Title: Implementing incomplete inverse decomposition on graphical processing units

Author: Jan Dědeček

Department: Department of Applied Mathematics

Supervisor: prof. Ing. Miroslav Tůma, CSc. , Institute of Computer Science of the ASCR, v. v. i.

Abstract: The goal of this Thesis was to evaluate a possibility to solve systems of linear algebraic equations with the help of graphical processing units (GPUs). While such solvers for generally dense systems seem to be more or less a part of standard production libraries, the Thesis concentrates on this low-level parallelization of equations with a sparse system that still presents a challenge. In particular, the Thesis considers a specific algorithm of an approximate inverse decomposition of symmetric and positive definite systems combined with the conjugate gradient method. An important part of this work is an innovative parallel implementation. The presented experimental results for systems of various sizes and sparsity structures point out that the approach is rather promising and should be further developed. Summarizing our results, efficient preconditioning of sparse systems by approximate inverses on GPUs seems to be worth of consideration.

Keywords: Approximate inverse, incomplete decomposition, graphical processing unit (GPU)

Contents

Introduction **3**

1 Systems of linear equations **6**
1.1 Methods for solving systems of linear equations 6
1.2 Direct methods 7
1.2.1 LU Decomposition 7
1.2.2 Cholesky decomposition 8
1.2.3 QR Decomposition 9
1.3 Iterative methods 11
1.3.1 The conjugate gradient method 11
1.3.2 The preconditioned conjugate gradient method 14

2 Sparse matrices **16**
2.1 Sparse systems of linear equations 16
2.1.1 Fill-in 18
2.2 Sparse matrix representation 19
2.2.1 Diagonal format 19
2.2.2 Compressed rows format 20
2.2.3 Coordinate format 20
2.2.4 Block format 21
2.2.5 Matrix-vector products 21

3 Preconditioners **23**
3.1 The incomplete Cholesky factorization 24
3.2 Approximations of inverse matrices 25
3.2.1 The inverse factorization 25
3.2.2 The inverse factorization of nonsymmetric systems 27
3.2.3 The stabilized inverse factorization 28
3.2.4 Comparison with other methods 30

4 Block diagonal matrices **31**
4.1 Issue of parallelism 31
4.2 Graph partitioning 32
4.2.1 Multilevel graph bisection 33
4.3 Relation of graphs and matrices 33
4.4 Matrix partitioning 34

5 Platform for parallel computing **35**
5.1 Kernels . . . 35
5.2 Thread Hierarchy . . . 35
5.3 Memory Hierarchy . . . 36
5.4 Matrix-Vector product . . . 37
5.4.1 Coordinate format . . . 37
5.4.2 Compressed rows format . . . 38

6 The stabilized inverse factorization implementation **41**
6.1 Basic method . . . 42
6.1.1 Parallel merging . . . 43
6.1.2 Parallel conditioned moving . . . 44
6.2 Coordinate method . . . 44
6.2.1 Practice realization . . . 45
6.2.2 Pros & Cons . . . 46
6.3 Implementation notes . . . 47

7 Numerical experiments **48**
7.1 Testing methodology . . . 48
7.1.1 Setting of the conjugate gradient method . . . 48
7.1.2 The testing hardware . . . 50
7.2 The conjugate gradient method . . . 50
7.3 The inverse factorization of block diagonal matrices . . . 53
7.3.1 Comparison with other methods . . . 54
7.3.2 Computation on GPU . . . 56
7.4 The stabilized inverse factorization . . . 57

8 Conclusion **59**

Introduction

Consider the problem of finding a solution for partial differential equations. Most of the problem can not be solved analytically, because their solution may not even exist. Therefore some numerical method must be used to obtain usually only an approximation of a solution. Two of many methods are the finite elements method or the finite difference method. They are based on discretization of a processed space and transforming a given problem into finding a solution for system of linear equation. Naturally, greater systems usually yield a better final approximation of the solution.

Now, consider a completely different problem. Back in old times the central processor unit (or any arbitrary processor) was sped up by increasing its clock rate, i.e., the speed at which the processor executed instructions. However, this has its own physical limits. The emitted heat increased quadratically with the clock rate, and so energy consumption also rose, which required better power sources and cooling systems.

If there is no possibility to take a step forward, one should take a sideways step. And that really happened in the computer world. Now, the chip industry is currently mainly focused on increasing the number of cores and more efficient ways to process code flow instead of rising clock rates. The result of this effort is, that the clock rate has been stable for the past five years but the real performance is much higher.

The graphics accelerators or the graphics processor units (GPU) have been going that way from the beginning. This is mainly caused by the different nature of their purpose. Acceleration graphics in most cases means processing a lot of very simple tasks, which are independent and therefore they can be processed concurrently. This allows graphics accelerators to be divided into multiple computing units, which are very simple. Theoretically, this concept may be more devoted to computation itself instead of instruction flow control and error correction.

Scientific GPU computing has been valid since year 2007, when NVIDIA released its platform for GPU computing. Before that year, GPU computing was very limited. Later on other platforms for GPU computing like OpenCL, which is open standard for GPU programming, were released. Nowadays, parallel computing is possible on various platforms on hardwares from different producers.

The goal of this Thesis is to try to utilize graphics accelerator in solving system of linear equation. There are many methods for solving system of linear equation, and this Thesis is only focused on one of them. I use a very famous iterative method called the conjugate gradient method to obtain a solution of systems of linear equations. Although this method is simple, it is also very efficient and optimal.

Various techniques exist to decrease times of computation of the conjugate

gradient method. They affect solving of system by changing numerical properties of process systems. One of these such techniques is called a preconditioning. The idea of the preconditioning is to somehow get nearer to the solution. The preconditioners used in this Thesis were based on an algorithm proposed by Benzi and my advisor Tůma [7].

A lot of research has been done in this area lately. Authors of the paper [12] tried to solve symmetric and positive definite system of linear equation with GPU, but they used a different preconditioner. They were not interested in the computing preconditioner itself, but they focused on its effect on the parallel implementation of related iterative method.

Work similar to this Thesis was done in an unpublished paper [8], where authors used the preconditioner of the same family, although they focused on nonsymmetric system of linear equation. Unlike paper [12], they were interested in the parallel implementation of the used preconditioner rather then parallel implementation of related iterative method.

Structure of Thesis

The Thesis is organized as follows:

Chapter 1 gives a brief introduction of systems of linear equations. It contains descriptions of the most common methods for obtaining solutions of system of linear system. There are mentioned both types of method, direct and iterative. The iterative method is represented only by the conjugate gradient method, because the scope of this Thesis is not about solving linear itself.

The following Chapter 2 introduces sparse matrices and sparse system of linear equation.The sparse system arises naturally from many applications. There is also defined the fill-in, property of direct solver of sparse system, and how it is affected by matrix reordering. The requirement on different realization of operation for sparse matrices is demonstrated on matrix-vector product, which is crucial for many algorithms.

Chapter 3 is about the preconditioning of system of linear equation. It contains an explanation of what implicit and explicit preconditioners are. The implicit preconditioners, like the incomplete Cholesky decomposition, is only briefly mentioned there, since a lot of research was done in this area. The explicit preconditioners are represented by the inverse factorization and the stabilized inverse factorization.

In Chapter 4 the block diagonal matrices is described. There is defined what are block diagonal matrices and their properties related to the parallel implementation method of the working of them. A way, how to decompose any arbitrary matrix into blocks is also briefly described there.

The used platform for GPU computing is described in Chapter 5. It contains information like organization of memory and threads of NVIDIA CUDA. There is also a description how to compute sparse matrix-vector product on GPU which is required by the implementation of the conjugate gradient method on GPU.

Chapter 6 contains the description of implementation of the stabilized inverse factorization. There is an explanation why straightforward implementation of stabilized inverse factorization is not efficient on GPU (but it is possible), and

what changes have to be done to the algorithm. There is also a discussion on some difficulties of realization.

Chapter 7 contains practical tests on available hardware.

Chapter 1

Systems of linear equations

A general system of m linear algebraic equations with n unknowns can be written as

$$\begin{array}{ccccccccc} a_{11}x_1 & + & a_{12}x_2 & + & \dots & + & a_{1n}x_n & = & b_1 \\ a_{21}x_1 & + & a_{22}x_2 & + & \dots & + & a_{2n}x_n & = & b_2 \\ \vdots & & \vdots & & \ddots & & \vdots & = & \vdots \\ a_{m1}x_1 & + & a_{m2}x_2 & + & \dots & + & a_{mn}x_n & = & b_m \end{array},$$

where $x_1, x_2, \dots, x_n$ are the unknowns, $a_{11}, a_{12}, \dots, a_{mn}$ are the coefficients of the system, and $b_1, b_2, \dots, b_m$ are the constant terms.

If we denote m by n matrix $A = [a_{ij}]$ and vectors $x = [x_1, \dots, x_n]$ and $b = [b1, \dots, b_m]$, then we can rewrite system of linear equation in the following matrix form

$$Ax = b$$

1.1 Methods for solving systems of linear equations

Method for solving system of linear equations can be classified as either direct or iterative, depending on how they obtain the solution.

Direct methods attempt to solve the problem by a finite sequence of operations. In the absence of rounding errors, direct methods would deliver an exact solution. Usually, they implicitly construct an inverse matrix A^{-1} and then obtain the solution x^*. Computed inverse matrix A^{-1} can be used multiple times for different right sides b.

Iterative methods for solving system of linear equations is a procedure that generates a sequence of improving approximate solutions. An iterative method uses an initial guess x_0 to generate successive approximations to a solution. The time of computation is usually dependent on how far guess x_0 is from optimal solution x^*. Generally, iterative solvers return only approximation of x^* after a finite number of iterations.

1.2 Direct methods

We divide direct solver into two groups. The first group contains methods based on matrix realization of Gauss elimination like LU and Choelsky decompositions. The second group are QR decompositions, which construct orthonormal matrices Q and upper triangular matrices R.

1.2.1 LU Decomposition

The next discussed method decompose n by n matrix A with full rank into product $A = LU$, where matrix L is lower triangular matrix and U is upper triangular matrix. It should be mentioned, that LU decomposition is the well-known Gaussian elimination written in matrix form. Let us expand product $A = LU$:

$$\begin{pmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \dots & a_{nn} \end{pmatrix} = \begin{pmatrix} l_{11} & \dots & 0 \\ \vdots & \ddots & \vdots \\ l_{n1} & \dots & l_{nn} \end{pmatrix} \begin{pmatrix} r_{11} & \dots & r_{1n} \\ \vdots & \ddots & \vdots \\ 0 & \dots & r_{nn} \end{pmatrix}$$

Product can be also written by definition as those n^2 equations:

$$a_{ij} = l_{i1}u_{1j} + \sum_{k=2}^{min\{i,j\}} l_{ik}u_{kj}$$

In Gaussian elimination, diagonal entities l_{ii} are usually set to ones. If we stay with this convection, values of elements $l_{11}, \dots, l_{n1}, u_{11}, \dots, u_{1n}$ can be computed directly. However, when element a_{11} is zero, LU decomposition is not possible.

Let us denote vertices $l_k = (0, \dots, 1, l_{k+1k}, \dots, l_{nk})^T$ and vertices $u_k = (0, \dots, u_{kk}, \dots, u_{kn})$. Then, product LU can be written in the following recurrent form

$$A^{(k+1)} = A^{(k)} - l_k u_k,$$

where $A^{(1)} = A$ and $A^{(k)} = [a_{ij}^{(k)}]$ is matrix which have first $k-1$ rows and columns filled only with zeros. Diagonal element $a_{kk}^{(k)}$, also called pivots, must be nonzeros, otherwise LU decomposition is not feasible. Vertices l_k and u_k can be computed directly from matrix $A^{(k)}$. Because the matrix contains only zeros, matrix A can be expressed as

$$\mathbf{0} = A^{(n+1)} = A^{(1)} - \sum_{i=1}^{n} l_i u_i \Rightarrow A = \sum_{i=1}^{n} l_i u_i.$$

It can be seen, that vertices l_i are columns of factor L and vertices u_i are rows of factor R.

$$A = \sum_{i=1}^{n} l_i u_i = \sum_{i=1}^{n} \sum_{j=i}^{n} l_i u_{ij} = \sum_{i=1}^{n} \sum_{j=1}^{i} l_j u_{ji} = LU$$

Now, we examine some properties of LU decompositions. Note, that we suppose the diagonal entities l_{ii} to be ones.

Theorem *If decomposition $A = LU$ exists, then it is unique.*

Proof See [14, p. 76].

The last theorem spoke about the uniqueness of LU decomposition but nothing about existence itself. Existence is conditioned by nonzero values of pivot elements $a_{kk}^{(k)}$, otherwise LU decomposition can be completed. The following theorem will determine the requirement on matrices to be LU decomposable.

Theorem *Pivots elements $a_{kk}^{(k)}$ are nonzero if and only if main submatrices*

$$\begin{pmatrix} a_{11} & \dots & a_{1i} \\ \vdots & & \vdots \\ a_{i1} & \dots & a_{ii} \end{pmatrix}$$

are nonsingular for $i = 1 \dots n$.

Proof See [2, p. 13] .

It is obvious, that there are matrices which satisfy the given conditions and therefore their LU decomposition exists and it is also unique.

LU decomposition isn't defined for all kinds of matrices. LU decompositions are impossible, when pivot elements $a_{kk}^{(k)}$ are zero. The solution is to select different pivot nonzero element $a_{pk}^{(k)}$ below diagonal $a_{kk}^{(k)}$. Then kth and pth rows are switched and $a_{pk}^{(k)}$ become the new pivot. From the point of view of linear equation system operation switching rows are correct and do not affect the solution set.

The final question is to how to obtain solution set from decomposition $PA = LU$. Neither L nor U are invertible by transposition, so system of linear equation cannot be solved in the same way. Fortunately, an exact solution can be obtained by these clever substitutions:

$$Ly = Pb$$

$$Ux = y$$

Those equations can be solved by an operation called backward sweep.

1.2.2 Cholesky decomposition

Symmetric real or hermitian complex matrices, which are also positive definite, can be decomposed into products of matrices LL^*, where L is lower triangular matrix. There are several variants of decomposition algorithm, we will show so called outer product variant.

Let us introduce matrices $A^{(1)}, \dots, A^{(n)}$, where $A^{(1)}$ is A. For convenience, all introduces matrices will be symbolically decomposed to

$$A^{(i)} = \begin{pmatrix} I_{i-1} & 0 & 0 \\ 0 & a_{ii} & b_i^* \\ 0 & b_i & B_i \end{pmatrix},$$

where b_i is vector of size $n - i$ and B_i is matrix of size $n - i$ by $n - i$. One step of algorithm is defined as

$$A^{(i)} = L^{(i)} A^{(i-1)} L^{(i)*},$$

where L_i are lower triangular matrices equal to

$$L^{(i)} = \begin{pmatrix} I_{i-1} & 0 & 0 \\ 0 & \sqrt{a_{ii}} & 0 \\ 0 & \frac{1}{\sqrt{a_{ii}}} b_i & I_{n-i} \end{pmatrix}.$$

To be complete, following equations describes relation among B_i matrices as

$$\begin{pmatrix} a_{ii} & b_i^* \\ b_i & B_i \end{pmatrix} = B_{i-1} - \frac{1}{a_{i-1i-1}} b_{i-1} b_{i-1}^*$$

Final triangular matrix L can be obtained by the product of matrices $L = L^{(1)} L^{(2)} \dots L^{(n)}$. Note that, it is not necessary to explicitly multiply those matrices. Except the main diagonal (which contains only ones) they have only one used column and each matrix has a different column. So matrix L is formed by those columns.

Theoretically, if matrix A is positive definite, then diagonal elements a_{ii} are always grater than zero, so the next step is always defined and no permutation is needed. It is not possible for elements a_{ii} to be at most zero. However, in practice float arithmetic, and its rounding errors, could cause the element value to fall below zero, especially when matrix A was ill-conditioned. In that case LDL^T decomposition should be used.

1.2.3 QR Decomposition

Let A be an m by n matrix with $m \geq n$ and full column rank. Then matrix A can be decomposed into a product of two special matrices.

$$A = QR,$$

where Q is an orthogonal m by m matrix and R is an upper triangular matrix. In solving systems of linear equations, QR decomposition can be used to transform linear equation into directly solvable form

$$Ax = QRx = b \Leftrightarrow Rx = Q^T b,$$

where exact solution can be obtained by backward sweep. In the following text we discus variants of QR decompositions.

Gram-Schmidt process

The Gram-Schmidt process constructs an orthonormal base from a given linear base. In QR decomposition, columns $a_1, \dots, a_n$ of matrix A are transformed to orthonormal vertices $q_1, \dots, q_n$ by formula:

$$u_i = a_i - \sum_{j=1}^{i-1} \langle a_i, q_j \rangle \, q_j, \quad q_i = \frac{u_i}{\|u_i\|}, \quad i = 1, \dots, n$$

It is clear, that vertices $q_i, \dots, q_n$ are orthonormal, and therefore they can be used as columns of matrix Q. If $A = QR$ product is expanded then the first

equation is $a_1 = r_{11}q_1$. But, that means $r_{11} = \langle a_1, q_1 \rangle$ because q_1 is the unit vector. So, product equations can be rearranged to

$$a_i = \sum_{j=1}^{i} \langle a_i, q_j \rangle \, q_j.$$

Above equations express columns $a_1, \ldots, a_n$ as linear combinations of vertices $q_1, \ldots, q_n$. Clearly, those combinations can be written as a multiplication of matrices Q and R, where Q is already constructed and matrix R formed by elements

$$r_{ij} = \begin{cases} \langle a_j, q_i \rangle, & \text{if } j \leq i \\ 0, & \text{if } j > i \end{cases}.$$

Givens rotations

Givens rotation is a linear transformation which affects linear space by rotating its two dimensional plane projection. From the point of view of rotated vector, Givens rotation only changes its two elements. Givens rotation can be expressed in matrix form

$$Q^{(i,j,\theta)} = \begin{array}{c} \\ i \\ \\ j \\ \\ \end{array} \begin{array}{c} \begin{array}{ccccc} & & i & & j \end{array} \\ \begin{pmatrix} \ddots & & & & \\ & cos(\theta) & & -sin(\theta) & \\ & & \ddots & & \\ & sin(\theta) & & cos(\theta) & \\ & & & & \ddots \end{pmatrix} \end{array}$$

where θ is a rotation angle and i,j are projected coordinate to two dimensional plane. Matrices $Q^{(i,j,\theta)}$ are called a rotation because they are orthogonal, and therefore length preserving.

In QR decomposition, elements under main diagonal have to be set to zeros. Let us suppose, that a_{kj}, a_{lj} of matrix A are rotated elements and a_{lj} is a zeroed element. Unknowns $cos(\theta), sin(\theta)$ can be obtained by solving these two equations

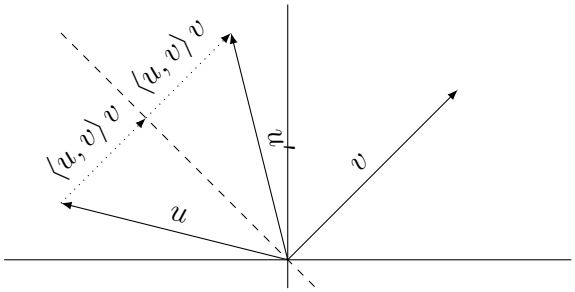
$$\begin{pmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{pmatrix} \begin{pmatrix} a_{kj} \\ a_{lj} \end{pmatrix} = \begin{pmatrix} \sqrt{a_{kj}^2 + a_{lj}^2} \\ 0 \end{pmatrix}.$$

Note, that it is not necessary express angle θ. This process shall be repeated until matrix R is complete.

It is required to do at most $\frac{1}{2}(n-1)^2$ rotation to reduce matrix A to R. However, matrices representing rotations mustn't be constructed explicitly and rotation may be implemented to affect only incident columns (matrix R) or rows (matrix Q).

Householder reflections

The main idea of Householder reflections is to reflect vector x around a hyperplane containing the origin. Let v be unit vector orthogonal to hyperplane and u is reflected vector, then its reflection u' is equal to $u' = u + 2\langle uv \rangle v$. Reflection equation can be written in matrix form $Q_v = I + 2v^*v$. Matrix Q_v is unitary.



In QR decomposition, householder reflections are used to eliminate nonzeros under the main diagonal by reflecting columns a_i. The first step of algorithm reflects first column a_1 to $e_1 \|a_1\|$. The first unit vector v_1 orthogonal to hyperplane is equal to $\frac{e_1\|a_1\|-a_1}{\|e_1\|a_1\|-a_1\|}$. After application of reflection, all element under the first diagonal are zero. Then, the first row and first column are dropped and the process repeated until there are no row and column to drop.

$$R = Q_{n-1} \dots Q_1 A \Rightarrow Q = Q_1^* \dots Q_{n-1}^*$$

1.3 Iterative methods

In this section we will discuss two types of iterative methods. Stationary methods are older, simpler to understand and implement, but usually not as effective. Nonstationary methods are a relatively recent development; their analysis is usually harder to understand, but they can be highly effective.

Stationary Iterative Methods are those methods, which can be written as the simple form

$$x^{(k+1)} = Bx^{(k)} + c,$$

where neither B nor c depend upon the iteration count k. The most famous representatives of stationary iterative method are the Jacobi method, the Gauss-Seidel method, and the Successive Overrelaxation (SOR).

Nonstationary Iterative Methods differ from stationary methods in that the computations involve information that changes at each iteration. Typically, constants are computed by taking inner products of residuals or other vectors arising from the iterative method.

1.3.1 The conjugate gradient method

The introduction of the conjugate gradient method starts quite unusually. The original problem, obtaining solution of $Ax = b$, is being transformed to a different one, where quadratic function $f(x)$ is minimized. We show efficient algorithm, which can minimize functional $f(x)$ and therefore solve related system of linear equation. Function $f(x)$ is defined as follows:

$$f(x) = \frac{1}{2}\langle x, Ax\rangle - \langle x, b\rangle = \frac{1}{2}x^T Ax - x^T b, \qquad x \in \mathbb{R}^n$$

Function f is at least two times continuously differentiable, and its gradient is equal to $Ax - b$. Hessian, matrix form by second order derivations, is equal to the matrix A. If the matrix A is positive definite, then function $f(x)$ is convex on its domain. If function $f(x)$ is convex and its domain is also a convex set, then there is a unique solution x^* such that minimizes function $f(x)$ and satisfies $Ax - b = 0$.

The conjugate gradient method is restricted to positive define[1] and symmetric matrices. Requirement on positive definiteness follows from above paragraph and symmetry is required in further formulas. Positive define and nonsymmetric system can be solved by similar algorithm, the biconjugate gradient method [14, p. 370].

Minimum of function f can be obtained by the method of steepest descent. This method approaches to the optimal solution x^* by improving current until desired optimum is reached. One step can be written as follows:

$$x_{i+1} = x_i + \alpha_i p_i,$$

or if previous equations are multiplied by matrix A, then they can be rewritten in residual form

$$r_{i+1} = r_i - \alpha_i A p_i,$$

where $r_i = b - Ax_i$ are ith residuals.

Parameters α_i should minimize values of function f, where arguments are restricted to those which are laying on the line determined by current approximation of x_i and current direction p_i. Generally, a difficult multivariate problem is reduced to univariate, which can be solved more easily. The following equation expands function f. (Notice that the matrix A is symmetric.)

$$\begin{aligned} f(x_i + \alpha_i p_i) &= \frac{1}{2}\langle x_i + \alpha_i p_i, A(x_i + \alpha_i p_i)\rangle - \langle x_i + \alpha_i p_i, b\rangle \\ &= f(x_i) - \alpha_i \langle p_i, r_i\rangle + \frac{1}{2}\alpha_i^2 \langle p_i, Ap_i\rangle \end{aligned}$$

Minimum of the previous equation can be obtained by derivation with respect to variable α_i and solving the resulting equation.

$$-\langle p_i, r_i\rangle + \alpha_i \langle p_i, Ap_i\rangle = 0 \Leftrightarrow \alpha_i = \frac{\langle p_i, r_i\rangle}{\langle p_i, Ap_i\rangle}$$

Obtained parameters α_i minimize function $f(x{+}\alpha p_i)$. Following inner product trivially holds

$$\langle r_{i+1}, p_i\rangle = 0.$$

We applied the so called line search and found optimal value of function $f(x)$, where argument x was restricted to selected direction. However, the required

[1]Equation $Ax - b = 0$ also has a unique solution when matrix A is nonsingular which yield minimum of function $f(x)$. However, when matrix A is not positive definite, then there is x^* for which ${x^*}^T A x^*$ is below zero. Let us multiply x^* by positive value τ, then function $f(\tau x_*)$ is equal to $\tau^2 {x^*}^T A x^* + \tau {x^*}^T b$. It is clear, when variable τ approaches to a positive infinity, function f falls to negative infinity.

number of steps to obtain the solution is also depended on properly selected search directions.

In the next step we select orthogonal search directions. There are many possibilities to choose search direction but a very convenient choice is to choose then from Krylov spaces $K_n = \{b, Ab, \ldots . A^n b\}$. Let us assume, that direction $p_0, \ldots, p_i$ are mutually conjugate orthogonal.

$$\langle p_i, Ap_j \rangle = 0, \qquad 0 \leq j \leq i$$

and also assume that residuals are orthogonal to previous search direction

$$\langle r_i, p_j \rangle = 0, \qquad 0 \leq j < i.$$

Now, the following induction step can be proved

$$\begin{aligned} \langle r_{i+1}, p_j \rangle &= \langle r_i, p_j \rangle - \frac{\langle r_i, p_j \rangle}{\langle Ap_i, p_i \rangle} \langle Ap_i, p_j \rangle = \\ &= \begin{cases} 0 - \frac{\langle r_i, p_j \rangle}{\langle Ap_i, p_i \rangle} 0 = 0, & \text{if } j < i \\ \langle r_i, p_i \rangle - \langle r_i, p_i \rangle = 0, & \text{if } j = i \end{cases} \end{aligned}$$

It was shown that $i + 1$th residuals are also orthogonal to previous search direction. Now, it is important to find the next search direction, which will also be conjugate orthogonal to previous ones. The last thing, that needs to be decided is how to choose the next search direction. Let method advance to the next direction p_{i+1} by this recurrent formula:

$$p_{i+1} = r_{i+1} + \beta_i p_i$$

From induction assumption follows, that such direction choosing is conjugate orthogonal to previous direction $p_0, \ldots, p_{i-1}$. It only has to be shown, that the same relation holds for direction p_i and p_{i+1}:

$$\langle p_{i+1}, Ap_i \rangle = \langle r_{i+1}, Ap_i \rangle + \beta_i \langle p_i, Ap_i \rangle = 0 \Leftrightarrow \beta_i = -\frac{\langle r_{i+1}, Ap_i \rangle}{\langle p_i, Ap_i \rangle}$$

We have shown, that direction can be selected to be conjugate orthogonal and residuals are also orthogonal to previous direction. Now, it is important to determine the relation among residuals $r_0, \ldots, r_{i+1}$

$$\langle r_{i+1}, r_j \rangle = \langle r_i, r_j \rangle + \alpha_i \langle Ap_i, r_j \rangle$$

Residuals r_i are also mutually orthogonal. Therefore, in exact arithmetic the conjugate gradient method computes the exact solution in a finite number of steps. Because r_{n+1} residual is orthogonal to previous ones, which form space V_n of full rank, so the only possible value is 0. This, so called finite termination property, is in contradiction to the definition of iterative method and in some terms this methods can be classified as a direct solver.

However, in finite arithmetic, especially when matrix A is ill-conditioned, search directions lose their conjugate orthogonality during computation and an exact solution can not be obtained in a finite number of steps. Algorithm 1 ends, until the norm of residual falls bellow the given value.

Various identities allow a number of formulations of the conjugate gradient methods. There are other expressions for α_k and β_k. α_k can be expressed as follows:

$$\alpha_i = \frac{\langle p_i, r_i \rangle}{\langle p_i, Ap_i \rangle} = \frac{\langle r_i + \beta_{i-1} p_{i-1}, r_i \rangle}{\langle p_i, Ap_i \rangle} = \frac{\langle r_i, r_i \rangle}{\langle p_i, Ap_i \rangle}$$

and when expression $Ap_i = -\frac{1}{\alpha_i}(r_{i+1} - r_i)$ is substituted to formulation of β_k we get

$$\beta_i = -\frac{\langle r_{i+1}, Ap_i \rangle}{\langle p_i, Ap_i \rangle} = \frac{1}{\alpha_i} \frac{\langle r_{i+1}, r_{i+1} - r_i \rangle}{\langle p_i, Ap_i \rangle} = \frac{\langle r_{i+1}, r_{i+1} \rangle}{\langle r_i, r_i \rangle}.$$

Algorithm 1 The conjugate gradient method

$r_0 \leftarrow b - Ax_0$
$p_0 \leftarrow r_0$
$k \leftarrow 0$
while $\|r_k\| < tol$ **do**
  $\alpha_k \leftarrow \frac{r_k^T r_k}{p_k^T A p_k}$
  $x_{k+1} \leftarrow x_k + \alpha_k p_k$
  $r_{k+1} \leftarrow r_k - \alpha_k A p_k$
  $\beta_k \leftarrow \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$
  $p_{k+1} \leftarrow r_{k+1} + \beta_k p_k$
  $k \leftarrow k + 1$
end while

1.3.2 The preconditioned conjugate gradient method

The idea of preconditioning is usually based on the multiplication of original system $Ax = b$ by some matrix M, which approximate matrix A. The use of preconditioners can increase the rate of convergence of iterative solution methods considerably.

Consider the conjugate gradient method, where standard inner product $\langle x, y \rangle = x^T y$ is replaced by $\langle x, y \rangle_M = x^T M y$, preconditioned system $M^{-1}A = M^{-1}b$ is solved instead of system $Ax = b$ and its pseudoresiduals $z_k = M^{-1}(b - Ax_k)$ are defined. Then, the standard algorithm can be rewritten in a different form:

$$\alpha_i = \frac{\langle z_i, z_i \rangle_M}{\langle p_i, M^{-1} A p_i \rangle_M} = \frac{\langle r_i, z_i \rangle}{\langle p_i, Ap_i \rangle}$$

and

$$\beta_i = \frac{\langle z_{i+1}, z_{i+1} \rangle_M}{\langle z_i, z_i \rangle_M} = \frac{\langle z_{i+1}, r_{i+1} \rangle}{\langle z_i, r_i \rangle}.$$

Properly defined inner product $\langle x, y \rangle_M$ requires matrix M to be positive definite. Matrix $M^{-1}A$ is positive definite with respect to inner product.

Algorithm 2 The preconditioned conjugate gradient method

$r_0 \leftarrow b - Ax_0$
$z_0 \leftarrow M^{-1}r_0$
$p_0 \leftarrow z_0$
$k \leftarrow 0$
while $\|r_k\| < tol$ **do**
  $\alpha_k \leftarrow \frac{r_k^T z_k}{p_k^T A p_k}$
  $x_{k+1} \leftarrow x_k + \alpha_k p_k$
  $r_{k+1} \leftarrow r_k - \alpha_k A p_k$
  $z_k \leftarrow M^{-1} r_k$
  $\beta_k \leftarrow \frac{r_{k+1}^T z_{k+1}}{r_k^T z_k}$
  $p_{k+1} \leftarrow z_{k+1} + \beta_k p_k$
  $k \leftarrow k + 1$
end while

Chapter 2

Sparse matrices

Matrices, whose majority of elements are equal to zeros, are called sparse. Dense matrices are those, which have only a minority of nonzero elements. The fraction of zero elements in a matrix is called the sparsity (density).

Used matrix operation is related to its matrix sparsity. If the matrix is sparse then the time complexity of the used operation should be depended on the number of nonzero elements instead of the number of rows or columns, like the time complexity of a lot of dense algorithms are. So, it is often necessary to use specialized algorithms and data structures. However, when the matrix is dense, sparse data structures and algorithm are usually less efficient and memory saving.

2.1 Sparse systems of linear equations

Sparse systems of linear equations are not a unique coincident but it can be shown that they naturally rise from many applications. The significant sparse structure of such a system allows us to solve larger instances of decomposed problem, which often leads to greater accurateness of the final solution.

Finite element method. Finite element method is a numerical technique for finding approximate solutions to boundary value problems. It is used frequently for the numerical modelling of physical systems in a wide variety of engineering disciplines, e.g., electromagnetism, heat transfer, and fluid dynamics.

In this section, we show dicretization of one boundary value problem described in paper[1]. The Poisson problem (P) is defined as follows: Given $f \in L^2(\Omega)$, $u_D \in H^1(\Omega)$, and $u_N \in L^2(\Gamma_N)$, we are searching for the solution $u \in H^1(\Omega)$, which satisfies the Poisson equation

$$-\Delta u = f \quad \text{in } \Omega, \qquad u = u_D \quad \text{on } \Gamma_D, \qquad \frac{\partial u}{\partial n} = u_N \quad \text{on } \Gamma_N,$$

where $\partial\Omega = \Gamma_D \cup \Gamma_N$ is polynomial boundary with Dirichlet conditions on closes subset Γ_D and Neumann boundary conditions on remaining part Γ_N.

The weak formulation of the boundary value problem (P) is then obtained by the multiplication of equation $-\Delta u = f$ with $w \in H^1_D(\Omega) := \{w \in H^1(\Omega) | w = 0 \text{ on } \Gamma_D\}$ and integration over Ω:

$$-\int_\Omega \Delta u \cdot w \,\mathrm{d}x = \int_\Omega f \cdot w \,\mathrm{d}x.$$

If we integrate by parts using a form of Green's identities and incorporate inhomogeneous Dirichlet conditions through the decomposition $v = u - u_D$ (satisfies $v = 0$ on Γ_D), then we gets the weak formulation of the Poisson problem P which reads: Find $v \in H^1_D(\Omega)$ such that

$$\int_\Omega \nabla v \cdot \nabla w \,\mathrm{d}x = \int_\Omega f w \,\mathrm{d}x + \int_{\Gamma_N} u_N w \,\mathrm{d}s - \int_\Omega \nabla u_D \cdot \nabla w \,\mathrm{d}x, \quad w \in H^1_D.$$

The weak formulation of the Poisson equation can be discretized using the standard Galerkin method. The solution space of the numerical solution U is restricted to a finite dimensional subspace S of $H^1(\Omega)$. Accordingly $U_D \in S_D := S \cap H^1_D$ approximates u_D on Γ_D.

If we assume that $(\phi_1, \ldots, \phi_N)$ is a basis of the finite dimensional space S, and $(\phi_{i_1}, \ldots, \phi_{i_M})$ is a basis of $S_D := S \cap H^1_D$, where $I = \{i_1, \ldots, i_M\} \subseteq \{1, \ldots, N\}$ is an index set. Then, the discretized problem P_S can be written as: Find $V \in S_D$ such that

$$\int_\Omega \nabla V \cdot \nabla \phi_j \,\mathrm{d}x = \int_\Omega f \phi_j \,\mathrm{d}x + \int_{\Gamma_N} u_N \phi_j \,\mathrm{d}s - \int_\Omega \nabla U_D \cdot \nabla \phi_j \,\mathrm{d}x, \quad j \in I.$$

If we now make a series expansion of V and U_D in terms of ϕ_k

$$V = \sum_{k \in I} x_k \phi_k \quad \text{and} \quad U_D = \sum_{k=1}^{N} U_k \phi_k,$$

then above equations can be finally simplified to a system of linear equations

$$Ax = b.$$

The stiffness matrix $A = (A_{jk})_{j,k \in I}$ and the right-hand side $b = (b_j)_{j \in I}$ are defined as

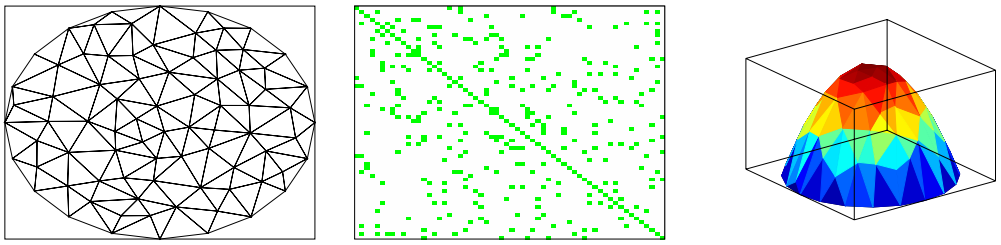
$$A_{jk} = \int_\Omega \nabla \phi_j \cdot \nabla \phi_k \,\mathrm{d}x,$$

$$b_j = \int_\Omega f \cdot \phi_j \,\mathrm{d}x + \int_{\Gamma_N} u_N \phi_i \,\mathrm{d}s - \int_\Omega \nabla \phi_k \cdot \nabla \phi_j \,\mathrm{d}v.$$

The stiffness matrix is symmetric and positive definite. Thus, linear system $Ax = b$ has exactly one solution $x \in \mathbb{R}^M$, which gives the Galerkin solution

$$U = U_D + V = \sum_{k=1}^{N} U_j \phi_j + \sum_{k \in I} x_k \phi_k.$$

Figure 2.1: Finite element process for $-\Delta u = 4$



(a) A triangulation $\mathcal{T}$ (b) A sparse matrix (c) The result $1 - x^2 - y^2$

The important question is a sparsity of stiffness matrix, which depend on the selection of a basis $(\phi_1, \ldots, \phi_N)$ of the finite dimensional space S.

The finite element method requires the discretization of the spatial domain Ω with finite elements.

It can be covered by a regular triangulation $\mathcal{T}$ of triangles and quadrilaterals.

When a regular triangulation $\mathcal{T}$ has been generated for the domain Ω, the space S of the numerical solution U has to be defined. A common choice of basis functions for the spline spaces S and S_D are tent functions, e.q.,the piecewise linear functions such, that for each vertex v_i of $\mathcal{T}$ $\phi_i(v_i) = 1$ and $\phi_i(v_j) = 0$ if $j \neq i$.

Figure 2.1 demonstrates the finite element process of solving simple variant of Poisson equation $-\Delta u = 4$ with zero Dirichlet boundary conditions. Matrix A is sparse by choice of basis functions $(\phi_1, \ldots, \phi_N)$.

2.1.1 Fill-in

Direct methods, like LU decomposition, usually process matrix A and the result is modified matrix A. The fill-in of a matrix are those elements which were zero at the beginning but during execution were changed to nonzero values. Naturally, it is desired to choose such algorithms which reduce fill-in as much as possible.

The second option is to use smart rows and columns reordering, which minimize fill-in of result matrix. An example in book [10] demonstrates how reordering is important. If one tries to apply Gauss elimination on two reordered but equivalent systems of linear equations in figure 2.2, then computed fill-ins will be minimal and also maximal possible. It hasn't been mentioned, that reordering does not affect the solution in exact arithmetic.

Figure 2.2: Ordering of same system

$$\begin{pmatrix} a_{11} & & & & a_{15} \\ & a_{22} & & & a_{25} \\ & & a_{33} & & a_{35} \\ & & & a_{44} & a_{45} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{pmatrix} \quad \begin{pmatrix} a_{55} & a_{54} & a_{53} & a_{52} & a_{51} \\ a_{45} & a_{44} & & & \\ a_{35} & & a_{33} & & \\ a_{25} & & & a_{22} & \\ a_{15} & & & & a_{11} \end{pmatrix}$$

Let us focus only on ordering, which reduces fill of LU based algorithm. Such ordering can be divided into three groups according their criterion.

- Let β denote distance of the farthest element from the diagonal. It can be proven, that none of the elements of the result factor will lay above that distance. Method, which minimize parameter β, also reduces fill-in of resulting factors.
- It can be shown, that fill-in of result L or U factors are strictly bounded by the profile of matrix A. The second kind of method specializes in ordering matrix A so, that the profile of matrix A bounds fill of factor as much as possible.
- The last method does not focus on minimizing the profile of matrix A. But it internally simulates the process of selected decomposition and selects such ordering, which minimize fill-in of resulting factors.

2.2 Sparse matrix representation

Smart sparse matrix representation could allow the matrix to be efficiently modified and supports efficient sparse matrix operation. Sadly, those two criteria are often in contradiction and both can not be fulfilled.

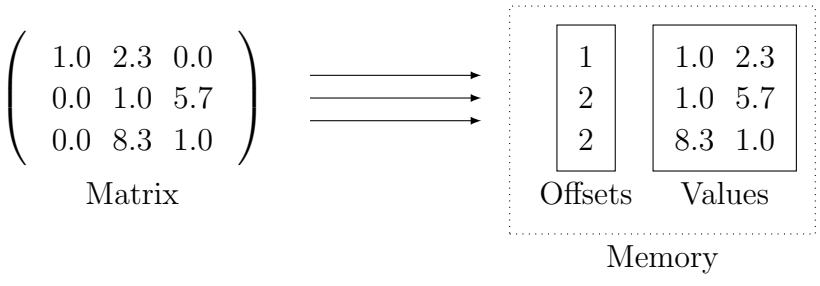
There are many of such used formats in practice, but only a few of them, the most common, will be described in this section.

2.2.1 Diagonal format

Diagonal representation is one of the simplest to implement and it is suitable to store diagonal or some of band matrices. Realization consists of two arrays. The first array of size RxN, where N is the number of rows and R is the number of elements per row. The second array of size N stores the first column of each row. The element's column index is determined by its position in the row and the starting column of the row. Therefore, the order of the element is given and couldn't be changed. Figure 2.3 shows how given matrix is stored in memory in diagonal representation.

Described matrix representation is memory economic, because rows and column indices are stored implicitly. Unfortunately, not all of the matrix can be efficiently represented by diagonal format. If a few elements lay out of the main diagonal or band, then those elements can be represented by a different matrix format. The final matrix is represented by sums of those two matrices.

Figure 2.3: Diagonal format



2.2.2 Compressed rows format

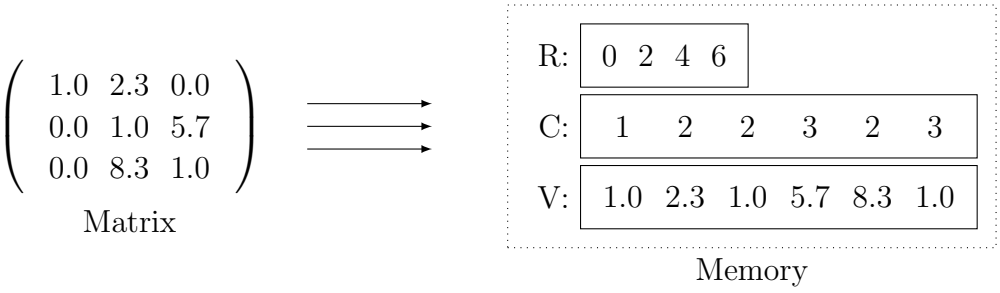
Compress rows format is a general scheme able to represent any type of matrix. Despite its variability, adding new elements or removing old ones isn't effective. Implementation consists of three vector arrays:

- The array I stores column indices of size nnz.
- The array V stores values of size nnz.
- The last array stores beginnings of rows of size $N + 1$. Every i-th element except the last one points on start of i-th rows. The last element is always equal to $n+1$. The first element is always zero, however it is better to store it for convenient implementation.

Representation of matrix in memory is displayed in Figure 2.4. The order of the elements in one row is unimportant. However, in some application it could be beneficial to order elements by theirs column index.

The proper parallel implementation may be complicated. Dividing work by rows couldn't be sufficient, because rows may have various lengths. It should be considered trying a different parallel scheme.

Figure 2.4: Compress rows format



2.2.3 Coordinate format

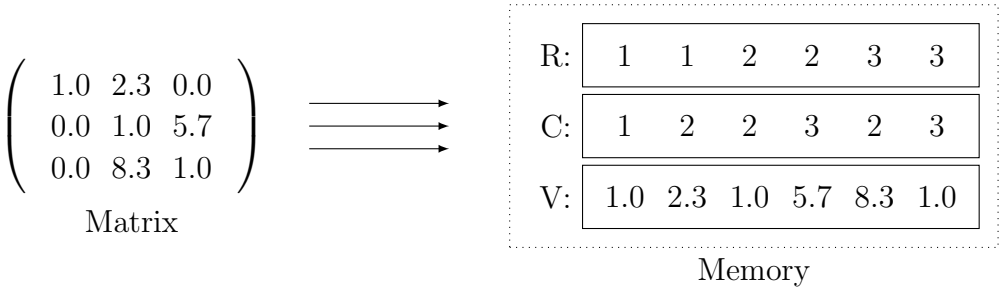
Coordinate matrix format is very versatile and it doesn't rely on the structure of a represented matrix. Each nonzero element is represented by its own value, row, and column index. Therefore the implementation is represented by three arrays of length nnz. The first array I stores row indices, the second array J stores columns indices and the last array is filled with values. Figure 2.5 shows how given matrix is stored in memory in coordinate representation.

The order of the elements is unimportant within the whole matrix. This is very useful, because the matrix can be constructed by one run. However, sometimes it could be beneficial to order elements by some key, like by their column index.

The main advantage from the view of this Thesis is efficient parallel implementation of some matrix operations, like matrix products are. Coordinate representation can be easily divided into equal sized blocks and distributed among computation processors.

Because every element bears its own row and column index, coordinate matrix format is less memory saving than compress row format. The only exception is matrices, which have less non-zero element than rows.

Figure 2.5: Coordinate format



2.2.4 Block format

Block format is not matrix representation at all. It is the representation of the element itself. Normally one element is presented by one value, but there one element is presented by blocks of elements. The sizes of all blocks are equal and they are not verisaiable. Block representation is suitable for all of the described sparse matrix representations, but it is beneficial to use it with compress rows format.

Block element representation is not memory economic. Block may contain a lot of zero values. However, it is extremely useful in some application, where those blocks are created naturally.

Access to values inside block is continuous, so it is useful to use vector computer or vector instruction.

2.2.5 Matrix-vector products

The operation of multiplication of matrix and vector is the basis of many numerical methods. Its effective implementation could play a key role in the overall performance. Sparse product should be computed in linear time to the number of nonzero elements with any used sparse matrix representation. This is an important property, because dense matrix and vector product can be computed with quadratic complexity to number of rows.

Let us suppose, that sparse matrix A of size $m \times n$ is formed by sparse vector rows $a_1, \ldots, a_m$ and every sparse vector a_i contains at most n pairs of column and value, whose represent nonzero elements. Values vectors $x = (x1, \ldots, x_n)$ and $y = (y_1, \ldots, y_m)$ are dense to allow random access. Then, the matrix product for most representations could be written as:

Algorithm 3 Right sparse matrix-vector product $y = Ax$.

for $i = 1 \rightarrow m$ **do**
  for all $(column, value) \in a_i$ **do**
    $y_i \leftarrow y_i + value * x_{column}$
  end for
end for

Every value of nonzero element of matrix A is read only once. Then, the read value is multiplied by x_j, where j is its adjoining column. The result is added to y_i, where i is current row. If all operations are counted over all elements (nz), than

the total number of arithmetic operations is equal to $2 * nz$ and the total number of memory access is $4 * nz$. The difference between the number of arithmetic and memory operations is interesting, because one arithmetic operation requires two memory accesses. However, hardware memory caching could reduce that ratio.

Sometimes it could be useful to compute vector and sparse matrix product $x = y^T A$. It can be easily transformed to previous variant by transposing matrix A. However, depending on used matrix representation matrix transposing may be very inefficient. Fortunately, those products can be computed directly without need to transpose matrix A.

Algorithm 4 Left sparse matrix-vector product $x^T = y^T A$.

for $i = 1 \rightarrow n$ **do**
  for all $(column, value) \in a_i$ **do**
    $x_{column} \leftarrow x_{column} + value * y_i$
  end for
end for

Chapter 3

Preconditioners

The rates of convergence of many iterative methods for solving $Ax = b$ depends on the condition number $\kappa(A)$ and/or the distribution of the eigenvalues of A. Therefore, it is a natural idea to transform the original linear system so that the new system has the same solution (or a solution from which the original one is easily recovered) and the transformed matrix has a smaller condition number and/or a better distribution of the eigenvalues.

$$M^{-1}Ax = M^{-1}b$$

We could also transform the system by right preconditioning as

$$AM^{-1}y = b, Mx = y.$$

if the matrix A is symmetric positive definite, we might want to keep the transformed system symmetric. In this case, we suppose M to be symmetric and positive definite and instead of above equation, we use

$$M^{-\frac{1}{2}}AM^{-\frac{1}{2}}y = M^{-\frac{1}{2}}b,$$

The meaning of the preconditioners is to allow us to solve system of linear equation more conveniently. Preconditioner doesn't compute the solution by itself but it will affect the convergence rates of the used iterative method, in the majority of cases conjugate gradient method.

From a certain point of view precoditioner may look like something between direct and iterative solvers. The solution is obtained by modified iterative method, but preconditioner itself is computed by means of direct solver.

Most existing preconditioners can be classified as being either the implicit kind or the explicit kind, depending on their realization in iterative solver.

Implicit A precondioner is implicit if its application, within each step of the chosen iterative method, requires the solution of a linear system. The most significant example of such preconditioner are those, which are based on incomplete LU (LL^T) decomposition. Applying the preconditioner requires the solution of two sparse triangular systems (the forward and backward sweeps). It very important to mention, that forward and backward sweep can be hardly done in parallel.

Explicit If an approximation of inverse matrix A^{-1} is known then the preconditioning operation reduces to forming one (or more) matrix-vector product. Such preconditioners are called explicit.

3.1 The incomplete Cholesky factorization

The basic idea in the point of the Cholesky preconditioner is to modify Cholesky factorization to allow fill-ins at only a restricted set of positions in the L factor.

Let $A^{(k)} = [a_{ij}^{(k)}]$ denote the matrix at the kth stage, where $A^{(1)} = A$ and $a_{kk}^{(k)}$ is the current pivot entry. We know, that the complete Cholesky process always satisfies $a_{kk}^{(k)} \neq 0$.

In Chapter 1 we presented The outer product variant of Cholesky decomposition. Now, we use a less explanatory but more convenient equivalent way to compute Cholesky decomposition:

$$A^{(k+1)} = A^{(k)} - l_k l_k^T, \quad \text{where} \quad l_i = (0, \ldots, 0, a_{kk}^{(k)}, \ldots, a_{nk}^{(k)}) / \sqrt{a_{kk}^{(k)}}.$$

The entries $a_{ij}^{(k+1)}$ of the matrix $A^{(k+1)}$ are defined as follows:

$$a_{ij}^{(k+1)} = a_{ij}^{(k)} - \frac{a_{ik}^{(k)} a_{kj}^{(k)}}{a_{kk}^{(k)}}.$$

However, in the incomplete factorization method, nonzero entries $a_{ij}^{(k+1)}$ are accepted only if $\{i, j\} \in \mathcal{S}^{(k)}$, where $\mathcal{S}^{(k)} \subseteq \mathcal{S}$ is an index set that at stage k defines the position where nonzero entries are accepted in positions where i and $j \geq k + 1$.

Algorithm 5 The incomplete Cholesky factorization algorithm.

$A^{(1)} \leftarrow A$
for $k = 1 \rightarrow n$ **do**
  for $i = k \rightarrow n$ **do**
    if $\{i, k\} \in \mathcal{S}$ **then**
      for $j = k \rightarrow n$ **do**
        if $\{i, j\} \in \mathcal{S}$ **then**
          $a_{ij}^{(k+1)} \leftarrow a_{ij}^{(k)} - \frac{a_{ik}^{(k)} a_{kj}^{(k)}}{a_{kk}^{(k)}}$
        end if
      end for
    end if
  end for
end for

$$L \leftarrow (l_{ik}) = \begin{cases} a_{ik}^{(k)} / \sqrt{a_{kk}^{(k)}}, & \text{if } \{i, k\} \in \mathcal{S} \\ 0, & \text{if } otherwise \end{cases}$$

If $a_{kk}^{(k)} \neq 0$, $k = 1, 2, \ldots, n-1$, we can continue the algorithm until the final stage, where $k = m - 1$, to form an approximate or incomplete matrix

factorization LL^T, where L is a lower triangular matrix of order $n \times n$ consist of entries $a_{ik}^{(k)}/\sqrt{a_{kk}^{(k)}}$ if $\{i,k\} \in \mathcal{S}$ or zero otherwise.

The question remains how to choose the sparsity set $\mathcal{S}$. A commonly used strategy is to do $\mathcal{S}$ by:

$$\mathcal{S} = \{(i,j)|a_{ij} \neq 0\}.$$

In case such strategy was used, this algorithm has constructed a complete decomposition of $M = A + R$, where the matrix $R = [r_{ij}]$ is a priori unknown with $r_{ij} = 0$ if $\{i,j\} \in \mathcal{S}$. This strategy have predictable memory requirements but are independent of the entries of A because the dropped elements depend only on the structure of A.

The interesting question is to know the conditions under which such a decomposition is feasible. It is shown in book [2, p. 259], that if A is an M-matrix, then the incomplete Cholesky factorization exists for any predetermined sparsity pattern $\mathcal{S}$, and gives a symmetric positive definite matrix if it is symmetric positive definite.

There are also different kinds of fill strategies. The numerical fill strategies include nonzero elements in the incomplete factor if they are larger than some threshold parameter. For example, drops $a_{ij}^{(k)}$ during the kth step if

$$|a_{ij}^{(k)}| \leq \tau\sqrt{|a_{ii}^{(k)}a_{jj}^{(k)}|},$$

where τ is the drop tolerance. Although this definition makes more sense mathematically, it is harder to implement in practice, since the amount of storage needed for the factorization is not easy to predict[13]. If τ is large, then L will have few nonzero elements but will also tend to be a poor preconditioner.

Applying the incomplete Cholesky preconditioner $z = M^{-1}r = (LL^T)^{-1}r$ requires two triangular solves $Lt = r$ and $L^Tz = t$ which makes this preconditioner to be considered implicit. Although computing triangular solve in parallel may seem difficult, it is possible. A way how to vectorize the preconditioning part can be found in book [10, p. 215] or in book [3, p. 43].

3.2 Approximations of inverse matrices

In this section we discuss a method for computing an incomplete factorization of the inverse of a symmetric positive definite (SPD) matrix $A \in \mathbb{R}^{n\times n}$. The resulting factorized sparse approximate inverse is used as an explicit preconditioner for the solution of $Ax = b$ by the preconditioned conjugate gradient (PCG) method.

3.2.1 The inverse factorization

Let A be a n by n SPD matrix, then a factorization of A^{-1} can readily be obtained from a set of conjugate directions $z_l, z_2, \ldots, z_n$ for A. If

$$Z = [z_1, z_2, \ldots, z_n]$$

is the matrix whose ith column is z_i, we have

$$Z^T AZ = D = \begin{pmatrix} p_1 & 0 & \dots & 0 \\ 0 & p_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & p_n \end{pmatrix}, \quad \text{where} \quad p_i = z_i^T A z_i \neq 0$$

Once matrices D and Z are obtained, inverse of matrix A can be expressed as

$$A^{-1} = ZD^{-1}Z^T.$$

Now, let us use a standard form of Gram–Schmidt process with inner product defined as $\langle u, v\rangle = \langle u, Av\rangle = u^T Av$. From assumption matrix A is SPD, so such an inner product is well defined. Linearly independent vectors $z_i^{(0)} = e_i$, where e_i is ith unit vector, are transformed to conjugate orthogonal vectors $z_1, z_2, \dots, z_n$. They can be computed by recurrent formula:

$$z_j^{(i)} = z_j^{(i-1)} - \frac{\langle z_j^{(i-1)}, a_i\rangle}{\langle z_i^{(i-1)}, a_i\rangle} z_i^{(i-1)}, \ (j < i)$$

Inner products, in the above recurrent formula, are written in compact form. Because vectors $z_1, z_2, \dots, z_n$ are conjugate orthogonal, the following forms are equal in exact arithmetic:

$$\langle z_j^{(i-1)}, a_i\rangle = \langle z_j^{(i-1)}, Az_i^{(i-1)}\rangle.$$

A way, to compute ith orthogonal vector z_i, was described. From now, an inverse factorization algorithm can be introduced.

Algorithm 6 The Inverse Factorization Algorithm

$z_i^{(0)} = e_i$
for $i = 1 \to n$ **do**
  for $j = i \to n$ **do**
    $p_j^{(i-1)} \leftarrow \langle a_i, z_j^{(i-1)}\rangle$
  end for
  for $j = i + 1 \to n$ **do**
    $z_j^{(i)} \leftarrow z_j^{(i-1)} - \frac{p_j^{(i-1)}}{p_i^{(i-1)}} z_i^{(i-1)}$
  end for
end for
$z_i = z_i^{(i-1)}$
$p_i = p_i^{(i-1)}$
$Z = [z_1, z_2, \dots, z_n]$
$D = diag(p_1, p_2, \dots, p_n)$

It is good to mention, that A mustn't be explicitly stored, because it only required the capability of forming inner products involving the rows of A. This is an attractive feature for cases where the matrix is only implicitly given as an operator. Once Z and D are available, the solution of $Ax = b$ can be computed as

$$x^* = A^{-1}b = ZD^{-1}Z^Tb = \sum_{i=1}^{n}(\frac{z_i^Tb}{p_i})z_i$$

Computed inverse of matrix A should be sparse. Complete process may produce Z matrices with high amount of fill-in, which can be considered as dense. One possible solution is to ignore elements produced during computation of z vectors, whose values fall below a specified tolerance or they are located outside specified positions. This procedure is called dropping.

If the incomplete inverse factorization process is successfully completed, one obtains a unit upper triangular matrix Z and a diagonal matrix D with positive diagonal entries such that

$$M^{-1} = ZD^{-1}Z^T \approx A^{-1}$$

is a factorized sparse approximate inverse of A.

Relation to LDL^T Decomposition

Now, we will discuss the relation of inverse factorization and LDL^T decomposition. This relation should be examined, because some properties of LDL^T decomposition may be useful in inverse factorization as well.

Let us denote matrix $P = [p_{ij}]$ of coefficients computed by the algorithm 6. Elements of matrix are equal to

$$p_{ij} = \begin{cases} p_j^{(i-1)} & \text{if } i < j \\ 0 & \text{otherwise} \end{cases}.$$

Computation decomposition of an inverse matrix can be viewed as QR decomposition of eye matrix with rank n. Then can be shown, that the transposed matrix P and matrix D forming LDL^T decomposition of matrix A:

$$A = IAI = P^TZ^TAZP = P^TDP.$$

Matrix P is incrementally constructed during factorization. The relation to LDL^T decomposition exposes another way to compute inverse factorization. Vectors z_i are not constructed from preceding vectors z_j but they are computed directly from matrix P. This computation method is generally less expensive, but it can be done with difficulty in parallel.

LDL^T reordering can be used as well when inverse decomposition is computed. Reordering, which reduces fill-in of LL^T decomposition, directly affects computation time of inverse decomposition. It is caused by the decreasing number of vector additions needed to be done, because corresponding elements are zero. Fill-in of sparse inverse factors are affected indirectly, but not as significantly as computation time.

3.2.2 The inverse factorization of nonsymmetric systems

The algorithm described above required matrix A to be symmetric and positive define. The algorithm in paper [7] works on a similar idea, but it constructs two

sets of A-biconjugate vectors $\{z_i\}_{i=1}^n$ and $\{w_i\}_{i=1}^n$ for a given nonsingular matrix $A \in \mathbb{R}^{n\times n}$. If

$$Z = [z_1, z_2, \ldots, z_n]$$

is the matrix whose ith column is z_i and

$$W = [w_1, w_2, \ldots, w_m]$$

is the matrix whose ith column is w_i, then

$$W^T AZ = D = \begin{pmatrix} p_1 & 0 & \ldots & 0 \\ 0 & p_2 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & p_n \end{pmatrix}, \text{ where } p_i = w_i^T A z_i \neq 0.$$

Once matrices W, D, and Z are obtained, the inverse of matrix A can be expressed as

$$A^{-1} = ZD^{-1}W^T = \sum_{i=1}^{n} \frac{z_i w_i^T}{p_i}.$$

The algorithm runs basic factorization of inverse matrix two times, one time computes factorization Z and D_1 of matrix A and the second time computes factorization W and D_2 of its transpose. Because matrix A is nonsymmetric, neither set $\{z_i\}_{i=1}^n$ nor set $\{w_i\}_{i=1}^n$ is conjugate orthogonal. In exact arithmetic, computed diagonal matrices D_1 and D_2 are equal, so it doesn't depend which one is used.

Once Z, W, and D are available, the solution of $Ax = b$ can be computed as

$$x = A^{-1}b = ZD^{-1}W^T b = \sum_{i=1}^{n} (\frac{w_i^T b}{p_i}) z_i$$

This algorithm was mentioned to show that there are also explicit preconditioners for nonsymmetric system of linear equations. But this Thesis is primary focused on symmetric and positive definite systems.

3.2.3 The stabilized inverse factorization

The inverse factorization may fail in same cases. If one operation of original algorithm is expanded, one can get the stabilized variant. First we need to take a close look at the mechanism of breakdown. We begin by writing down the explicit formula for the p_j's:

$$p_j^{(i-1)} = \langle a_i, z_j^{(i-1)} \rangle = \sum_{l=1}^{i-1} a_{il} z_{lj}^{(i-1)} + a_{ij} \quad (i \leq j \leq n).$$

When A is an M-matrix, it is easily seen by induction that all the $z_{lj}^{(i-1)}$ are nonnegative.

The way to avoid nonpositive pivots is simply to recall that in the exact conjugate orthogonalization process, the p_i's are the diagonal entries of matrix D which satisfies the matrix equation

$$Z^T AZ = D;$$

hence for $i = 1, \ldots, n$

$$p_i = \langle z_i, Az_i \rangle > 0$$

since A is SPD and $z_i \neq 0$. (Recall that the ith entry of z_i is equal to 1.) In the previous section, the following identity was described:

$$p_i = \langle z_i, Az_j \rangle = \langle a_i, z_j \rangle$$

This identity follows immediately from the fact that vectors z_i are conjugate orthogonal. Clearly, it is more economical to compute the pivots using the expression on the right-hand side of above formula rather than that in the middle. However, when dropping is applied or inexact arithmetic is used, resulting z_i vectors loss their conjugate orthogonality and the above formula becomes invalid. Then for some matrices one can have

$$\langle a_i, z_j \rangle \ll \langle z_i, Az_j \rangle$$

with the concomitant possibility of breakdowns.

One idea is to compute p_j in the standard way, until the process fails and then try to use more expensive and appropriate bilinear form. However, in paper [5] it was mentioned that a robust algorithm requires that the p_j be computed using the bilinear form $z_j^T Az_i$ throughout the entire AINV process, for $i = 1, \ldots, n$. It could have been very expensive to compute p_j that way but they also point out that there is a more efficient way. In the latter case, we have

$$p_j = \langle v_i, z_j \rangle \text{ where } v_i = Az_i$$

Because the vector v_i has already been computed as part of the calculation of p_i, this approach may be as effective as to compute p_j by nonstabilized way $p_i = \langle a_i, z_j \rangle$. However, a skipping addition of z_j vector depends on zero value of corresponding p_j, which can be nonzero with grater chance in the case of used stabilized process, because generally v_i^T may contains more nonzero elements than a_i^T. This could cause possible slow downs in the stabilized process over the regular one.

Algorithm 7 The Stabilized Inverse Factorization Algorithm

$z_i^{(0)} = e_i$
for $i = 1, \ldots, n$ **do**
  $v_i \leftarrow A z_i^{(i-1)}$
  for $j = i \rightarrow n$ **do**
    $p_j^{(i-1)} \leftarrow \langle v_i, z_j^{(i-1)} \rangle$
  end for
  for $j = i + 1 \rightarrow n$ **do**
    $z_j^{(i)} \leftarrow z_j^{(i-1)} - \frac{p_j^{(i-1)}}{p_i^{(i-1)}} z_i^{(i-1)}$
  end for
end for
$z_i = z_i^{(i-1)}$
$p_i = p_i^{(i-1)}$
$Z = [z_1, z_2, \ldots, z_n]$
$D = diag(p_1, p_2, \ldots, p_n)$

Obviously, Algorithms 6 and 7 are mathematically equivalent. However, when dropping is applied in the second inner loop after vector sums then the incomplete process leads to a more reliable approximate inverse procedure.

A very important question is the cost of SAINV. At first sight it might look very expensive to compute the preconditioner on the basis of Algorithm 7, which requires the computation of matrix-vector products $v_i = Az_i$. It should be noticed that due to used dropping and reordering vectors z_i are sparse so the matrix-vector products can be reduced to forming a linear combination of a few columns of A, namely, those which correspond to nonzero entries in z_i.

3.2.4 Comparison with other methods

The purpose of the inverse factorization is to speed up rates of convergence of PCG. There exist various methods, which can do the same. In practice, it is important to select the most appropriate method for a given problem. The concern of this Thesis is not to compare the inverse factorization with similar methods. Such comparison can be found in various papers, for example [6, 5].

Discussion, when to use regular inverse decomposition over stabilised, is not important from the point of view of this Thesis. But, even though the stabilized inverse factorization is more expensive to compute, it is interesting because it can benefit from modern hardware more that its simpler version.

Chapter 4

Block diagonal matrices

A block diagonal matrix, also called a diagonal block matrix, is a square diagonal matrix in which the diagonal elements are square matrices (submatrices) of any size, and the off-diagonal elements are zero matrices. A block diagonal matrix A has the form

$$A = \begin{pmatrix} A_1 & 0 & \dots & 0 \\ 0 & A_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & A_n \end{pmatrix}$$

where A_i is a square matrix. Trivially, a block diagonal matrix A is symmetric and positive definite if and only if its submatrices A_i are also symmetric and positive definite.

The attractive feature of block diagonal matrices is, that computed inverse factorizations are also block diagonal. The inverse factorization of block diagonal matrix A can be written as follows

$$\begin{pmatrix} Z_1^T & 0 & \dots & 0 \\ 0 & Z_2^T & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & Z_n^T \end{pmatrix} A \begin{pmatrix} Z_1 & 0 & \dots & 0 \\ 0 & Z_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & Z_n \end{pmatrix} = \begin{pmatrix} D_1 & 0 & \dots & 0 \\ 0 & D_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & D_k \end{pmatrix},$$

where $Z_i^T A_i Z_i = D_i$ is factorization of ith submatrix of A.

The maximal fill-in of factor Z is limited by the sizes of submatrices A_i. Moreover, a fill-in reducing reordering may be applied to each submatrix A_i individually.

4.1 Issue of parallelism

Block diagonal matrices may be interesting from the point of view of parallel computing. Parallel processing of block diagonal a matrix A by many algorithms may be realized by concurrent processing its submatrices A_i.

The inverse factorization. Block diagonal structure of matrix A exposes a new way to compute inverse factorization in parallel by executing inverse

factorization of submatrices A_i simultaneously. The resulting factors Z and D consist of the results of inverse factorization of submatrices A_i.

The conjugate gradient method. Applying an explicit preconditioner in iteration of the preconditioned conjugate gradient methods is done by forming one or more matrix-vector products. The computation of matrix-vector product can be divided into independent vector-vector products, which are then solved concurrently.

The algorithm of approximate inverse factorization decomposes a matrix A into two matrices with full rank. The precondtioner phase of PCG is realized by forming three (two in practice, because matrix D is diagonal) matrix-vector products which may require at least two synchronization points in the case of parallel computation of product. Parallel realization, which divides work by submatrices A_i, may omit at least one synchronization point.

Unfortunately, there are a lot of not block diagonal matrices, which rise from real application. Even through they can not be transformed by simple operation as rows and columns orderings into desired structure.

4.2 Graph partitioning

The graph partitioning problem is defined as follows: Given a graph $G = (V, E)$ with $|V| = n$, partition V into k parts (subsets) $V_1, V_2, \ldots, V_k$ such that $V_i \cap V_j = \emptyset$ for $i \neq j$, $|V_i| = n/k$, and the number of the edges with endpoints in different partitions is minimized. Given a partitioning P, the number of edges whose incident vertices belong to different partitions is called the edge-cut of the partitioning.

It is difficult to obtain the exact solution, because graph partition is a hard problem. Therefore a heuristic, which usually returns only approximation of the solution, has to be used. Available heuristic can be classified as either local or global. Well known local methods are the Kernighan–Lin algorithm, and Fiduccia-Mattheyses algorithms, which rely on arbitrary initial partitioning of the vertex set, which can affect the final solution quality. On the other side, global approaches rely on properties of the entire graph rather than an arbitrary initial partition. Another way to obtain approximate solution is to use multilevel algorithm.

The basic idea of a multilevel partitioning algorithm is to reduce the given problem into a smaller and easily processable one. The graph $G = (V, E)$ is first coarsened down to a small number of vertices, a k-way partitioning of this much smaller graph is computed, and then this partitioning is projected back toward the original graph by successively refining the partitioning at each intermediate level.

The following description is the essential knowledge, how multilevel partitioning algorithm works. It is important to be able to properly use an external graph partitioning library. As that library I preferred to use METIS [11] developed by George Karypis.

4.2.1 Multilevel graph bisection

Formally, a multilevel graph bisection algorithm works as follows: consider a weighted graph $G_0 = (V_0, E_0)$, with weights both on vertices and edges. A multilevel graph bisection algorithm consists of the following three phases.

Coarsening phase. During the coarsening phase, a sequence of smaller graphs $G_i = (V_i, E_i)$, is constructed from the original graph $G_0 = (V_0, E_0)$ such that $|V_i| < |V_{i-1}|$. Given a graph $G_i = (V_i, E_i)$, a coarser graph can be obtained by collapsing adjacent vertices. The next level coarser graph G_{i+1} is constructed from G_i by finding a matching of G_i and collapsing the vertices being matched into multinodes. The unmatched vertices are simply copied over to G_{i+1}.

For this reason, maximal matchings are used to obtain the successively coarse graphs. A matching is maximal if any edge in the graph that is not in the matching has at least one of its endpoints matched. Note that depending on how matchings are computed, the number of edges belonging to the maximal matching may be different.

The coarsening phase ends when the coarsest graph G_m has a small number of vertices or if the reduction in the size of successively coarser graphs becomes too small.

Partitioning phase. The second phase of a multilevel k-way partitioning algorithm is to compute a k-way partitioning P_m of the coarse graph $G_m = (V_m, E_m)$ such that each partition contains roughly $|V_0|/k$ vertex weight of the original graph.

One way to produce the initial k-way partitioning is to keep coarsening the graph until it has only k vertices left. These coarse k vertices can serve as the initial k-way partitioning of the original graph.

Uncoarsening phase. During the uncoarsening phase, the partitioning P_m of the coarser graph G_m is projected back to the original graph, by going through the graphs $G_{m-1}, G_{m-2}, \ldots, G_1$. Even if the partitioning of G_i is at a local minima, the projected partitioning of G_{i-1} may not be at a local minima.

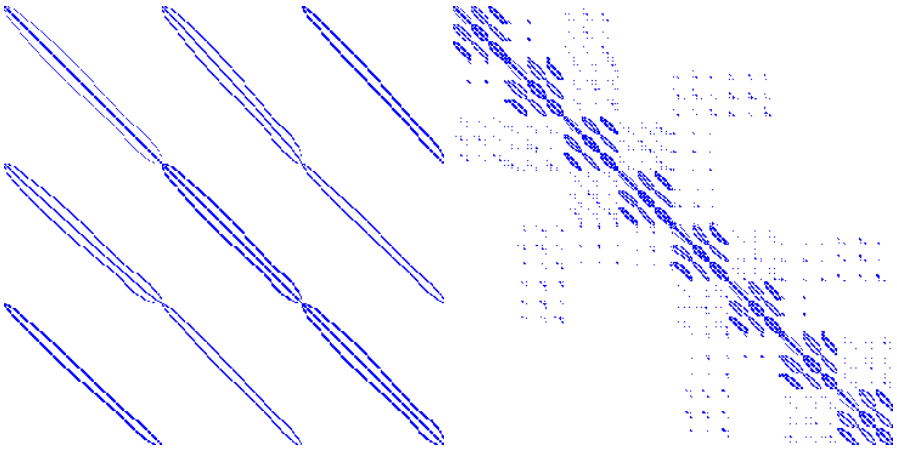
4.3 Relation of graphs and matrices

A conversion from matrix to graph is mainly interesting from the point of view of row and/or column orderings. Many fill-in reducing orderings are based on algorithm from graph theory which operate on graphs instead of matrices.

Any general and unstructured m by n matrix can be converted to undirected graph $G = (V, E)$, where vertex set $V = (r_1, \ldots, r_m, c_1, \ldots, c_m)$ consists of its rows and columns and edge set is $E = \{(r_i, c_j) | a_{ij} \neq 0\}$.

A symmetric (not necessary positive definite) n by n matrix can be converted to graph by the same way as nonsymmetric. However, such conversion does not ensure, that computed row and column ordering will persevere symmetry. Better conversion is to use only rows or columns as vertex set V.

Figure 4.1: Matrix partitioned into 8 partitions.



(a) An original matrix (b) A partitioned matrix

4.4 Matrix partitioning

A way, how to split a graph into equal-sized partitions, was described and the relation between matrix and graph was examined. Now, one can take arbitrary graph partitioner and transform general matrix A to block diagonal matrix A'.

Even through the best possible partitioner is used, there will still be edges connecting two different partitions (unless partitioned graph is discontinuous). This edge cut represents elements of matrix A', which stays out of block diagonal. To be able to execute inverse factorization in parallel, diagonal submatrices A_i should be independent. Figure 4.1 shows partitioned matrix with elements outside the main block diagonal.

One possible approach is to ignore all bad positioned elements. Normally, a dropping is applied during computation of an inverse factor Z. But in this case, the dropping is realized by modifying matrix A' itself before the inverse factorization is executed. However, such modification can be projected into dropping executed in process of computation of an inverse factor Z.

The only question, which needs to be answered, is how this dropping affects rates of convergence of the conjugate gradient method. Numerical properties of that dropping will be examined in Chapter 7.

Chapter 5

Platform for parallel computing

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model created by NVIDIA Corporation and supported by the graphics processing units (GPUs) that they produce.

This design is more effective than general-purpose CPUs for algorithms where processing of large blocks of data is done in parallel. GPU offers more floating operations per second and theoretical higher memory bandwidth. NVIDIA Corporation claims[15], that more chip transistors are used for computation purpose rather than data caching and instruction flow control.

5.1 Kernels

CUDA Platform allows the programmer to define C functions, called kernel. When kernels are called, then they are executed N times in parallel by N different CUDA threads. A kernel is defined using the __global__ declaration specifier.

5.2 Thread Hierarchy

In CUDA platform, threads are arranged in three dimensional grids called blocks, so each threads id consists of three indices (one for each dimension). The benefit of that thread organization is to be able to easily operate on such structure as array, matrix, or volume.

The multiprocessor creates, manages, schedules, and executes threads in groups of 32 parallel threads called warps. Individual threads composing a warp start together at the same program address, but they have their own instruction address counter and register state and are therefore free to branch and execute independently.

A warp executes one common instruction at a time. Full efficiency is achieved only if all threads of a warp agree on their execution path. Threads of a warp may diverge via a data-dependent conditional branch. Divergence of threads force the warp to serialize execution of each branch path with disabled threads that are not on that path. When all paths are completed, the threads converge back to the same execution path.

Every multiprocessor has a limited register pool. To be able to execute one thread block, its number of threads multiply by register per threads required by

kernel must be lower than the multiprocessor register limit. However, a kernel can be executed by multiple equally-shaped thread blocks, so that the total number of threads is equal to the number of threads per block times the number of blocks.

Blocks are organized into a three-dimensional grid of thread blocks. The number of thread blocks in a grid is usually dictated by the size of the data being processed or the number of processors in the system, which it can greatly exceed.

One mustn't rely on thread block execution order. They can be scheduled in any order, in parallel or in series, across any number of cores. This independence allows programmers to write code that scales with the number of cores. Hence, it is not possible to synchronize threads across the blocks other than finishing the execution of all scheduled blocks.

Threads within a block can cooperate by sharing data through some shared memory and by synchronizing their execution to coordinate memory accesses. To be more specific, a thread can be synchronized at specified points in the kernel by calling the __syncthreads() intrinsic function. This function block all proceeding threads in the block until all of then reach this point.

5.3 Memory Hierarchy

CUDA platform contains several types of memory, which are accessible to programmer. The proper usage is important for overall performance.

Global memory The biggest part of the memory. It is shared among multiprocessor. Access to device memory is done by 32-,64- and 128-bytes memory transactions. When threads in warp access the same aligned memory segment, then its request is preformed by only one memory transaction and it is called coalesces memory access. Otherwise, memory access is split into multiple memory transaction.

At the current generation graphics devices, all accesses to the global memory are cached in L1 cache. Shared memory and L1 cache share the same on-chip memory. A programmer could set type of preference L1 cache over shared memory. It is useful in case it is required implicitly accessed.

Shared memory The shared memory is on-chip memory. Each multiprocess has its own private memory. Memory is called shared, because it can be only accessed by threads in the same blocks and its durability confirms to the durability of its block.

To be able to access shared memory simultaneously, memory is divided into the 4-byte banks. If two or more threads want to access the same bank at the same time, their requests have to be serialized. These situation are called bank-conflicts.

The amount of used shared memory is one of the criterion, which define how many blocks can be executed on one multiprocessor simultaneously. The rest of the criteria are total number of threads per blocks and register usage per kernel.

Constant memory The constant memory resides in device memory. Access to the constant memory is cached on all generations of CUDA compatible

graphics devices. If threads within the same warp request a values at the same address from constant memory, then values are retrieved and broadcasted among threads. However, when threads of the same warp request different addresses, then their demands have to be serialized.

Texture and surface memory The texture and surface memory spaces reside in device memory and the accesses to them are cached in the texture cache. The texture cache is optimized for 2D spatial locality, so threads of the same warp that read texture or surface addresses that are close together in 2D will achieve best performance.

5.4 Matrix-Vector product

We have already mentioned, that matrix-vector product is a very important operation in solving system of linear equations. Proper selection of implementation of those products could speed up computation of many numeric algorithms.

In technical report [4] several ways to compute matrix-vector product on GPU are compared. The best results were achieved by using Diagonal matrix representation. However, we do not have such luxury to restrict kernels to diagonal or band matrices, because we have to be able to process any type of matrix.

5.4.1 Coordinate format

Each element of the matrix represented by coordinate format consists of three elements, row index, column index and value. In practice, the matrix is represented by tree arrays *rindices*, *cindices* and *values*, where each array contains one part of the matrix element.

From the point of view of CUDA platform the coordinate matrix can be easily paralleled. Computation can be divided into independent tasks, which can be distributed among threads and then computed independently. Coordinate kernel shown in Figure 5.1 can be very simple.

Figure 5.1: Coordinate kernel $y = Ax$.

```
__global__ void product_coo(const int nnz,
                            const int *rindices,
                            const int *cindices,
                            const float *values,
                            float *x,
                            float *y)
{
    int index = blockDim.x * blockIdx.x + threadIdx.x;

    if (index < nnz)
    {
        int row = rindices[index];
        int col = cindices[index];

        atomicAdd(y + row, values[index] * x[col]);
    }
}
```

The order of elements is unimportant for implementation of the product. However, it is difficult to avoid write conflicts in the result array y and therefore memory atomic operation must be used to preserve the correctness of computation. Theoretically, atomic memory operation force threads to serialize access to the memory and they deny effective parallelism. However, if the write conflicts are sparse, then their usage insignificantly slow down computation.

5.4.2 Compressed rows format

The matrix is represented by three arrays, array *row* containing pointer on begins of each row, arrays *indices* and *values* containing column index and value of every element.

The probably simplest implementation assigns every row to one warp. Each warp compute vector product and first threads of warp write the number to the result array. The number of warps has to be grater than number of rows. The first CSR kernel in Figure 5.2 shows, that implementation of matrix-vector product can be tricky.

Figure 5.2: CSR kernel $y = Ax$.

```
__global__ void product_csr(const int num_rows,
                            const int *rows,
                            const int *indices,
                            const float *values,
                            float *x,
                            float *y)
{
    int index = blockDim.x * blockIdx.x + threadIdx.x;
    int row = index / warpSize;

    if (index < num_rows)
    {
        int begin = rows[row];
        int end   = rows[row + 1];

        float val = 0;

        for (; begin != end; ++begin)
                val += x[indices[begin]] * values[begin];

        y[row] = warp_reduce(val);
    }
}
```

This type of parallelism has one shortcoming. It is the possible low utilization of some threads in case that the numbers of elements per rows are versatile. If the number of elements per rows is not a multiple of threads in the warp, then some threads have to be inactive. It is impossible to avoid those matrices, because compress rows format is frequently used there where such matrices are.

An alternative solution is to divide work by nonzero elements instead of by rows. The function of that distribution is depended on the existence of effectively evaluated method *get_rows*, which returns rows according to the given position of the element. Realization of that method is possible. (For example, binary search could be used in rows array.) The second CSR kernel in Figure 5.3 uses function *get_rows*.

Figure 5.3: CSR Kernel $y = Ax$.

```
__global__ void product_csr(const int num_rows,
                            const int *rows,
                            const int *indices,
                            const float *values,
                            float *x,
                            float *y)
{
    int index = blockDim.x * blockIdx.x + threadIdx.x;

    if (index < nnz)
    {
        int row = get_row(rows, index);
        int col = indices[index];

        atomicAdd(y + row, values[index] * x[col]);
    }
}
```

Function *get_row* has to be evaluated in constant time, otherwise computation may be slowed down. This requirement cannot be satisfied for every kind of matrix, but relatively weak restriction on processed matrices, that they don't contain empty rows (nonsingular matrices), could allow function *get_row* to be evaluated in constant time.

Write accesses by threads to memory is not deterministic. This behaviour may lead to unpredictable results on CUDA platform therefore atomic arithmetic operation should be used.

Chapter 6

The stabilized inverse factorization implementation

This chapter describes parallel implementation of the stabilized inverse factorization (SAINV). The implemented algorithm is restricted to symmetric and positive definite matrices A. Number n marks number of input matrix rows (columns).

The algorithm of the stabilized inverse factorization can be computed in parallel by design. However, special multi-thread devices, especially those which are SIMD (single instruction and multiple data) as graphic processors, require special threat.

There are several implementation questions which have to be answered. The first questions is which element processing should be used.

Left-looking. Natural right-looking variant of the stabilized inverse factorization can be rewritten to left-looking. However, this type of processing has one shortcoming. Very limiting is the dependence of iterations in inner loops, that could prevent effective parallelism.

Right-looking. Right-looking processing is natural for the stabilized inverse factorization and it can be computed in parallel by design. There is no obvious limitations for parallel computing.

The right-looking variant seems more suitable for parallelism than the left-looking variant due to lack of temporal memory and independence of iterations in inner loops. The second question is how many rows process in one iteration.

Per iteration. Straight way to start parallel implementation. In each step only one row is processed. Simplicity of the implementation is a trade-off lowering the possible device opacity.

Multiple iterations. In this type of parallelism multiple rows are processed in one step. This type could achieve higher opacity of device then the previous one. However, in the view of method each iteration is highly depended on the previous one, because current iteration could require vector computed in the last iteration. From a different point of view, multiple iteration processing is a compromise between left and right looking processing. Sadly, it inherits all or some disadvantages from both variants.

The second option is to ignore same inner product to force several following iteration to be independent. This technique would allow the computing of several iterations at once. However, such dropping may significantly affect the numerical properties of computed inverse factors.

Per iteration processing is more promising to achieve better parallel performance. It is due to the lack of additional data structures and unnecessary threads synchronization, that could slow down the process.

6.1 Basic method

The basic approach is to take the given algorithm and try to rewrite it to the designated hardware. Algorithm constructs approximation of inverse matrix Z so choosing the right representation of that matrix is very important. Selected representation should fulfil several requirements. It should offer efficient computation of matrix-vector products and it also should allow to access and modify each row individually. Therefore it seems reasonable to store current matrix Z_i as an array of independent sparse vectors.

Every iteration of SAINV could be divided into three steps. After every step, threads synchronization should follow.

$t_i \leftarrow z_i^{(i)} A$

The first step is the product of matrix A and vector z_i^i. In general, those products are expensive because they require read whole matrix A. However, when very aggressive dropping strategy is used, then vector z_i^i contains only a few nonzero elements. Dropping is expected to be aggressive, otherwise more efficient and less stable methods should be used. Therefore the product could be computed as linear combination of a few columns of sparse matrix A.

Linear combination could be computed in parallel by letting each thread append one row to the result vector. If the number of threads is greater then the number of rows, then each row should process a group of threads. This computation does not avoid read or write hazards, so every arithmetic operation processing result vector should be atomic.

for $j = i \rightarrow n$ **do**
  $p_j^{(i)} \leftarrow t_i^t z_j^{(i)}$
end for

The second step is probably the most important for overall performance. It consists of several dot products. But, it could be seen as matrix and vector product. Implementation of that product is described in Chapter 2.

The resulting values $p_j^{(i)}$ form vector $\overrightarrow{p_i}$. Usually, those vectors are sparse, and for further usage it is more useful to use sparse vector representation. Fortunately, Dense vector could by converted to sparse vector effectively by using parallel move.

for $j = i + 1 \rightarrow n$ **do**
  $z_j^{(i+1)} \leftarrow z_j^{(i)} - \frac{p_j^{(i)}}{p_i^{(i)}} z_i^{(i)}$
end for

The last step involves several sums of vectors and also dropping. However, if very aggressive dropping is used (as mentioned above, only aggressive droppings

are meaningful), then those vectors are sparse. Moreover, lot of sums mustn't be computed, because their multiplication elements p_j^i are equal to zero. Therefore, effectiveness of the last step is not so critical for overall performance and so less efficient sums implementation could be used. Note, that the sum of two sparse vectors could be viewed as a merging sorted array.

The following definitions should save some space and increase the document's readability. The first definition is content of matrix Z in ith iterations:

$$Z_i = (z_1^i, \ldots, z_n^i)$$

And the second is the shortcut for current process row (ith rows):

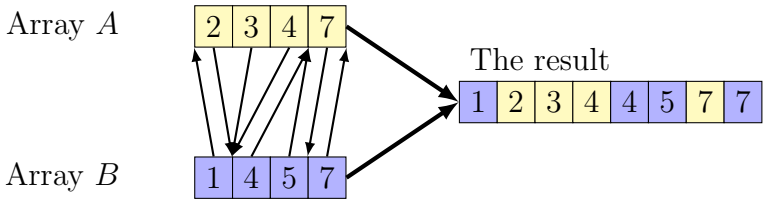
$$\overrightarrow{z_i} = z_i^{(i)}$$

6.1.1 Parallel merging

There is a simple way to merge sorted arrays A and B in one thread. (In the case of unsorted arrays, merging is even simpler, it can be done by concatenation of input arrays.) It consists of one loop and two pointers on the start of both of input arrays. In each iteration one pointer which point on smaller value advance and no longer pointed value is written to the result array. The loop ends, when the ends of both arrays are reached. Clearly, this process run in linear time to sizes of the input arrays. Moreover duplicities, elements with equal values, can be removed at no cost.

However, described merge can with difficulty be done in parallel because of iterations dependence. But, there is a way to deal with that. The idea of parallel merge is to determine the position of each element in result array independently. The final position of element in array A consists of its current position and hypothetical position in array B. The hypothetical position is the position of the first element in array B, which is not less in value. The same principle holds for final positions of elements of array B except that hypothetical positions are positions of the first elements from array A, in which the values are greater. If some elements in array A and B are equal, then equal elements from array A always precede before elements from array B. Figure 6.1 shows how to find new positions when two arrays are merged.

Figure 6.1: Parallel merging of two sorted arrays by finding final positions.



As opposed to regular merge, parallel merging doesn't run in linear time. Selecting positions is not a constant operation, but is upper bounded by logarithmic time in case binary search is used.

Parallel merging leaves duplicities in the result array. When they are not wanted, then additional processes should be executed to remove them.

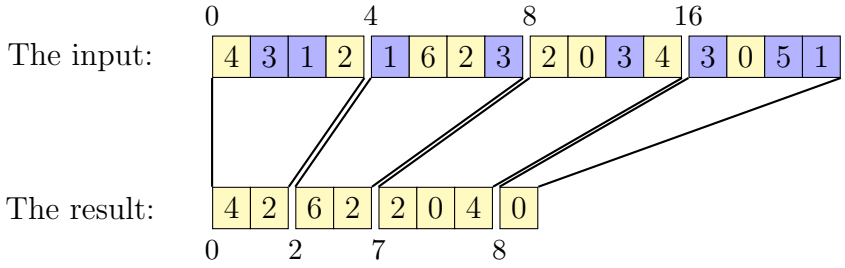
6.1.2 Parallel conditioned moving

Operation of moving elements specified by given condition from one array to another is very important. Despite single thread moving is simple, parallel moving is quite complicated. Parallel moving has to read input array at least twice, but it still runs in linear complexity. It also requires additional threads synchronization. Figure 6.2 demonstratives idea of parallel moving.

Process of parallel moving is divided in three steps. Between following steps is always required threads synchronization.

- In the first step input array is divided into equal sized blocks. Each thread processes its own block and counts elements, which satisfy the given condition.
- The next step is to compute new elements positions in result array. Each thread is required to know how many elements in the previous blocks will be moved. This operation is called prefix sum, and it is referred as sums of previous element counts. Prefix sum could be computed easily in parallel. However, in practice the number of threads is always less than the number of processed elements, so prefix sum can be computed by thread itself.
- The last step is to write elements on their new positions. Each thread process the same block as in the first step and elements, which satisfied the given condition, are moved to new positions. Each thread reads its own blocks and also writes to its own segment. There are no read or write collisions.

Figure 6.2: Parallel conditioned moving.



Conditional moving can be viewed as conditional removing when input array is dropped.

6.2 Coordinate method

The next research leads to the development of the method based on coordinate sparse matrix representation. Each element of coordinate representation consists of row index, column index and value. The main advantage of this representation is that the order of matrix elements is not important. The third step of SAINV could be viewed as matrix sum:

$$Z_i = Z_{i-1} - \overrightarrow{p_i}^t \overrightarrow{z_i}$$

The sum of two coordinate matrices can be implemented as concatenation of indices and values array. However, this implementation could cause, that the resulting matrix may contain multiple elements with same row and column indices. But, this issue can't damage the correctness of the result from the view of implementation of sparse matrix product.

The disadvantage of that method is the realization of dropping. Dropping has to deal values distributed among several elements and because there is no obvious way to aggregate these values, so dropping has to process possible incomplete values. Due to this dropping realization, The method diverges from the original SAINV algorithm.

If the dropping is unchanging during computation, than only the second operand is affected by dropping. The first operand is unchanged, so to let it process by dropping makes no sense.

6.2.1 Practice realization

The implementation described above is only correct if no dropping is used. Otherwise it could diverge from the original SAINV. Divergence is caused be values distributed among multiple elements. The following text contains a way of how to join them together.

The first implementation sums coordinate matrices simply by concatenating their values and indices array. If the orders of elements are unknown, then the concatenation is only one efficient method. However,the structure of the second operands are known. The second operands are formed by products of vectors $\overrightarrow{p_i}$ and $\overrightarrow{z_i}$. Therefore the second operands are dense matrices. (Generally, the second operands are not dense matrices but they have nice structures.)

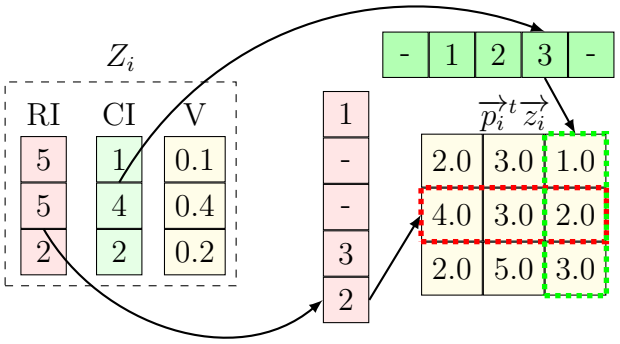
The third step of SAINV can be viewed as sums of sparse matrix and dense matrix. Those sums can be implemented very easily. All element of the first operand (sparse matrix) are read and their vales are added to values of the second operand with same row and column indices. The resulting dense matrix is converted to a sparse matrix by parallel move. Dropping is used as the moving condition.

In practice, it is not possible to explicitly construct and process those dense matrices. If vectors $\overrightarrow{p_i}$ and $\overrightarrow{z_i}$ are sparse (in practice they are), then those dense matrices haven't full ranks and also contain a lot of pure zero rows and columns. Every zero element of $\overrightarrow{p_i}$ yields zero row and every zero element of $\overrightarrow{z_i}$ yields zero column.

All zero rows and columns can't be stored, but nonzero elements must be recognized during the preprocessing. The trick is to involve two inverse indexing arrays of n size. These arrays map rows and columns indices to theirs positions in sparse vectors $\overrightarrow{p_i}$ and $\overrightarrow{z_i}$ (see Figure 6.3). If the sparse vector doesn't contain certain elements, then the element's position in inverse indexing array is marked as invalid.

When a sparse matrix is processed, it is required to append the rest of the elements of dense matrix. Append could be done by already described parallel moving. Then two matrices are added and the resulting matrix doesn't contain any duplicities. Dropping, executed during parallel moving, process complete values, therefore coordinate method converges with original SAINV.

Figure 6.3: Selecting values by inverse indexing.



In ith iteration the original algorithm only process elements in which row index is at least $i+1$. However, with used matrix representation it is not possible to select these elements other than by reading the whole matrix. Reading the whole matrix in each iteration may significantly slow down computation. One possible solution is to remove rows, which will no longer be used.

Elements no longer used can be removed by previously described parallel move. This operation is relatively expensive. (Read whole matrix in each iteration is expensive in comparison with what is done in one iteration.) However, it is not required to remove them in each iteration. It is better to do removal in each fourth or tenth iteration. Moreover unused elements mustn't be set to zeros.

6.2.2 Pros & Cons

+ Method is more friendly with hardware with lots of independent threads. It also gets on with implicit threads grouping presented by some hardware.

+ Theoretically, computed inverse factors are equal to inverse factors computed by original algorithm (even with dropping used). In practice, the computed result is different, because the order of arithmetic operation is not the same and standard computed arithmetic isn't associative (due to rounding errors). This property is useful when the method is compared with the original algorithm.

- Coordinate matrix representation may require more memory space than other representations because it uses less efficient coordinate representation. Also additional memory is needed to store one temporal matrix per iteration. However, only one bit per element of temporal matrix is satisfactory in the case of smart implementation and that matrix could be recycled during iterations.

- Selected rows of the matrix Z_i have to be processed at least twice every iteration. Processing matrix Z_i has a great impact on overall performance in contrast to what is done in one iteration.

6.3 Implementation notes

The previously described implementations required several threads synchronization per iteration. Generally it is not possible to avoid them, but their numbers can be decreased. The trick is to let threads process several rows alone. (For example, every ith thread will process only each ith row).

If threads process their own rows exclusively, the final number of threads synchronization is reduced to one. The only required synchronization is after the first step of original SAINV.

The disadvantage of this concept is the possibility of lower opacity of some threads. In practice, it is better to let a group of threads process multiple rows. For example, graphics processors offer implicit threads grouping by its multiprocessor and synchronization of threads in one multiprocessor is always faster then whole device synchronization.

Chapter 7

Numerical experiments

This Chapter contains numerical experiments. The results presented here may be considered as competition between a general processor unit (CPU) and a graphic process unit (GPU). All work can be divided into three areas.

The first series of experiments compares several implementations of the conjugate gradient method (CG) and the preconditioned gradient method (PCG). The goal was to discover how CG and PCG behave on GPU and which matrix representation is the best. The possible matrix representation and their properties were discussed in Chapter 2 and Chapter 5.

The second area contains several series of tests, in which the purpose is to examine numerical properties of the inverse decomposition of block diagonal matrix (BAINV), more precisely described in the Chapter 3. The algorithm decomposes given matrix into blocks, which are factorized separately by the regular inverse factorization (AINV). There are also comparisons with different methods.

The last area is about the examination of properties of computation of the stabilized inverse factorization (SAINV), which was described in Chapter 3. The GPU variant is not straightforward implementation of standard SAINV. It required some workaround to be efficiently working on GPU.

7.1 Testing methodology

7.1.1 Setting of the conjugate gradient method

It is quite difficult to determine the quality of a preconditioer, when the convergence rates of PCG are dependent on the initial guess x_0 and the right side b. Apparently, this issue is not so important, when two variants of the same algorithm are compared. However, the comparison of different algorithms may be unreliable and produce various results with setting of CG.

Matrix A. The matrices were mainly taken from Tim Davis sparse matrix collection [9]. Referenced Table 7.1 shows brief description of chosen test matrices.

Table 7.1: Tested matrices.

Matrix	n	nnz	Kind
1138_bus	1138	4054	Power network problem
af_5_k101	24910	17550675	Structural problem
apache1	80800	542184	Structural problem
bcsstk15	3948	117816	Structural problem
bcsstk36	23052	1143140	Structural problem
bundle1	10581	770811	Computer graphics
cbuckle	13681	676515	Structural problem
cfd1	70656	1825580	Computational fluid dynamics problem
cvxbqp1	50000	349968	Optimization problem
denormal	89400	1156224	Counter-example problem
finan512	74752	596992	Economic problem
G2_circuit	150102	726674	Circuit simulation problem
gridgena	48962	512084	Optimization problem
gyro_k	17361	1021159	Model reduction problem
Kuu	7102	340200	Structural problem
msc10848	10848	1229776	Structural problem
msc23052	23052	1142686	Structural problem
offshore	259789	4242673	Electromagnetics problem
olafu	16146	1015156	Structural problem
pdb1HYS	36417	4344765	Weighted undirected graph
Pres_Poisson	14822	715804	Computational fluid dynamics problem
pwtk	217918	11524432	Structural problem
raefsky4	19779	1316789	Structural problem
ship_003	121728	3777036	Structural problem
shipsec5	179860	4598604	Structural problem

Right side b. The best possible option for the right side is to choose sum of rows of matrix A. Therefore the element of right side are equal to

$$b_i = Ae_i, \quad (1 \leq i \leq n).$$

Naturally, the final solution is equal to a vector filled with ones.

Initial guess x_0. The choice of the initial guess is related to the choice of right side. Since row sums were used as right side, the initial guess was chosen as zero vector.

Scaling. The main reason for using scaling is to prevent possible numerical mistakes caused by the significant difference between values. Scaling was taken from paper [13].

$$\hat{A} = D^{-\frac{1}{2}} A D^{-\frac{1}{2}}, \quad \text{where} \quad D = diag(\|Ae_i\|_2)$$

The tested methods requires matrices to be symmetric and positive definite. Used scaling multiplies matrix A by nonsingular diagonal matrix from both sides, so the result is symmetric and also positive definite.

Stopping criteria. The method is considered divergent if it doesn't achieve estimative criteria until 1000 iteration passed. Used criterion was, that ratio between starting residuum r_0 and r_i is under certain value.

$$\frac{\|r_0\|_2}{\|r_i\|_2} \leq 10^{-5}$$

7.1.2 The testing hardware

The majority of tests were about comparison between implementation on CPU and GPU. Since those two devices are based on various architectures and their designs are focused on a slightly different purpose, it is quite difficult to decide how to compare them. Naturally, they were compared by computation times, but the question can be those two devices be considered to be equal may be difficult to answer.

A business principle would probably be to compare them by their sale price. From that point of view, they are almost equal, because their sale prices were similar. However, their prices do not not stay forever, so there is only hope, that ratios between cost and performance will be the same in the future.

CPU. The testing CPU was a quad-core Intel Core i5. The attractive feature of this CPU is the so called Turbo boost ability. It enables the processor to run above its base operating frequency via dynamic control of the CPU's "clock rate", when the operation system detects that the task is one core demanding.

The installed memory is DDR3 with nominal frequency 1600 MHz. Theoretical memory bandwidth is 25.6 GB/s. According to memory benchmark, practical memory bandwidth is up to 16 GB/s.

GPU. All the experiments were performed on NVIDIA GeForce 660 GT. That GPU contains 960 independence cores. It has own dedicated memory of size 2GB with theoretical memory bandwidth 140 GB/s. Practical benchmark shows, that practical memory bandwidth can be up to 110 GB/s.

Double precision performance of the available GPU is very bad, since its purpose is accelerating computer graphics, where single precision is mainly used. Therefore, all experiments had to be done under single precision. NVIDIA Corporation offers a special type graphic accelerator focused on scientific computation, which may deliver better double precision performance.

Since the majority of sparse matrix operation used by tested algorithms highly rely on practical memory bandwidth, GPU may have significant advantage over CPU.

7.2 The conjugate gradient method

The first series of tests compare CPU and GPU implementation of the conjugate gradient method and the preconditioned conjugate gradient method. It was the first attempt to utilize GPU, because implementation of CG is very simple (PCG is a little bit complicated, but it can be also considered as simple) by the means of GPU.

Since matrices A remain unchanged during computation, the best possible matrix representation may be used. In Chapter 5 several matrix representations were described. The GPU implementation was done for CSR a COO matrix representation, on the other side CPU implementation used only CSR one, because it is more general and memory efficient.

A very tricky question is which variant $x^T A$ or Ax of matrix-vector product to use. Since matrices A are required to be symmetric, both variants are valid and both can be used.

Although they are mathematically equivalent and produce the same result, the performance could be significantly different. This may be caused by different access patterns to the memory. I found by experiment, that left variant $x^T A$ is more suitable for GPU, on the other side CPU perform better with right variant Ax. It is probably due to realisation of the memory caches of CPU and GPU.

Table 7.2 and Table 7.3 contains the results of numerical experiments of CG a PCG. From the experiments it can be seen, that both GPU implementations of CG (and PCG) beat CPU in all cases. The only exception is a small matrices with low rank and a few nonzero elements. In such cases GPU can be hardly fully utilized.

Same results apply for the GPU implementations of PCG. GPU outperforms CPU in every taken test. This is very important, because it may seem reasonable to compute a preconditioner on GPU, since it is not required to move result to CPU memory.

Table 7.2: The comparison of implementations of CG on CPU and GPU.

Matrix	CPU		GPU			
			CSR		COO	
	Time	It	Time	It	Time	It
1138_bus	20	620	70	620	70	620
af_5_k101	24910	1000	3980	1000	3600	1000
bcsstk15	170	1000	130	1000	140	1000
bcsstk17	550	1000	200	1000	200	1000
bcsstk36	1410	1000	300	1000	310	1000
cbuckle	570	686	160	686	150	685
cfd1	2930	1000	540	1000	490	1000
cvxbqp1	1060	1000	380	1000	230	1000
denormal	2370	1000	600	1000	350	1000
finan512	20	11	10	11	10	11
G2_circuit	1720	581	510	582	220	582
gridgena	1280	1000	360	1000	230	1000
gyro_k	1190	1000	310	1000	310	1000
Kuu	140	325	50	325	60	325
msc10848	1260	1000	350	1000	330	1000
msc23052	1440	1000	320	1000	320	1000
offshore	6300	680	1960	683	2490	682
olafu	1160	1000	280	1000	290	1000
pdb1HYS	4490	1000	760	1000	840	1000
Pres_Poisson	540	605	140	604	140	604
pwtk	14310	1000	2110	1000	2230	1000
raefsky4	1520	1000	330	1000	330	1000
ship_003	9440	1000	1660	1000	1820	1000
shipsec5	12190	1000	2100	1000	2130	1000

Table 7.3: The comparison of implementations of PCG on CPU and GPU.

Matrix	nz(Z)	CPU		GPU			
				CSR		COO	
		Time	It	Time	It	Time	It
1138_bus	4865	10	67	10	65	20	65
af_5_k101	2578989	35660	1000	8480	1000	4810	1000
bcsstk15	25822	60	169	30	167	40	167
bcsstk17	96654	950	1000	250	762	220	763
bcsstk36	328307	2480	1000	580	1000	520	1000
cbuckle	131492	260	192	60	175	60	175
cfd1	793660	5870	1000	1100	775	750	774
cvxbqp1	435078	2770	1000	1320	1000	860	1000
denormal	1617307	7070	1000	1630	1000	1130	1000
finan512	187850	30	8	20	8	10	8
G2_circuit	737027	1340	216	500	211	180	211
gridgena	260548	2330	1000	800	988	390	988
gyro_k	113924	1780	1000	500	1000	420	1000
Kuu	62325	60	96	20	94	30	94
msc10848	105340	1680	1000	470	1000	450	1000
msc23052	527497	2890	1000	1000	1000	1130	1000
offshore	1198822	5570	326	1680	317	1490	316
olafu	155731	1790	1000	460	1000	420	1000
pdb1HYS	65947	5310	1000	1090	1000	940	1000
Pres_Poisson	87131	440	328	130	326	120	326
pwtk	2711031	23960	1000	4320	1000	3470	1000
raefsky4	105432	2130	1000	540	1000	440	1000
ship_003	674321	13400	1000	2800	1000	2320	1000
shipsec5	1764774	19320	1000	4120	1000	3370	1000

GPU versions of PCG and CG were implemented for a certain reason, which is to get a clue about how GPU behave. This knowledge was useful in further implementation of an inverse factorizations, because PCG and an inverse factorization have a lot of in common.

7.3 The inverse factorization of block diagonal matrices

The second series of test was about the comparison of factorization of a block diagonal matrices (BAINV). It is clear, that the inverse factorization of a block diagonal matrix can be computed in parallel very easily.

In one of the previous chapters we mentioned, that any general matrix could not be transformed to block diagonal matrix by row and column reordering. One possible (and tested) option is to ignore elements located out of the main block diagonal. The number of elements outside the main block diagonal should be as minimal as possible.

Higher number of partitions mean lower size of factor Z and also shorter factorization time. On the other side, it also means longer partitioning time and slow rates of convergences. To find the right balance of these two criteria is complicated and it can be only done by experiment.

The following table7.4 shows dependence of times of decomposition on the numbers of blocks. Time of factorization consists of partitioning time and factorization itself. These two times are in contradiction, i.e., when time of decomposition decreases, time of partitioning increases and so on.

The external library METIS [11] was used to partition matrices. Therefore naturally partitioning time is not a concern of this Thesis. There is a chance, that different setting of parameters or other libraries, which could do the same, may deliver better partitioning times. Another option is to use the parallel version of METIS crated by the same authors. This option wasn't tested.

Table 7.4: Dependence between times of decomposition and numbers of parts.

Matrix	Parts	Time	nz(Z)	It	CG Time
bundle1	128	410 + 20	142476	31	40
bundle1	256	460 + 10	148100	30	40
bundle1	512	680 + 10	69696	32	40
cbuckle	128	120 + 50	736429	108	270
cbuckle	256	290 + 30	371668	144	260
cbuckle	512	680 + 20	188343	218	310
G2_circuit	256	310 + 2450	40735297	234	20170
G2_circuit	512	560 + 780	19860311	196	8920
gyro_k	256	380 + 40	585315	1000	2600
gyro_k	512	1000 + 30	262928	1000	2020
offshore	512	1420 + 7510	64561935	436	65320
offshore	1024	2410 + 2490	32491762	371	30020
offshore	2048	4250 + 950	16354213	428	20200
offshore	4096	7980 + 450	8245145	460	14080

The size of inverse factor is bounded from above by number of partitions and its sizes. More partitions yields lower size of inverse factor but also worse convergence rates. One should find the biggest number of partitions, where inverse factorization is still economical.

Increasing number of parts make sense until certain value is reached (this values is unknown), where inverse decomposition is still economical. Above this certain value, partitioning is to much expensive, or CG became divergent. There weren't done any research to obtains this value. One should use trial and fail.

7.3.1 Comparison with other methods

A very important part of the research is to compare numerical properties with competitive methods. For the record it was recalled, that ignoring elements outside main block diagonal is actually a special kind of dropping implicitly used during computation of Z factor.

In contrast the regular inverse factorization (or its stabilized variant) is that method, which uses a different dropping strategy, i.e., ignoring elements with values that fall under certain tolerance. And here comes the question. How to decide which preconditioner is better when sizes of inverse factors Z are different? The answer is that it is impossible to decide. The one used solution is to set parameters such, that sizes of inverse factor are approximately the same. I found the right sizes by trial and error.

Firstly BAINV were compared with the inverse factorization (AINV) with dropping strategy, which ignored elements of an inverse factors Z under tolerance equal to 10^2.

Table 7.5: Comparison between BAINV and AINV(10^{-2}).

Matrix	partitioning + ainv				sainv		
	k	Time	nz(Z)	It	Time	nz(Z)	It
bcsstk15	29	30 + 20	261510	210	90	319606	71
bundle1	450	720 + 10	3880	33	2610	93981	22
G2_circuit	1400	1420 + 230	7271048	231	660	6136601	80
gyro_k	30	60 + 850	4619008	1000	2250	4592265	1000
Kuu	25	30 + 80	983189	73	300	1016194	152

The Table 7.4 didn't reveal any surprising results. Times of computation are approximately the same but numerical stability of BAINV is always better. It has to be mentioned, that computation of AINV fail in some cases. The method BAINV is at least as good as regular AINV.

The comparison between BAINV and SAINV, which ignored values fallen bellow 10^2, is displayed in Table 7.6. It is much more interesting. The stabilized variant of AINV is more numerical stable but more expensive to compute. The difference is in the computation of inner product, where SAINV uses selection of several rows (columns) instead of only one row.

Table 7.6: Comparison between BAINV and SAINV(10^{-2}).

Matrix	partitioning + ainv				sainv		
	k	Time	nz(Z)	It	Time	nz(Z)	It
bcsstk15	29	30 + 20	261510	210	1490	264434	58
bundle1	450	720 + 10	76086	33	5750	75384	21
cbuckle	65	70 + 100	1444391	101	8600	1426542	65
G2_circuit	1400	1420 + 230	7271048	231	8150	7553447	76
gyro_k	100	120 + 170	1465118	1000	11130	1420169	804
Kuu	40	40 + 40	623966	91	2070	664237	47
offshore	4096	7980 + 450	8245145	460	83480	9750279	114

Generally, the inverse factorization of blocks is always faster than SAINV, when sizes of computed factors are equal. However, one should count a partition time, which increases the overall time of BAINV. As was mentioned above, the partitioning time can possibly be reduced by a different library.

One look at convergence rates reveal surprising information. In some cases of convergence rates are much better. This may be seen as competition between

value and position dropping strategies, where position strategy doesn't behave badly.

It is important to mention one observation. At a certain point, the time of computation of BAINV stops decreasing with growing number of partitions. From that point of view SAINV may beat BAINV in time of computation, when dropping strategy is too strict. The open question is how those strict strategies are meaningful, because a very sparse inverse factor may have bad numerical properties.

BAINV can be considered as inverse factorization where its dropping is a trade off between speed and numerical stability. Generally, its computed factors are more numerically stable than those computed by AINV. On the other side, time of computation are generally better than SAINV could do. But the biggest advantage is the possibility to be computed in parallel without any issues.

7.3.2 Computation on GPU

The inverse factorization of block diagonal matrix can be computed on GPU very easily. Since each computation entity (thread, warp, or multiprocessor) may have an independent bunch of data, no synchronization point is required. The only one thing, which may deny full utilization of GPU, is unpredictable memory access pattern.

GPU and CPU implementations of the inverse factorization shared the same code, so it wasn't a surprise that sizes of inverse factors were the same. The equality of convergence rates of PCG was a consequence of similarities of inverse factors.

Table 7.7: The CPU and GPU implementation of the block inverse factorization.

Matrix	Parts	CPU			GPU		
		Time	nz(Z)	It	Time	nz(Z)	It
2cubes_sphere	512	660	9589716	10	360	9589716	10
2cubes_sphere	1024	270	4823678	10	350	4823678	10
finan512	128	1120	11276501	6	420	11276501	6
finan512	256	520	6548061	7	230	6548061	7
finan512	512	200	3498658	7	190	3498658	7
G2_circuit	256	2440	40735297	234	940	40735297	234
G2_circuit	512	790	19860311	196	450	19860311	196
G2_circuit	1024	320	9902278	222	320	9902278	222
offshore	1024	2500	32491762	371	1220	32491762	381
pwtk	1024	1660	22594722	1000	710	22594723	1000
ship_003	512	1190	12363699	1000	460	12363672	1000
ship_003	1024	540	6310271	1000	420	6310254	1000
shipsec5	512	2520	27762265	1000	890	27762146	1000
shipsec5	1024	1070	14194194	1000	540	14194210	1000

Table 7.7 shows the results of experiments. In this series of experiments GPU performed quite well. It is mainly due to computation without any synchronization point, which would break down parallelism. Also implementation is very

simple almost without dangerous write conflicts, which usually arise from parallel programming.

However, when the time of computation on CPU falls bellow one second, it doesn't make sense to process that task on GPU. To fully utilize GPU it is required to process at least a significant amount of data. In other cases, that GPU implementation outperformed CPU implementation in each test.

7.4 The stabilized inverse factorization

In this section I compared two different implementations of same method, the first one runs on CPU, the second one runs on GPU. The Tables 7.9 and 7.8 show computed results of SAINV with dropping strategy, which ignores elements under given value. I recorded times of factorization, sizes of factors, and rates of convergences of PCG of both implementations.

It was desired for matrices A and Z to be same sized. However, with used dropping strategy, which ignores elements with value under certain tolerance, it was quite difficult to predict the sizes of resulting inverse factors. So two different tolerances were tested.

Table 7.8: Comparison of CPU and GPU implementation of SAINV(10^{-2}).

Matrix	CPU			GPU		
	Time	nz(Z)	It	Time	nz(Z)	It
apache1	5440	2810010	1000	3120	2810139	1000
bcsstk15	1490	264434	58	200	267679	53
bcsstk36	66780	4275545	1000	3910	4259273	1000
bundle1	5750	75384	21	1140	75379	21
cbuckle	8600	1426542	65	930	1426295	69
cfd1	87570	10749225	1000	10230	10741461	1000
cvxbqp1	91480	4828354	1000	28430	5961674	1000
denormal	92630	7561636	1000	4500	7432288	1000
finan512	2280	602815	4	4820	603327	4
G2_circuit	8150	7553447	76	11770	7573564	76
gridgena	6340	3736771	1000	2220	3749532	1000
gyro_k	11130	1420169	804	1220	1420951	721
Kuu	2070	664237	47	370	664170	47
msc10848	11830	1068235	1000	1800	1070727	1000
msc23052	24150	2388104	1000	5370	2366921	1000
offshore	83480	9750279	114	66140	9748978	114
olafu	8120	1413710	1000	940	1413795	1000
pdb1HYS	14840	1647407	1000	2510	1647398	1000
Pres_Poisson	1700	491506	133	570	491497	132
raefsky4	10350	1208187	1000	1460	1212191	1000
ship_003	64270	5341267	1000	10850	5343773	1000
shipsec5	103280	12210990	1000	22230	12248448	1000

Table 7.9: Comparison of CPU and GPU implementation of SAINV(10^{-1}).

Matrix	CPU			GPU		
	Time	nz(Z)	It	Time	nz(Z)	It
apache1	370	367347	1000	2240	367339	1000
bcsstk15	90	25646	173	120	25819	170
bcsstk36	1400	331374	1000	800	332477	1000
bundle1	100	16648	30	270	16648	30
cbuckle	450	131607	186	420	130836	167
cfd1	1920	793662	1000	2290	793744	1000
cvxbqp1	890	427316	1000	3890	434813	1000
denormal	1550	1637752	1000	2730	1617236	1000
finan512	380	187803	8	2660	187849	8
G2_circuit	550	735929	216	4220	737009	216
gridgena	260	260556	1000	1310	260548	1000
gyro_k	640	113925	1000	580	113924	1000
Kuu	140	62345	96	220	62325	96
msc10848	850	105370	1000	550	105346	1000
msc23052	5740	562671	1000	1930	531636	1000
offshore	3160	1198835	326	12200	1198822	325
olafu	620	156056	1000	520	155039	1000
pdb1HYS	500	65947	1000	1140	65947	1000
Pres_Poisson	270	87142	328	440	87129	328
raefsky4	600	105427	1000	620	105406	1000
ship_003	3100	674296	1000	4000	674318	1000
shipsec5	6280	1764109	1000	6880	1764776	1000

The implementation of inverse factorization was focused on not changing the numerical properties of the process. The experiments showed, that this effort was successful, and the sizes of computed inverse factor by GPU and CPU are equal. This leads to equal rates of convergence of the method of preconditioned gradients.

The biggest slowdown of GPU implementation is when very a sparse matrix is factorized with a strict dropping strategy. To fully utilize GPU it has to process at least hundreds of elements in one iteration, but better is to process thousands and more elements. When this condition is not satisfied, the computation can be considered as serial. In that case, GPU could not achieve any advantage over CPU. But, still it can be useful to employ GPU on inverse factorization, even though with CPU it can done faster.

On the other side, there are plenty of matrices, which can be successfully factorized on GPU with performance benefits. It highly depends on size of result factor Z. More nonzero element means more utilization of GPU.

Chapter 8

Conclusion

The goal of this Thesis was to demonstrate that one particular method for solving system of linear equation can be successfully computed on highly parallel device as graphic processors are. It was proved, that both phases (preconditioning and solving itself) of solving system of linear equation can be successfully computed on GPU.

It was showed, that GPU can significantly reduce convergence times of the conjugate gradient method. In each iteration of that method, matrix-vector product is computed once, which yields enough elements to process, if the matrix is satisfyingly large. The second reason of performance benefits is, that the matrix is unchanged in the whole process. GPU can speed up the convergence up to 6 times, sometimes even more.

Also, GPU computation can fully benefit from the usage of explicit preconditioner, when the preconditioned gradient method is computed. Applying explicit preconditioner involves forming several matrix products instead of performing some triangular solves, which can with difficulty be done in parallel. The preconditioned conjugate gradient method also uses same matrices in all iterations and therefore the same result as for CG applies also for PCG. Smarter matrix representation, which would focus on GPU cache, could achieve even better speed up. But those representations are dependent on the structure of the processed matrix, so they are not included in this Thesis. I found achieved speed up satisfactory.

Author of paper [12] tried to compute PCG with implicit preconditioner based on the incomplete Cholesky decomposition. From my point of view, they achieved very good results for that method. It is good to mention, that their results are unstable and quite unpredictable. In contrast to implicit preconditioner, explicit preconditioner brings reliable and stable speed up independent of structure and size of matrix (with the exception of very small matrices, which are not interesting). The issue of parallelism is the reason why one should use explicit preconditioner

The inverse factorization of block diagonal matrices was tested. It required the decomposing of matrix into blocks, which shows as the biggest problem. The inverse factorization of block diagonal matrices is not suitable for GPU as I expected. In my opinion, it is caused by the higher importance of decomposition to block rather then the inverse factorization itself. So the most time of the computation is taken by CPU.

I don't consider that method very suitable for GPU computing. But I believe,

that there is a chance, that a combination of the position based and value based dropping strategy would allow a decrease in the number of partition with the same or better numerical properties and benefits of parallelism. Also, it would put higher pressure on GPU by computing factorization itself rather then decomposition to block on CPU.

The stabilized inverse factorization may be successfully computed on GPU. However, speed up is not reliable and it is dependent on used dropping strategy. The worst cases are sparse matrices with few element in rows combined with very aggressive dropping. In such cases, one iteration involves processing tens or up to hundreds of elements on which it is not possible to fully utilize GPU.

But in the case that those two conditions don't meet, GPU still has brought speed up over CPU. Although speed up is not reliable, still one should let GPU compute inverse approximate, because it is not necessary to transfer the result back to GPU memory for computation of the conjugate gradient method, which always benefits GPU.

Authors of unpublished paper [8] examine GPU computing of the regular inverse approximation method. I tried only computing of the stabilized inverse approximation, because I considered converge times of the conjugate gradient methods as a bigger problem than the time of factorization itself.

In future work, the means of how to compute several iterations of the stabilized inverse factorization at once should be examined. This could be done by either smart matrix row reordering or by ignoring some products, hopefully that will not affect numerical stability and fill-in. The iteration dependence was the biggest limit of full utilization of GPU.

One particular problem is a used precision of float arithmetic. Double precision computation wasn't tested due to the lack of an appropriate graphics accelerator. This problem was decreased by usage of scaling, which prevents overflowing but still higher precision would bring better convergence rates of the methods of conjugate gradient.

For the purpose of testing a new CUDA framework was created. In some ways usage of that framework is inconvenient, because it is required to know how GPU computing works. It would be nice to allow people, who do not know how GPU works, to express their sparse matrix algorithm with all benefits of GPU.

Bibliography

[1] Jochen Alberty, Carsten Carstensen, and Stefan A. Funken. Remarks around 50 lines of matlab: short finite element implementation. *Numerical Algorithms*, 20:117–137, 1999.

[2] Owe Alexsson. *Iterative Solution Methods*. Cambridge University Press, 1994.

[3] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.

[4] Nathan Bell and Michael Garland. Effcient sparse matrix-vector multiplication on cuda. Technical report, NVIDIA Corporation, 2008.

[5] M. BENZI, J. K. CULLUM, and M. TŮMA. Robust approximate inverse preconditioning for the conjugate gradient method. *SIAM J. SCI. COMPUT.*, 22(4):1318–1332, 1998.

[6] M. BENZI, CARL D. MEYER, and M. TŮMA. A sparse approximate inverse preconditioner for the conjugate gradient method. *SIAM J. Sci. COMPUT.*, 17(5):1135–1149, 1996.

[7] M. Benzi and M. Tůma. A sparse approximate inverse preconditioner for nonsymmetric linear systems. *SIAM J. SCI. COMPUT.*, 19(3):968–994, 1998.

[8] DANIELE BERTACCINI and SALVATORE FILIPPONE. Sparse approximate inverse preconditioners on high performance gpu platforms. University of Rome "Tor Vergata".

[9] T. A. Davis and Y. Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38(1):1:1 – 1:25, 2011. http://www.cise.ufl.edu/research/sparse/matrices.

[10] Dongarra, Duff, Sorensen, and van der Vorst. *Numerical Linear Algebra for High-performance Computers*. The Society for Industrial and Applied Mathematic, 1998.

[11] George Karypis and Vipin Kumar. A fast and highly quality multilevel scheme for partitioning irregular graphs. *SIAM J. SCI. COMPUT.*, 20(1):359–392, 1999.

[12] R. Li and Y. Saad. GPU-accelerated preconditioned iterative linear solvers. *The Journal of Supercomputing*, 63:443–466, 2013.

[13] CHIH-JEN LIN and JORGE J. MORÉ. Incomplete cholesky factorizations with limited memory. *SIAM J. SCI. COMPUT.*, 21(1):24–45, 1999.

[14] Gerard Meurant. *Solution of Large Linear Systems*. ELSEVIER, 1998.

[15] NVIDIA Corporation. *CUDA C PROGRAMMING GUIDE*, 5 edition, October 2012.

List of Tables

7.1 Tested matrices. 49
7.2 The comparison of implementations of CG on CPU and GPU. . . 52
7.3 The comparison of implementations of PCG on CPU and GPU. . 53
7.4 Dependence between times of decomposition and numbers of parts. 54
7.5 Comparison between BAINV and AINV(10^{-2}). 55
7.6 Comparison between BAINV and SAINV(10^{-2}). 55
7.7 The CPU and GPU implementation of the block inverse factorization. 56
7.8 Comparison of CPU and GPU implementation of SAINV(10^{-2}). . 57
7.9 Comparison of CPU and GPU implementation of SAINV(10^{-1}). . 58