UNIVERSITY OF ZAGREB

FACULTY OF SCIENCE

DEPARTMENT OF MATHEMATICS

Anastasia Kruchinina

PARALLELIZATION OF THE SYMMETRIC INDEFINITE FACTORIZATION

Diploma Thesis

Advisor:
Prof. dr. sc. Sanja Singer

Zagreb, July, 2013

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

To my mother and grandmother
For their love and encouragement

Contents

Introduction

Parallel computing is a topic that became very popular in the last few decades. Parallel computers are being used in many different areas of science such as astrophysics, climate modelling, quantum chemistry, fluid dynamics and medicine. Parallel programming is a type of programming where computations can be performed concurrently on different processors or devices. There are two different approaches to parallel computing. One of them is moving the sequential program to multiple cores, which optimized for execution of the sequential code. The other one is the developing of programs for many-threads processors with a large number of light weight threads, such as **graphics processing units (GPUs)**. There is a big difference in the design of many-threads GPUs and general-purpose CPUs. Depending on the problem we choose one of these architectures. The main purpose of this study is to develop an understanding of parallel programming on GPUs and implement an algorithm for symmetric indefinite factorization for CUDA-enabled GPUs.

Symmetric indefinite matrices are matrices with both positive and negative eigenvalues. They are very important and they arise in many field of science. Some of those fields are nonlinear optimization problems that use Newton's method, certain methods in nonlinear programming, the augmented system of general least squares problems, discretized incompressible Navier–Stokes equations, and control theory.

The **symmetric indefinite factorization** was introduced by Bunch and Parlett [4]. Pivots of size 1×1 and 2×2 are used to reduce symmetric indefinite matrix A to block-diagonal form:

$$PAP^T = MDM^T, \tag{1}$$

where P is a permutation matrix, M is a lower triangular matrix with 1's on the diagonal and D is a block-diagonal with 1×1 and 2×2 blocks. Bunch and Parlett also suggest to use complete pivoting to ensure the stability of the factorization. Furthermore, Bunch and Kaufman [3] developed a partial pivoting strategy which is less complex, but Ashcraft, Grimes, and Lewis [1] have shown that it does not provide a bound for elements of matrix M. In the same paper Ashcraft, Grimes, and Lewis introduced an improved version of the Bunch–Kaufman pivoting which is usually called the bounded Bunch–Kaufman or rook strategy.

One of the most important problems is the problem of finding solutions for linear systems. For solving linear system $Ax = b$, where matrix A is a symmetric indefinite, we can use Gaussian elimination with pivoting to ensure stability. We find the LU factorization of matrix A and after that we have to solve two triangular systems. The stability of the factorization depends on the pivoting strategy which we use, and it costs $O(n^3)$. Solving triangular systems costs $O(n^2)$. Gaussian elimination does not take advantage on the symmetry of the matrix A, because after the first stage of elimination matrix is no longer symmetric. We are looking for a method which can use the symmetry to reduce cost and storage.

Symmetric indefinite factorization is used for solving such systems. To obtain solution we have to find the symmetric indefinite factorization, solve two triangular systems and solve several 2×2 systems for 2×2 blocks of the matrix D (for example by using the Gaussian elimination). Therefore the complexity is again $O(n^3)$, but because of the symmetry amount of operations and storage is halved.

By using the symmetric indefinite factorization we can also obtain the inertia of the matrix. The *inertia* of the matrix is a triple (n_+, n_-, n_0) where n_+ is the number of positive, n_- of the negative and n_0 of equal to zero eigenvalues. Every 2×2 block of the matrix D is indefinite and therefore corresponds to one negative and one positive eigenvalues of the original matrix A. In the factorization $\mathrm{MDM^T}$ if the matrix D has p_+ positive 1×1 diagonal blocks, p_- negative 1×1 diagonal blocks, p_0 zero 1×1 diagonal blocks, and q 2×2 diagonal blocks, then the inertia of matrix A is $(n_+, n_-, n_0) = (p_+ + q, p_- + q, p_0)$.

Combined with the symmetric indefinite factorization, the block J-Jacobi methods become accurate and efficient eigensolvers for symmetric indefinite matrices [7]. The first phase of the eigensolver computes the symmetric indefinite factorization (1) of the initial matrix A. An additional diagonalization of the diagonal blocks of D and the appropriate scaling of the columns of M yield

$$PAP^T = GJG^T,$$

where G is a non-singular lower block-triangular with diagonal 1×1 and 2×2 blocks, and $J = \mathrm{diag}(j_1, \ldots, j_n)$, where $j_i \in \{-1, 1, 0\}$. Matrix J gives as an inertia of the original matrix A.

Let's consider the eigenproblem $Ax = \lambda x$ and premultiply it from the left with the matrix G^T we obtained in the first phase. We get $G^TGz = \lambda Jz$ where $z = JG^Tx$. Thus the second part of the eigensolver computes the eigenvalues and eigenvectors of the positive definite pair (A, J) where $A = G^TG$ is positive definite and J is diagonal matrix of signs. This is equivalent to solving eigenproblem for J-symmetric matrix JG^TG. The eigenvalues of JG^TG and of the pair (G^TG, J) are the non-zero eigenvalues of A. To compute this eigenvalues Hari, Singer and Singer [7] suggest to use the hyperbolic singular value decomposition of G by using one-sided version of the Jacobi-type algorithm by Veselić [15].

The overall structure of this study has four chapters plus this introductory chapter. Chapter 1 begins by laying out the symmetric indefinite factorization and the overview of three popular pivoting strategies: complete, partial and rook. Chapter 2 presents an overview of the error analysis of the factorization with complete pivoting. The explanation and details about our implementation of the symmetric indefinite factorization on the GPU is given in Chapter 3. Finally, Chapter 4 gives a brief summary and outlook on the possible future work.

I would like to thank my scientific advisor prof. dr. sc. Sanja Singer for her guidance and support, to Vedran Novaković for his helpful advices and explanations, and to all the people who helped and motivated me.

Chapter 1

Symmetric indefinite factorization

In this Chapter we introduce the symmetric indefinite factorization for the dense symmetric indefinite matrices. We will explain the construction of the factors and its properties. Also we will consider three different pivot strategies for choosing permutation of the original matrix which assures stability of the factorization.

1.1 Algorithm

Let us consider the $\mathrm{LDL}^{\mathrm{T}}$ factorization of the symmetric matrix $A = LDL^T$, where L is lower triangular matrix with 1's on the diagonal and D is the diagonal matrix:

$$\begin{bmatrix} -2 & 4 \\ 4 & -7 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ -2 & 1 \end{bmatrix} \begin{bmatrix} -2 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & -2 \\ 0 & 1 \end{bmatrix}.$$

For every *symmetric positive definite* matrix this factorization exists and it is unique and stable without pivoting. The problem is that for *indefinite* matrices this decomposition becomes unstable or does not even exist because of the impossibility of bringing large off-diagonal element to the pivot position by symmetric permutations. For instance, decomposition fails on this simple example of 2×2 matrix:

$$A = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}.$$

Interesting enough, that problem can be solved by a rather small increasing of the complexity, by allowing 2×2 blocks in the matrix D. Therefore, we use block factorization $\Pi A \Pi^T = MDM^T$ called the **symmetric indefinite factorization**, where Π is a permutation matrix, M is a lower triangular, and D is a block diagonal. Whenever the matrix D has 2×2 block, matrix M has the identity matrix at the same position (see Figure 1.1). Elements of the matrix M are usually called **multipliers**.

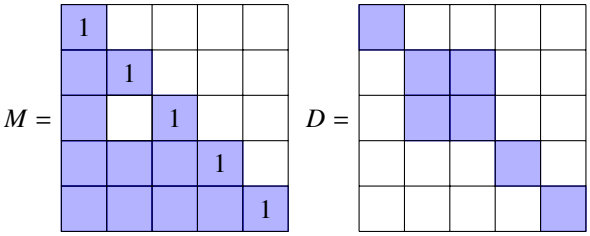


Figure 1.1: Matrices M and D in the symmetric indefinite factorization

Let $A \in \mathbb{R}^{n\times n}$ be a nonzero and symmetric. Therefore, we are able to find a permutation matrix Π so that P is non-singular matrix of order 1 or 2 and so that

$$\Pi A \Pi^T = \begin{bmatrix} P & C^T \\ C & B \end{bmatrix}. \tag{1.1}$$

Matrix P is usually called a **pivot** (analogy with Gaussian elimination) and there are different pivot strategies for choosing it. In the Section 1.2 we will consider some of the most popular pivot strategies. Once we choose Π, we can factorize A as

$$\Pi A \Pi^T = \left[\begin{array}{c|c} I_s & 0 \\ \hline CP^{-1} & I_{n-s} \end{array}\right] \left[\begin{array}{c|c} P & 0 \\ \hline 0 & B - CP^{-1}C^T \end{array}\right] \left[\begin{array}{c|c} I_s & 0 \\ \hline CP^{-1} & I_{n-s} \end{array}\right]^T. \tag{1.2}$$

The process of choosing the permutation (1.1) and the factorization (1.2) of the matrix is repeated recursively on the Schur complement $B - CP^{-1}C^T$.

Let denote $A^{[n]} := A$ and reduced matrix in the $(n-k)$ stage with $A^{[k]}$. One stage of the factorization of the $A^{[k]}$ is presented in Algorithm 1.

Algorithm 1: One stage of the symmetric indefinite factorization

$A^{[k]}$ is reduced matrix;
begin
    Choose 1×1 or 2×2 pivot P by some pivoting strategy;
    Find multipliers $C^{[k]}(P^{[k]})^{-1}$;
    Find Schur complement $B - C^{[k]}(P^{[k]})^{-1}(C^{[k]})^T =: A^{[k-s]}$ for $s = 1$ or 2;

The cost of the whole factorization *without pivoting* is approximately $n^3/3$ and this is the same as the cost of the Cholesky factorization of positive definite matrices. But here we must add the cost of identifying and performing permutation, and this cost can be significant.

This factorization is used for dense symmetric matrices because it does not preserve sparsity. There are many examples of sparse matrices where only after one stage of the

factorization, these matrices will have a lot of nonzero elements. For more information on algorithms for symmetric indefinite factorization for sparse matrices see Ashcraft, Grimes, and Lewis [1].

1.2 Pivot strategies

There exists a large number of pivoting strategies for the symmetric indefinite factorization with different complexities and stability properties. In this section we will briefly discuss three pivot strategies: complete, partial and rook. Here we can see the analogy with Gaussian elimination. Complete pivoting needs $O(n^3)$ comparisons, partial strategy $O(n^2)$ comparisons and rook between $O(n^2)$ and $O(n^3)$ depending on the initial problem. All of them are backward stable, but only complete and rook strategies provide bound for multipliers, i.e., elements of the matrix M.

Complete (Bunch–Parlett) pivoting

Complete pivoting was devised by Bunch and Parlett [4]. Let us define μ_1 as the maximal diagonal element by absolute value, and μ_0 as the maximal element by absolute value in the whole matrix (see Figure 1.2).

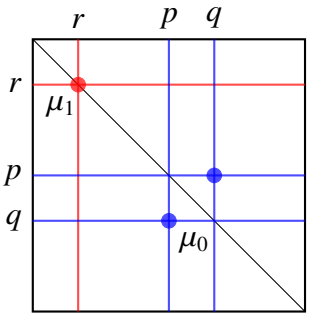
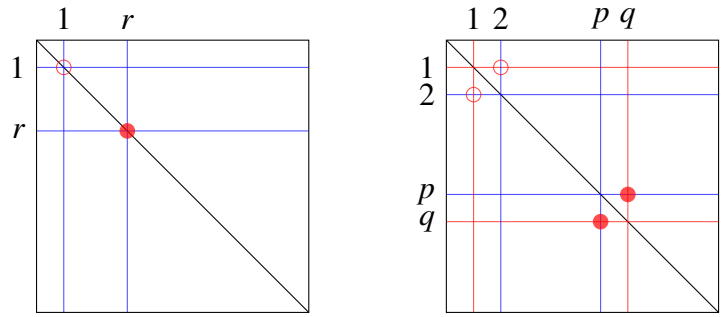


Figure 1.2: Looking for the pivot

We want to choose 1×1 *diagonal pivot* μ_1 whenever it is possible, and therefore we will choose it when μ_1 is not much smaller then *complete pivot* μ_0. Otherwise, we need to choose the complete pivot to avoid large element growth in the reduced matrix, in regard to the original matrix A (for more details see Chapter 2).

Since it is impossible to bring complete pivot element at pivot position $(1, 1)$ by symmetric permutations, we choose 2×2 pivot which has complete pivot as the off-diagonal element $(1, 2)$ (see Figure 1.3). Thus we have the complete pivoting presented in Algorithm 2, where parameter $0 < \alpha < 1$.



(a) 1×1 pivot (b) 2×2 pivot

Figure 1.3: Permutation of the matrix after choosing the pivot

Properties:

- backward stable (see more in Chapter 2),
- elements of the matrix M are bounded by $\max\{1/\alpha, 1/(1-\alpha)\}$,
- diagonal blocks in the matrix D are well conditioned,
- it searches the whole reduced matrix at each stage, thus requires a total of $O(n^3)$ comparisons.

Algorithm 2: Complete pivoting

$A^{[k]}$ is reduced matrix;
$0 < \alpha < 1$;
begin
  Find $\mu_1 = \max_i |A_{ii}^{[k]}|$ and the least integer r such that $|A_{rr}^{[k]}| = \mu_1$;
  Find $\mu_0 = \max_{ij} |A_{ij}^{[k]}|$ and the least row number p and the least column number q in the pth row such that $|A_{pq}^{[k]}| = \mu_0$;
  if $\mu_1 > \alpha\mu_0$ **then**
    Use 1×1 pivot $A_{rr}^{[k]}$;
    Interchange row and column 1 with r so that $P^{[k]}$ is maximal diagonal element μ_1;
  else
    Use 2×2 pivot $\begin{bmatrix} A_{pp}^{[k]} & A_{pq}^{[k]} \\ A_{qp}^{[k]} & A_{qq}^{[k]} \end{bmatrix}$
    Interchange rows and columns 1 with p and 2 with q, so that $P^{[k]}$ has a maximal off-diagonal element μ_0 on position $(1, 2)$;

Partial (Bunch–Kaufman) pivoting

Partial pivoting strategy for the symmetric indefinite factorization was devised by Bunch and Kaufman [3] and it is implemented in LAPACK as routine `xSYTRF`. It searches at most two columns at each stage, thus it requires in total only $O(n^2)$ comparisons. The whole strategy is given in Algorithm 3.

Algorithm 3: Partial pivoting

$A^{[k]}$ is reduced matrix;
$0 < \alpha < 1$;
begin
  Find $\gamma_1 = \max_{i \neq 1} |A_{i1}^{[k]}|$ and the least integer $r \neq 1$ such that $|A_{r1}^{[k]}| = \gamma_1$;
  if $\gamma_1 == 0$ **then**
    Go to the next stage; /* (0) */
  if $|A_{11}^{[k]}| \geq \alpha\gamma_1$ **then**
    Use 1×1 pivot $A_{11}^{[k]}$; /* (1) */
    No interchanges required;
  else
    Find $\gamma_r = \max_{i \neq r} |A_{ir}^{[k]}|$;
    if $|A_{11}^{[k]}|\gamma_r \geq \alpha\gamma_1^2$ **then**
      Use 1×1 pivot $A_{11}^{[k]}$; /* (2) */
      No interchanges required;
    else if $|A_{rr}^{[k]}| \geq \alpha\gamma_r$ **then**
      Use 1×1 pivot $A_{rr}^{[k]}$; /* (3) */
      Interchange rows and columns 1 with r;
    else
      Use 2×2 pivot $\begin{bmatrix} A_{11}^{[k]} & A_{1r}^{[k]} \\ A_{r1}^{[k]} & A_{rr}^{[k]} \end{bmatrix}$; /* (4) */
      Interchange rows and columns 2 with r;

In the Bunch–Kaufman pivoting strategy we are again looking for the diagonal pivot (1×1 pivot) whenever it is possible. We choose the diagonal pivot when it is not much smaller then off-diagonal elements in the same row or column. These are the cases (1) and (3) in the algorithm (see Figure 1.4). The case (2) may seem strange, but it is the condition which is needed to prevent the growth of elements in reduced matrix in the case (4).

Higham [8] has given an analysis of the Bunch–Kaufman pivoting and has showed that it is backward stable. One issue is that the elements of M are not bounded. The problem

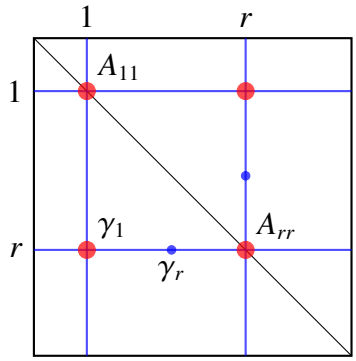


Figure 1.4: Possible pivots in Bunch–Kaufman strategy

arises when we compute the upper bound on elements of M for cases (2) and (4). The new column of M in case (2) (or first new column of M in case (4)) is bounded by a multiple of ratio γ_r/γ_1 and there is no upper bound on it, so upper bound on elements of M does not exist. The stability without bound on elements of M requires that large entries of M always have to be scaled by small entries of D. As Ashcraft, Grimes and Lewis have noted, the unusual aspect of the algorithm ensures precisely this. Therefore, elements of 2×2 blocks of the matrix D produced in case (4) can be arbitrarily badly conditioned.

Properties:

- backward stable,
- no bound for elements of the matrix M,
- blocks in the matrix D can be badly conditioned,
- needs $O(n^2)$ comparisons.

Rook (bounded Bunch–Kaufman) pivoting

In some applications it is required to have bounded matrix M (see the modified Cholesky algorithm [5]), but partial pivoting strategy does not provide this bound. To fix this problem Ashcraft, Grimes, and Lewis devised rook pivoting strategy [1]. In their strategy authors wanted to bound the elements of M, thus they put in the condition $\gamma_r/\gamma_1 = 1$.

The algorithm is presented in Algorithm 4. In this strategy we are again looking for the diagonal pivot whenever it is possible, therefore we have cases (1) and (2) in the algorithm. If we are not lucky enough to find the pivot in these cases and the condition $\gamma_r/\gamma_1 = 1$ is not true in the case (3), we continue to search. So, we allow 2×2 block just when we are able to obtain the upper bound on the elements of M.

We are searching for *local maximum* off-diagonal element in reduced matrix, i.e., an off-diagonal element $A_{ri}^{[k]}$ that is simultaneously the largest in magnitude in both the rth row

and the ith column. Ashcraft, Grimes and Lewis have conducted the probabilistic analysis of the number of stages which are needed to find a local maximum off-diagonal element of a random matrix and have shown that it is bounded above by en, where $e \approx 2.71828$.

Properties:

- backward stable,
- elements of the matrix M are bounded,
- blocks in the matrix D are well conditioned,
- needs between $O(n^2)$ and $O(n^3)$ comparisons.

Algorithm 4: Rook pivoting

$A^{[k]}$ is reduced matrix;
$0 < \alpha < 1$;
begin
  Find $\gamma_1 = \max_{i \neq 1} |A_{i1}^{[k]}|$ and the least integer $r \neq 1$ such that $|A_{r1}^{[k]}| = \gamma_1$;
  if $\gamma_1 == 0$ **then**
    Go to the next stage; /* (0) */
  if $|A_{11}^{[k]}| \geq \alpha\gamma_1$ **then**
    Use 1×1 pivot $A_{11}^{[k]}$; /* (1) */
    No interchanges required;
  else
    $i = 1$;
    while *pivot is not found* **do**
      Find the least integer $r \neq i$ such that $|A_{ri}^{[k]}| = \gamma_i$;
      Find $\gamma_r = \max_{j \neq r} |A_{jr}^{[k]}|$;
      if $|A_{rr}^{[k]}| \geq \alpha\gamma_r$ **then**
        Use 1×1 pivot $A_{rr}^{[k]}$; /* (2) */
        Interchange rows and columns 1 with r;
      else if $\gamma_i == \gamma_r$ **then**
        Use 2×2 pivot $\begin{bmatrix} A_{ii}^{[k]} & A_{ir}^{[k]} \\ A_{ri}^{[k]} & A_{rr}^{[k]} \end{bmatrix}$; /* (3) */
        Interchange rows and columns 1 with i and 2 with r;
      else
        $i = r$, $\gamma_i = \gamma_r$;

Chapter 2

Error analysis of the complete pivoting

In Chapter 1 we have considered three pivoting strategies for the symmetric indefinite factorization. All of them were backward stable. In this Chapter we provide an overview of the analysis of the complete pivoting strategy given by Bunch [2]. We will find the bound of the error matrix and show why it is important to have bounded element growth in reduced matrices.

2.1 Floating-point arithmetic

Rounding errors are unavoidable consequences of working in finite precision arithmetic. Let $fl(\texttt{exp})$ denote the computed value of the expression `exp`. We consider the **standard model of accuracy** for basic arithmetic operations [9]:

$$fl(x \text{ op } y) = (x \text{ op } y)(1+\delta), \quad |\delta| \leq u, \quad \text{op} = +, -, *, /,$$

where δ is a relative error of performed operation and u is a unit round-off error which represents an upper bound on the relative error due to rounding in floating point arithmetic, i.e., every number lying in the range of floating point number system can by approximated by an element of this system with the relative error no larger than u. This model holds for most computers, in particular for IEEE standard arithmetic.

2.2 Backward error analysis

By performing computations in the finite precision arithmetic we obtain matrices M and D, which represent exact factors of a slightly perturbed matrix $A + F = MDM^T$, where F

is the error matrix. Algorithm is **backward stable** if there exists a matrix F which is small in some norm.

Let $A^{[n]} := A \in \mathbb{R}^{n\times n}$ be a nonzero symmetric matrix and $A^{[k]}$, $k = 1, \ldots, n$ a reduced matrix of dimension k in the $(n-k)$ stage of algorithm. If in some stage k we perform 2×2 pivoting, then matrix $A^{[n-k]}$ does not exist. We will use the array `pivot` to remember which pivots were used for each reduced matrix:

$$\texttt{pivot}[k] = \begin{cases} 1 & \text{if } A^{[k]} \text{ exists and for this reduced matrix we used } 1\times 1 \text{ pivot,} \\ 2 & \text{if } A^{[k]} \text{ exists and for this reduced matrix we used } 2\times 2 \text{ pivot,} \\ 0 & \text{if } A^{[k]} \text{ does not exist.} \end{cases}$$

If we compute all the elements of $A^{[k]}$ in finite precision arithmetic then, in general, matrix $A^{[k]}$ will not be symmetric due to round-off errors. We will perform computation only on the lower part of matrix $A^{[k]}$ and simply define $A^{[k]}_{ji} = A^{[k]}_{ij}$ for $i < j$. Therefore, the error matrix F will be symmetric too.

It is important to mention that the error matrix F resulting from performing the decomposition on the lower part of matrix A, in general, is *not equal* to matrix $\widetilde{F}$ resulting from performing the decomposition on the upper part of matrix A. However, Bunch [2] has shown that F and $\widetilde{F}$ obtain the same bound.

Now, we are going to bound the error matrix F in terms of the maximal elements of reduced matrices. We assume that the original matrix is already permuted so that no interchanges are required to decompose it with the complete pivoting.

Bounding error matrix F

First of all, let us combine two stages of the algorithm. Let s and t be pivots used for reduced matrices $A^{[k]}$ and $A^{[k-s]}$, respectively. We have $s := \texttt{pivot}[k]$ and $t := \texttt{pivot}[k-s]$. We define $M^{[k]} := C^{[k]}(P^{[k]})^{-1}$. Then

$$A^{[k]} = M^{[k]}D^{[k]}(M^{[k]})^T = \left[\begin{array}{c|c} I_s & 0 \\ \hline M^{[k]} & I_{k-s} \end{array}\right]\left[\begin{array}{c|c} P^{[k]} & 0 \\ \hline 0 & A^{[k-s]} \end{array}\right]\left[\begin{array}{c|c} I_s & (M^{[k]})^T \\ \hline 0 & I_{k-s} \end{array}\right]$$

$$= \left[\begin{array}{c|c|c} I_s & \multicolumn{2}{c}{0} \\ \hline M^{[k]} & I_t & 0 \\ \hline & M^{[k-s]} & I_{k-s-t} \end{array}\right]\left[\begin{array}{c|c|c} P^{[k]} & \multicolumn{2}{c}{0} \\ \hline 0 & P^{[k-s]} & 0 \\ \hline & 0 & A^{[k-s-t]} \end{array}\right]\left[\begin{array}{c|c|c} I_s & \multicolumn{2}{c}{(M^{[k]})^T} \\ \hline 0 & I_t & (M^{[k-s]})^T \\ \hline & 0 & I_{k-s-t} \end{array}\right]. \qquad (2.1)$$

In finite precision arithmetic we have errors in every stage of the calculation. Let $A^{[k]}_{exact}$ be the reduced matrix of order k which we obtain by performing *one stage* in exact arithmetic

and $A^{[k]}_{finit}$ be the reduced matrix which we obtain by performing *one stage* in finite precision. Therefore

$$A^{[k]}_{finit} = A^{[k]}_{exact} + F^{[k]} = \left[\begin{array}{c|c} P^{[k]} & (C^{[k]})^T \\ \hline C^{[k]} & A^{[k-s]} \end{array}\right] + \left[\begin{array}{c|c} 0_s & (\theta^{[k]})^T \\ \hline \theta^{[k]} & H^{[k-s]} \end{array}\right],$$

where $F^{[k]}$ is the error matrix which we obtain in this stage, $\theta^{[k]} \in \mathbb{R}^{(k-s)\times s}$ and $H^{[k-s]} \in \mathbb{R}^{(k-s)\times(k-s)}$. Similarly $A^{[k-s]}_{finit} = A^{[k-s]}_{exact} + F^{[k-s]}$.

Using (2.1) it is easy to see that in two stages we obtain the error

$$F^{[k]} + 0_s \oplus F^{[k-s]} = F^{[k]} + \left[\begin{array}{c|c} 0_s & 0 \\ \hline 0 & F^{[k-s]} \end{array}\right].$$

Furthermore we can generalize this result and obtain the error matrix F:

$$F = \sum_{\substack{k=1 \\ \texttt{pivot}[k]\neq 0}}^{n} 0_{n-k} \oplus F^{[k]} = \sum_{\substack{k=1 \\ \texttt{pivot}[k]\neq 0}}^{n} \left[\begin{array}{c|c} 0_{n-k} & 0 \\ \hline 0 & F^{[k]} \end{array}\right],$$

or elementwise:

$$|F_{ij}| \leq \sum_{\substack{k=n-j+2 \\ \texttt{pivot}[k]=1}}^{n} \max_{ij} |H^{[k-1]}_{ij}| + \sum_{\substack{k=n-j+3 \\ \texttt{pivot}[k]=2}}^{n} \max_{ij} |H^{[k-2]}_{ij}| + \begin{cases} \max\limits_{j} |\theta^{[n-j+1]}_{j}| & \text{if } \texttt{pivot}[n-j+1] = 1, \\ \max\limits_{j} |\theta^{[n-j+1]}_{1j}| & \text{if } \texttt{pivot}[n-j+1] = 2, \\ \max\limits_{j} |\theta^{[n-j+2]}_{2j}| & \text{if } \texttt{pivot}[n-j+1] = 0. \end{cases}$$

Bunch shows that the bounds of elements of the error matrix $F^{[k]}$ in one stage are equal:

- if $\texttt{pivot}[k] = 1$,

$$|F^{[k]}| \leq 2^{-t}\mu^{[k]}_0 \left[\begin{array}{c|ccccc} 0 & 1 & 1 & \cdots & 1 \\ \hline 1 & & & & \\ \vdots & & (1 + 3/\alpha + O(2^{-t}))J_{k-1} & & \\ 1 & & & & \end{array}\right], \tag{2.2}$$

where J_{k-1} is the matrix of order $k-1$ with all elements equal to 1,

- if $\texttt{pivot}[k] = 2$,

$$|F^{[k]}| \le 2^{-t}\mu_0^{[k]}\left[\frac{3(1+\alpha)}{1-\alpha}+O(2^{-t})\right]\left[\begin{array}{cc|ccccc} 0 & 0 & 1 & 1 & \cdots & 1 \\ 0 & 0 & 1 & 1 & \cdots & 1 \\ \hline 1 & 1 & & & & \\ \vdots & \vdots & & (12-\alpha)/(3(1+\alpha))J_{k-2} & & \\ 1 & 1 & & & & \end{array}\right], \tag{2.3}$$

where J_{k-2} is the matrix of order $k-2$ with all elements equal to 1.

From (2.2) and (2.3) we get for $i \ge j$

$$|F_{ij}|2^t \le \left(1+\frac{3}{\alpha}+O(2^{-t})\right)\sum_{\substack{k=n-j+2\\ \texttt{pivot}[k]=1}}^{n}\mu_0^{[k]}+\left(1+\frac{11}{1-\alpha}+O(2^{-t})\right)\sum_{\substack{k=n-j+3\\ \texttt{pivot}[k]=2}}^{n}\mu_0^{[k]}$$
$$+\begin{cases}\mu_0^{[n-j+1]} & \text{if } \texttt{pivot}[n-j+1]=1,\\ \mu_0^{[n-j+1]}\left[\dfrac{3(1+\alpha)}{1-\alpha}+O(2^{-t})\right] & \text{if } \texttt{pivot}[n-j+1]=2,\\ \mu_0^{[n-j+2]}\left[\dfrac{3(1+\alpha)}{1-\alpha}+O(2^{-t})\right] & \text{if } \texttt{pivot}[n-j+1]=0.\end{cases}$$

Note that

$$a\sum_{\substack{k=n-j+2\\ \texttt{pivot}[k]=1}}^{n}\mu_0^{[k]}+b\sum_{\substack{k=n-j+3\\ \texttt{pivot}[k]=2}}^{n}\mu_0^{[k]} \le (j-1)\max\{a,b/2\}\max_{n-j+2\le k\le n}\mu_0^{[k]},$$

so we obtain the following bound for error matrix F:

$$|F_{ij}| \le 2^{-t}\max_{n-j+2\le k\le n}\mu_0^{[k]}\left[(j-1)\max\left(1+\frac{3}{\alpha}+O\left(2^{-t}\right),\frac{1}{2}\left\{1+\frac{11}{1-\alpha}+O\left(2^{-t}\right)\right\}\right)\right.$$
$$\left.+\begin{cases}1 & \text{if } \texttt{pivot}[n-j+1]=1,\\ \dfrac{3(1+\alpha)}{1-\alpha}+O\left(2^{-t}\right) & \text{if } \texttt{pivot}[n-j+1]\ne 1.\end{cases}\right] \tag{2.4}$$

$$\|F\|_1 \le 2^{-t}\max_{1\le k\le n}\mu_0^{[k]}\left[\frac{1}{2}n^2\max\left(1+\frac{3}{\alpha}+O(2^{-t}),\frac{1}{2}\left\{1+\frac{11}{1-\alpha}+O(2^{-t})\right\}\right)\right.$$
$$\left.+\frac{1}{2}n\left(\frac{3(1+\alpha)}{1-\alpha}+O\left(2^{-t}\right)\right)\right]. \tag{2.5}$$

Bound on element growth

Bounds (2.4) and (2.5) show the importance of preventing rapid growth of the elements in the reduced matrix. We are looking for bounds on elements of the reduced matrix $A^{[k]}$. As before, we define maximal by absolute value element in the reduced matrix with $\mu_0^{[k]} = \max_{ij} |A_{ij}^{[k]}|$ and maximal by absolute value diagonal element with $\mu_1^{[k]} = \max_i |A_{ii}^{[k]}|$. Maximal elements in the original matrix are $\mu_0 := \mu_0^{[n]}$ and $\mu_1 := \mu_1^{[n]}$.

We consider both cases: when pivot matrix P is of order $s = 1$ and $s = 2$. Again, we assume that all needed interchanges of rows and columns have already been made:

$$A^{[k]} = \left[\begin{array}{c|c} P & C^T \\ \hline C & A^{[k-s]} \end{array}\right] = \left[\begin{array}{c|c} I_s & 0 \\ \hline CP^{-1} & I_{k-s} \end{array}\right] \left[\begin{array}{c|c} P & 0 \\ \hline 0 & A^{[k-s]} \end{array}\right] \left[\begin{array}{c|c} I_s & 0 \\ \hline CP^{-1} & I_{k-s} \end{array}\right]^T. \tag{2.6}$$

• If `pivot`$[k] = 1$,

matrix P is of order 1 and $|P| = \mu_1^{[k]} > 0$.
From (2.6) multipliers and elements of the reduced matrix $A^{[k-1]}$ are bounded with

$$\max_i |(CP^{-1})_i| \le \frac{\mu_0^{[k]}}{\mu_1^{[k]}}, \tag{2.7}$$

$$\max_{ij} |A_{ij}^{[k-1]}| \le \mu_0^{[k]} \left(1 - \frac{\mu_0^{[k]}}{\mu_1^{[k]}}\right). \tag{2.8}$$

So, we can use the 1×1 pivot only if $\mu_1^{[k]}/\mu_0^{[k]}$ is bounded away from zero.

• If `pivot`$[k] = 2$,

matrix P is of order 2 and $|\det P| =: \nu > 0$.
From (2.6) multipliers and elements of the reduced matrix $A^{[k-2]}$ are bounded with

$$\max_{il} |(CP^{-1})_{il}| \le \mu_0^{[k]} \frac{\mu_0^{[k]} + \mu_1^{[k]}}{\nu}, \quad \text{for } l = 1, 2, \tag{2.9}$$

$$\max_{ij} |A_{ij}^{[k-2]}| \le \mu_0^{[k]} \frac{1 + 2\mu_0^{[k]}(\mu_0^{[k]} + \mu_1^{[k]})}{\nu}. \tag{2.10}$$

From above we can see that we can use the 2×2 pivot only if ν is bounded away from zero.

Therefore, we need to bound ν from below when we are not able to use the 1×1 pivot, i.e., when $\mu_1^{[k]}/\mu_0^{[k]}$ is near zero. The next theorem says that the complete pivoting strategy provides us the necessary bound.

Theorem 2.2.1. *Let A be a symmetric matrix, and let $\mu_0 = \max_{ij} |A_{ij}|$ and $\mu_1 = \max_i |A_{ii}|$. If in one stage of the method with complete pivoting strategy we use pivot matrix P of order 2, then*

$$\mu_0^2 - \mu_1^2 \leq |\det P| \leq \mu_0^2 + \mu_1^2.$$

Proof. Let

$$P = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix},$$

where $A_{12} = A_{21} = \mu_0 > \mu_1$.
Therefore,

$$|\det P| = |A_{11}A_{22} - A_{12}^2| = \mu_0^2 - A_{11}A_{22} \geq \mu_0^2 - \mu_1^2,$$

and

$$|\det P| = |A_{11}A_{22} - A_{12}^2| \leq |A_{11}A_{22}| + |A_{12}^2| \leq \mu_0^2 + \mu_1^2.$$ □

We want to find a number α where $0 < \alpha < 1$ whose usage in the complete pivoting strategy will provide us the best bound on the element growth (see Section 1.2). We will choose the 1×1 pivot whenever $\mu_1^{[k]}/\mu_0^{[k]} \geq \alpha$ and the 2×2 pivot otherwise.

Let us look on bounds which we can obtain from (2.7), (2.9), (2.8) and (2.10) by doing two consecutive stages with 1×1 pivots and one stage with the 2×2 pivot:

$$\begin{aligned} &\text{for } 1 \times 1 \text{ pivot:} \qquad \mu_0^{[k]} \leq \mu_0^{[k+2]} \left(1 + \frac{1}{\alpha}\right)^2, \\ &\text{for } 2 \times 2 \text{ pivot:} \qquad \mu_0^{[k]} \leq \mu_0^{[k+2]} \left(1 + \frac{2}{1-\alpha}\right). \end{aligned} \tag{2.11}$$

If $\alpha = 0$ (this case corresponds to the use of a pivot of order 1 in each iteration) or $\alpha = 1$ (this case corresponds to the use of a pivot of order 2 in each iteration) then the bound goes to infinity so the both cases are unstable. Let us find the optimal α to reduce the element growth.

We seek for a minimum

$$G(\alpha) = \min_{0<\alpha<1} \max \left\{ \left(1 + \frac{1}{\alpha}\right)^2, 1 + \frac{2}{1-\alpha} \right\}.$$

The minimum will be reached if the terms on the right-hand side are equal, i.e., if

$$\left(1 + \frac{1}{\alpha}\right)^2 = \left(1 + \frac{2}{1-\alpha}\right).$$

The positive root of this quadratic equation is $\alpha_0 = \frac{1+\sqrt{17}}{8}$, so the minimum $G(\alpha_0)$ is $\frac{9+\sqrt{17}}{2}$.

Now we are able to **bound the elements of reduced matrices** using (2.11) for $\alpha_0 = \frac{1+\sqrt{17}}{8}$:

$$\mu_0^{[k]} \leq \mu_0^{[k+2]} \left(\frac{9+\sqrt{17}}{2} \right) \leq \mu_0 \left(\frac{9+\sqrt{17}}{2} \right)^{\frac{n-k}{2}} < \mu_0 (2.57)^{n-k}.$$

This bound is overestimating. Bunch in [2] computed a better bound on elements in all the reduced matrices:

$$\max_k \max_{ij} |A_{ij}^{[k]}| < \sqrt{n} f(n) \mu_0 (3.07)(n-1)^{0.446}, \quad f(n) = \prod_{k=2}^{n} k^{1/(k-1)}.$$

Interesting enough that the obtained bound is no more than $(3.07)(n-1)^{0.446}$ times larger than the bound for LU factorization with complete pivoting.

Complete pivoting strategy also provides **bound on multipliers**. Note that this bound does not depend on the original matrix, just on the parameter α we chose. Using (2.7) and (2.9) we obtain for $\alpha_0 = \frac{1+\sqrt{17}}{8}$:

$$\max_{ij} |M_{ij}| \leq \begin{cases} \dfrac{1}{\alpha} < 1.562 & \text{for } 1 \times 1 \text{ pivot,} \\ \dfrac{1}{1-\alpha} < 2.781 & \text{for } 2 \times 2 \text{ pivot.} \end{cases}$$

From the analysis above we can conclude that α_0 is the best candidate for the parameter α. Therefore, in our implementation we will use $\alpha = \alpha_0$. For further comments on this topic, see Bunch [2], and Bunch and Parlett [4].

Chapter 3

Implementation

Graphics processing unit (GPU) is a device for PC or game console that performs graphic rendering. Modern GPUs process and display computer graphics very efficiently. Due to the specialized pipelined architecture, they process graphical information much more effectively than typical CPU.

GPU can be used together with a CPU to accelerate general-purpose scientific and engineering applications. NVIDIA cards support API extensions to the C programming language [14] such as CUDA (Compute Unified Device Architecture) which allows CPU-based applications to access directly the resources of a GPU for more general-purpose computing without the limitations of using a graphics API.

In this Chapter we will give details about our implementation of the symmetric indefinite factorization with complete pivoting on GPU with CUDA and an overview of the numerical results.

3.1 Parallelization on CUDA-enabled GPU

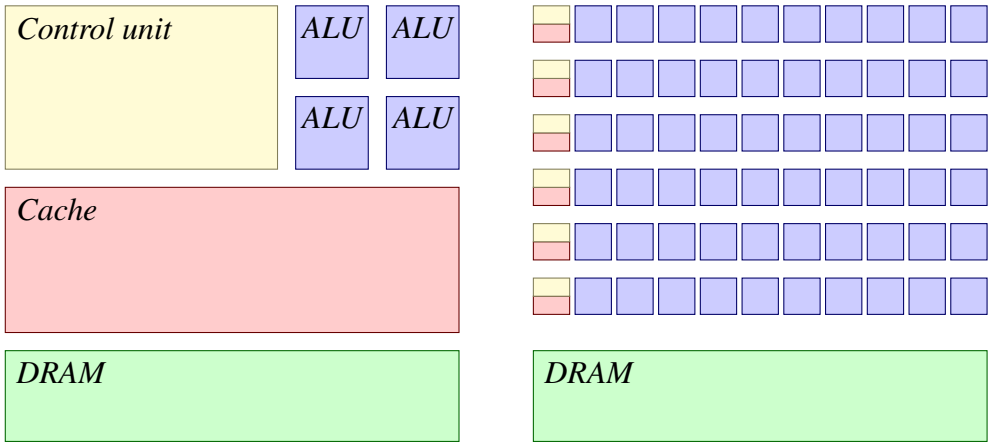
When we start working with GPU, we are faced with SIMT (Single Instruction–Multiple Thread) architecture, i.e., one instruction is performed by many independent **threads** on different data. The set of threads which are working on the same task is called **grid**. All of these threads are being executed by scalar processors (SP) or CUDA Cores, which are part of streaming multiprocessors (SM). The multiprocessor creates, manages, schedules, and executes threads in groups of 32 parallel threads called **warps**, which are parts of larger structures called **blocks**. For example, Fermi GPUs have 16 SMs and every one of them has 32 CUDA Cores.

Multiple thread blocks can execute concurrently on one SM. The threads of the one thread block can execute concurrently on just one SM, i.e., they cannot be divided between different SMs, and therefore have access only to its resources. Thread blocks never mi-

grate from one SM to another. In one moment SM executes exactly one warp and after its completion or while it is waiting for completing of some long-latency operation such as accessing global memory, SM is choosing another warp and starting to execute it. Therefore, threads in one warp are implicitly synchronized.

GPU and CPU

There are fundamental differences in design styles of GPU and CPU (see Figure 3.1).



(a) CPU (b) GPU

Figure 3.1: Differences in design styles of GPU and CPU

First, the CPU comes with large *caches*. The cache memory is on-chip memory and is designed to convert low latency memory accesses to short latency cache accesses by keeping as many as possible data in the cache. So, in many cases when we need to catch some data from DRAM which takes a long latency, we will find the needed data in the cache. CPU also possesses many important control mechanisms which help to reduce latency, for instance *branch prediction* and *data forwarding*. The branch prediction attempts to avoid waste of time waiting until the conditional jump instruction has passed the execute stage and tries to guess which way a branch will go before this is known for sure. Chosen branch is then fetched and partially executed. If the guess was wrong, instructions are discarded and the pipeline starts over with the correct branch, incurring a delay. Moreover, after producing operation, execution unit puts results to register file and then catches back this result for use by another operation. This takes some time. But with data forwarding technology we are able to use the result produced by previous instruction in the next clock cycle without going to register files. In addition, CPU uses powerful ALUs which

reduce operation latency. Because of these properties CPU design style is often referred as *latency-oriented* [11].

On the other side, the GPU design style is commonly referred to as *throughput-oriented* design. The GPU usually comes with very small caches which help to control bandwidth requirements so that all multiple threads that access the same memory location don't need go to the DRAM. GPU use energy efficient ALU and have a very simple control mechanisms, thus they have long operation latency and there are no techniques such as branch prediction or data forwarding. So, we have a large number of threads which take a potentially long time to execute. But hardware takes advantage of this situation and pushes some threads to work while others are waiting for long latency memory access or arithmetic operations.

Thus CPUs are great for task parallelism, i.e., to perform different calculations on the same or different sets of data, and GPUs are great for data parallelism when each scalar processor performs the same task on different pieces of distributed data. So CPU + GPU is a powerful combination because CPU consists of a few cores optimized for serial processing, while GPU consists of thousands of smaller, more efficient cores designed for parallel computations. Serial portions of the code run on the CPU while parallel portions run on the GPU. This is a reason why CUDA programming model supports both GPU and CPU executions of an application. CUDA C is an extension of the programming language C with additional keywords and application programming interfaces.

Traditionally CPU is called the **host** and GPU is called the **device**. The function that executes on the device and can be called from the host is called **kernel**. Actually, it executes as a grid of blocks of threads, which execute the same code on different data stored in GPU memory. Kernel must be declared with a specific qualifier (`__global__`, `__device__` or/and `__host__`), which determines which compilers are used to compile it.

The basic princip of programming on GPU with CUDA is following:

1. prepare data and define kernel functions,
2. load data from CPU to GPU,
3. start kernel function,
4. load data from GPU to CPU.

CUBLAS library

The *NVIDIA CUDA Basic Linear Algebra Subroutines (CUBLAS)* library is a GPU-accelerated version of the complete standard BLAS library. It is freely available as a part of the CUDA Toolkit. CUBLAS allows the user to access the computation resources of NVIDIA GPU. It supports all 152 standard routines for single, double, complex, and double complex precision plus extensions for writing and retrieving data from the GPU. Note

that the CUBLAS library uses column-major storage, and 1-based indexing for maximum compatibility with Fortran.

3.2 Implementation of complete pivoting

The symmetric indefinite factorization with complete pivoting presented in Algorithm 2 was implemented for CUDA-enabled GPU using CUBLAS library. The algorithm for symmetric indefinite factorization has two main parts which depend of each other and repeat iteratively: *choosing pivot* and *computing reduced matrix*.

Choosing pivot

Before implementing some algorithm we need to decide how to store data in the memory. The input for symmetric indefinite factorization is one matrix A of order n and the outputs are two factors M and D and permutation matrix Π.

Matrix M has 1's on the diagonal. As it was mentioned in Chapter 1 if D has 2×2 block on the diagonal then, in the corresponding place, matrix M has the identity matrix. Therefore, we can store both factors M and D into the same matrix. Additionally, in every stage we only need the reduced matrix, so computed parts of matrices M, D and the reduced matrix can overwrite the original matrix A. So, we need to allocate memory for just one matrix of order n and remember which pivot we were using in every stage.

One possibility is to store permutation vector and pivots in separate vectors, where the permutation vector is just a compact representation of the permutation matrix and vector of pivots is similar to vector `pivot` which was used in Chapter 2. In our implementation we use a different way to store the needed information. The similar approach was used in the implementation of symmetric indefinite factorization with Bunch-Kaufman pivot strategy in LAPACK function `DSYTRF`.

In every stage we are only working with the reduced matrix and interchange first row and column with another row and column in this reduced matrix (or two first rows and columns with another two rows and columns if we choose the 2×2 pivot). Therefore we need to store just numbers of rows and columns with which we made interchanges. To distinguish 2×2 from 1×1 pivots we add negative sign to the second number when we are using the 2×2 pivot.

For example, we made the following interchanges on a matrix of order 4:

1. 1×1 pivot: $1 \longleftrightarrow 3$,
2. 2×2 pivot: $2 \longleftrightarrow 2$ and $3 \longleftrightarrow 4$,
3. 1×1 pivot: $4 \longleftrightarrow 4$.

where $i \longleftrightarrow j$ means that we interchange rows and columns i and j. Then we obtain the following **vector of interchanges**: $I = \begin{bmatrix} 3 & 2 & -4 & 4 \end{bmatrix}^T$.

Therefore, as the result we obtain vector of interchanges by which we can easily obtain the permutation matrix Π such that $\Pi^T A \Pi = LDL^T$ and retrieve information about pivots. To obtain matrix Π we are going from the last element of vector I to the first (i.e., from the last interchange to the first) and in some way we reconstruct history of interchanges. For example, the permutation matrix Π obtained from vector of interchanges I is equal to

$$\Pi = \begin{bmatrix} 4 & 2 & 1 & 3 \end{bmatrix}^T$$

(see Figure 3.2). The advantage of this approach is that we are able to store all needed information in just one vector, rather than two.

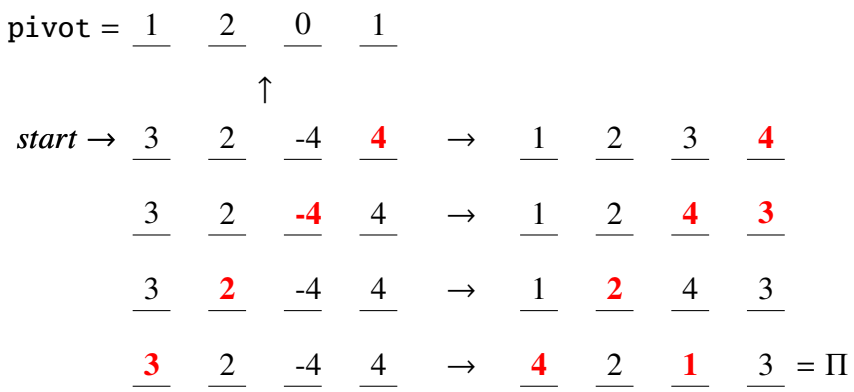


Figure 3.2: Restoring permutation vector Π and `pivot` vector

The next question is in what way we shall store a matrix in the GPU global memory. The matrix is symmetric so we are only using its lower or upper triangle. Therefore, we can store the whole squared matrix in GPU memory and work with just one triangle. Otherwise we can store matrix as a vector in a **packed format** (just one triangle stored column by column in one array). The second approach uses less memory but to access a random element in the matrix we need to compute its position in vector, which can significantly increase execution time.

We tried both approaches and by using the packed format we achieved a *significant speedup* in the contrast with the first whole-matrix storage format (more than 5 times on matrices of order 10000). In our implementation we did not need to access random elements in matrix very often and we used specialized functions for working with matrices in packed format provided by CUBLAS library. The matrix was stored in two parts: vector A which contained the whole matrix with zero diagonal (we will call it the matrix-vector) and the vector D which contained its diagonal (we will call it the diagonal-vector).

Working with a matrix in a packed format involves some complications during computing of the indexes. Here we provide functions for converting indexes of the whole matrix

to index of the matrix in a packed format (when the upper triangle of the matrix is stored row by row) and reverse:

```
__host__ __device__
int f2Dto1D( const int row, const int col, const int N)
{
  return row*(2*N-1-row)/2 + col; ;
}

__host__ __device__
void f1Dto2D( const int ind, const int N, int *rowA, int *colA)
{
  int row = 0, col;
  int count;
  double b=2*N+1, d;

  d = b*b - 8 * (ind-1);
  row = floor((b-sqrt(d))/2);
  *rowA = row;

  count = (2*N-row)*(row+1)/2;
  col = N-count+ind;

  *colA = col;
}
```

Listing 3.1: Functions for converting indexes

The packed storage format is very helpful for searching pivots. To find maximum by absolute value off-diagonal and diagonal elements in the reduced matrix, we just need two calls of CUBLAS function `cublasIdamax` on corresponding parts of matrix-vector and diagonal-vector.

After we know the indexes of maximum by absolute value off-diagonal and diagonal elements, we are able to decide which pivot to choose according to the complete pivoting.

Compute reduced matrix

In the previous part we explained how to choose a pivot. Now we will explain how to compute the reduced matrix. According to the chosen pivot we will do the following steps:

if the 1×1 *pivot P was chosen then*

1. swap rows and columns to put the pivot element on pivot position,

2. compute one rank-one update $A = A - PCC^T$ for the reduced matrix A,
3. compute multipliers.

if the 2×2 *pivot* P *was chosen then*

1. swap rows and columns to put pivot element on off-diagonal position in the 2×2 pivot,
2. compute eigendecomposition of the pivot matrix $QPQ^T = \Lambda$,
3. compute two rank-one updates $A = A - (CQ)\Lambda^{-1}(CQ)^T$ for the reduced matrix A,
4. compute multipliers.

Now we explain every step with more details. **Interchanges** of row p and column q are performed on GPU in three steps. For easier explanation, here we consider the whole squared matrix A of order n and represent needed interchanges using lower triangle of this matrix (see Figure 3.3):

- $A(p, i) \longleftrightarrow A(q, i)$ for $i = 1, \ldots, p-1$,
- $A(i, p) \longleftrightarrow A(i, q)$ for $i = q+1, \ldots, n$,
- $A(i, p) \longleftrightarrow A(q, i)$ for $i = p+1, \ldots, q-1$,
- $A(p, p) \longleftrightarrow A(q, q)$,

where $A(l, k) \longleftrightarrow A(n, m)$ means interchange of elements $A(l, k)$ and $A(n, m)$. In addition, for a matrix stored in the packed format we need to convert indexes using functions from Listing 3.1.

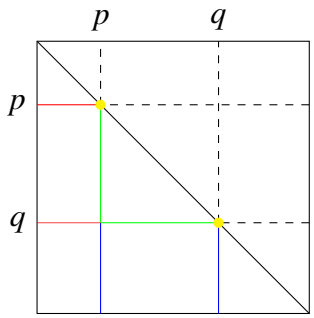


Figure 3.3: Interchanging rows and columns

We divide the process of **computing the reduced matrix** A if 2×2 pivot was chosen on three steps. Computing explicit inverse of P can be unstable, so we first find its spectral decomposition

$$QPQ^T = \Lambda = \begin{bmatrix} a & 0 \\ 0 & b \end{bmatrix},$$

and compute the reduced matrix A as two rank-one updates:

$$A = A - CP^{-1}C^T = A - (CQ)\Lambda^{-1}(CQ)^T = A - a(CQ)_{:1}(CQ)_{:1}^T - b(CQ)_{:2}(CQ)_{:2}^T,$$

where $(CQ)_{:i}$ for $i = 1, 2$ means ith column of the matrix CQ. Similarly for multipliers:

$$CP^{-1} = CQ\Lambda^{-1}Q^T = \begin{bmatrix} a(CQ)_{:1} & b(CQ)_{:2} \end{bmatrix} Q^T.$$

Rank-one update for matrices stored in packed format is implemented in function `cublasDspr` provided by CUBLAS library. To improve accuracy we decided to compute eigendecomposition on host by using LAPACK function `DLAEV2`. Therefore we need to load the pivot data from GPU to CPU and we are doing it by using *mapped memory*.

Mapped memory

For the host and all the devices of compute capability 2.0 and higher one single address space is used. This space is used for allocations made on the device or on the host via function `cudaHostAlloc`. Which memory (host or device) a pointer points is determined automatically. There are two types of memory which can be used to allocate memory on CPU: pinned and non-pinned memory. Pages of memory which is non-pinned can be moved from physical memory to disk or their location can be changed. Allocation stored in the pinned memory is always in physical memory and it will never be swapped out, so devices can fetch it without the help of CPU.

CUDA drivers use direct memory access (DMA) to transfer data between CPU and GPU. If data is allocated in the non-pinned memory, it is first transferred from the non-pinned memory to the buffer in the pinned memory and only after that to the GPU. Therefore, by using pinned memory we skip the first step of transferring data to the buffer in the pinned memory. The GPU memory size does not limit the size of the mapped host memory, thus it can be used when there is not enough memory on GPU. However the pinned memory is a limited resource. Any memory defined as being pinned must always be in RAM. Thus, that leaves less space in RAM for other system applications.

There is another advantage of the pinned memory. The block of the pinned memory can also be mapped into the address space of the device and provide asynchronous transparent access to the data without requiring an explicit programmer initiated copy. Such allocation can be done by putting `cudaHostAllocMapped` flag to `cudaHostAlloc` function, which returns pointer to allocated memory on the host. Device pointer to this mapped memory can be obtained by calling `cudaHostGetDevicePointer`.

Since the mapped pinned memory is shared between the host and the device, the application must synchronize memory accesses to avoid any potential errors caused by simultaneous memory access. Therefore, we need to put `cudaDeviceSynchronize` between kernel which writes data to the mapped memory and the function on CPU which uses written data:

```
1 write_to_mapped_mem<<<grid, block>>>(DEVICE_MAPPED_DATA, ...);
```

```
2 cudaDeviceSynchronize();
3 compute_pivot( CPU_MAPPED_DATA, ...);
```

Listing 3.2: Using mapped memory

There is also a danger that compiler will make some optimizations because it assumes that just the current CPU can change the value. Therefore, we must put the `volatile` keyword to the declaration of the array `CPU_MAPPED_DATA` in function `compute_pivot` in Listing 3.2 to prevent compiler to change the code (by assuming that the data does not change in ways it does not know about). Thus it will reload this array from memory every time it is read in code.

Hierarchical data format files (HDF5)

In our implementation we use hierarchical data format files (HDF5) for storing test matrices. HDF5 is a file format and a library designed to store and organize large amounts of data. We are working with matrices stored in the packed format and HDF5 library provides us the possibility for reading/writing just one triangle of the matrix column by column from/to the HDF5 file. HDF5 file consists of two major objects:

- **group** – a container of other groups and datasets,
- **dataset** – a multidimensional array of data of the same type.

Dataspace defines the size and shape of the dataset. In HDF5 there are three kinds of dataspaces: *scalar*, *simple* and *null*. We are interested in the simple dataspace because it defines a multidimensional array of elements. The dimensionality of the simple dataspace is fixed and is defined at the creation time. The size of the each dimension can grow during its lifetime, from the current size up to the maximum size specified during the creation. To define a simple dataspace one can use `H5Screate_simple`.

So, suppose we have a squared matrix stored in the HDF5 file. We wish to read just one triangle of the matrix and store it in the packed format. We have to do following:

1. define dataspaces in file and in memory,
2. select part of the dataset in file from which we want to read data and select part of the dataset in memory where we want to write data,
3. transfer data from selected part of the dataset in file to selected part of the dataset in memory.

We need to do similar steps for writing matrix into a file. There are two forms of selection provided by HDF5: *hyperslab* and *point*. **Hyperslab** allows us to select rectangle region in the dataset and point selection allows us to select individual points. An HDF5 hyperslab is a rectangular pattern defined by four arrays: offset (starting location), stride (distance between elements in the selection), count (number of the elements) and block (dimension of selected blocks). The function `H5Sselect_hyperslab` performs the selection.

We can do the reading or writing of the selected data by using function `H5Dread` and `H5Dwrite` provided by HDF5 library. Example of reading an upper triangle of matrix stored in the HDF5 file to matrix-vector *A* and diagonal-vector *D* is presented in Listing 3.3.

```
// reading NxN matrix from HDF5 file
// matrices A and D have been allocated before
int num = 0;
for( i = 0; i < N; ++i )
       {
         // selection in dataset in file
         offset[0] = i;
         offset[1] = i;
         count[0] = 1;
         count[1] = N-i;
         H5Sselect_hyperslab (dataspace, H5S_SELECT_SET, offset,
                         NULL, count, NULL);

         // selection in dataset in memory
         offset[0] = 0;
         offset[1] = num;
         count[0] = 1;
         count[1] = N-i;
         H5Sselect_hyperslab (memspace, H5S_SELECT_SET, offset,
                   NULL, count, NULL);

         // read data from one selection to another
         H5Dread (dataset, H5T_NATIVE_DOUBLE, memspace,
               dataspace, H5P_DEFAULT, A);
         D[i] = A[num];
         A[num] = 0;
         num += N-i;
       }
```

Listing 3.3: Reading matrix from HDF5 file and storing it in packed format

By reading just one half rather the whole matrix, we avoid the need to store the whole matrix into the memory, and read/write operations are *performed* 4 *to* 5 *times faster*.

3.3 Results

Algorithm for the symmetric indefinite factorization with complete pivoting was implemented for GPU using CUDA 5.5 RC. The host part was implemented in C. In this section we first present the performance analysis of our implementation which was made using the NVIDIA Visual Profiler. In the second part we give the obtained numerical results.

All the experiments to measure the performance of the implementation were conducted primarily on an NVIDIA Tesla S2050 graphics card with 3 GBs of memory, and Intel Xeon X5620 processor, 2.40 GHz with 4 dual-cores. Algorithm was coded in the C programming language and compiled with NVIDIA `nvcc` compiler, version 5.5.0, with `-O3` optimization level. All the references to LAPACK functions are actually references to the corresponding Intel MKL 11.0.4 implementations.

Profiler results

The NVIDIA Visual Profiler is a tool that allows us to visualize and optimize the performance of CUDA application. It provides a statistical summary of the events observed in the application and helps to identify the bottlenecks of the program. Visual Profiler constructs a timeline which contain a row for the each type of kernel executed by the application. Near the name of the each kernel is a label with the ratio of its execution time to the total execution time. Every row consists of intervals which indicate the total execution time of all instances of that kernel compared to the total execution time of all kernels. The timeline generated for our program with the random matrix of order 3000 input is presented in Figure 3.4.

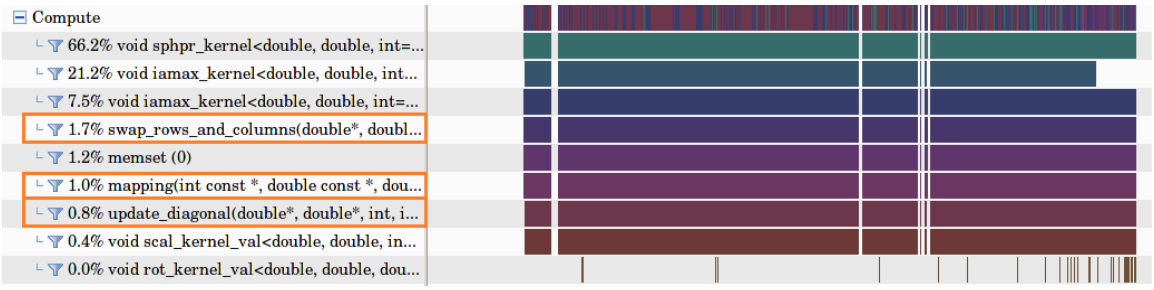


Figure 3.4: Timeline

Some of these kernels are from the CUBLAS library and we cannot influence on their performance. With the orange rectangles we denote our kernels. Note that the first two kernels in the timeline are the most expensive ones, together they require more than 80% of the total execution time. The first one of them performs rank-one updates and the second one computes maximum by absolute value off-diagonal element in reduced matrices. The

white lines crossing the timeline are periods without any useful work. They caused by function `CudaDeviceSynchronize`, which blocks the host thread until the device has completed all preceding requested tasks. This synchronization is needed before reading data from mapped memory.

The options `Zoom In`/`Zoom Out` allows us to see more details about kernel's execution time, for example we can consider just **one step** of the factorization (see Figure 3.5).

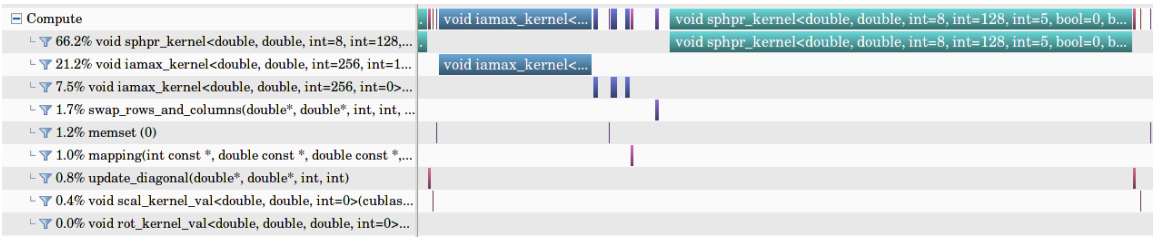


Figure 3.5: One step of the algorithm in timeline

Profiler can collect different **performance counters**. In a single run it can collect only a few of them. For more counters we need multiple runs, which is done by Visual Profiler automatically. In Figure 3.6 we give the summary counters for all kernels in our implementation. Again, we denote our kernels with the orange rectangle.

Details (Summary)

Name	Branch Efficiency	Warp Execution Efficiency	Global Memory Load Efficiency	Global Memory Store Efficiency
void iamax_kernel<double, double, int=256, int=1>(cublasIam	99.1%	99.1%	0%	18.8%
void iamax_kernel<double, double, int=256, int=0>(cublasIam	91.6%	85.2%	40.9%	18.8%
void sphpr_kernel<double, double, int=8, int=128, int=5, bool=	99.8%	99%	51.1%	84.1%
void scal_kernel_val<double, double, int=0>(cublasScalParam	99.7%	99.7%	51.9%	85%
void rot_kernel_val<double, double, double, int=0>(cublasRotI	99.8%	99.6%	40.9%	82.9%
update_diagonal(double*, double*, int, int)	100%	99.2%	11.2%	38.6%
swap_rows_and_columns(double*, double*, int, int, int)	98.3%	98.1%	15.8%	47.1%
mapping(int const *, double const *, double const *, int, int, dou	100%	69.5%	44.1%	400%

Figure 3.6: NVIDIA Visual Profiler counters

Warp execution efficiency is a ratio of the average active threads per warp to the maximum number of threads per warp supported on a multiprocessor. *Branch efficiency* is a measure of how many branches diverged in a warp. So for kernels `update_diagonal` and `mapping` we can conclude that there is no divergent branches in a warps. *Global store/read memory efficiency* counter gives the ratio of the global memory store/load throughput requested by the kernel to the actual global memory store/load throughput required for the kernel. For the `mapping` kernel global store memory efficiency is equal 400%, that means that requested by kernel store throughput is four times more than actual store throughput. It happens because some accesses to global memory are coalesced or combined to the single access.

Numerical results

For testing purposes, random symmetric indefinite matrices of order n, with equidistant spectra in interval $[-n, n-1]$, were generated in MATLAB [12].

Sequential implementation of the factorization was implemented in *C* using LAPACK. It was compiled by Intel `icc` compiler, version 13.1.2, with the optimization level `-O3`. Timing and achieved speedup of the parallel implementation for these random matrices, in the relation to the sequential one, is given in Figure 3.7, where n denotes the dimension of the test matrix. The sequential implementation is faster for matrices up to order 1000. The reason for this is that the parallel implementation needs time for initialization of CUDA.

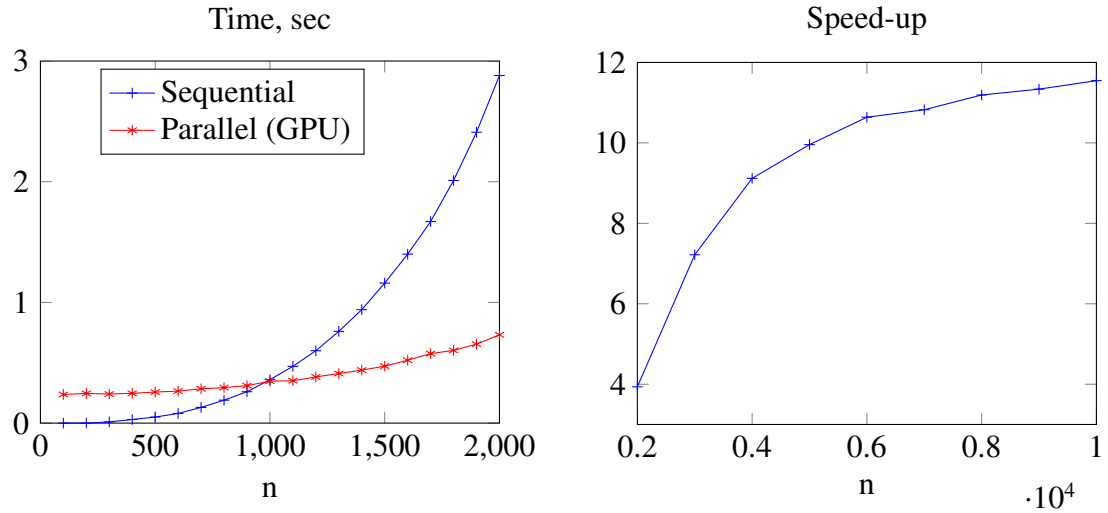


Figure 3.7: Time for sequential and parallel implementations and achieved speed up

Speed is very important for the scientific computations, but before computing fast, one has to compute correctly. The features supported by the GPU are encoded in the compute capability number. Current generations of GPUs support both single and double precision defined in IEEE 754 standard [10].

As explained in the whitepaper [16], because of the difference in architecture, the results on CPU and GPU will not be the same. Floating point operations are not associative. By using more threads we rearrange operations, therefore the same sequence of operations on CPU and GPU may not be performed and we may not got the same numerical result.

The plots in Figure 3.8 are the results of successive computations of the **relative error** $\|A - MDM^T\|/\|A\|$ as a function of the matrix size, where $\|\cdot\|$ is the matrix infinity norm. We show a relative errors for three different random symmetric indefinite matrices for every order n for both sequential and parallel implementations.

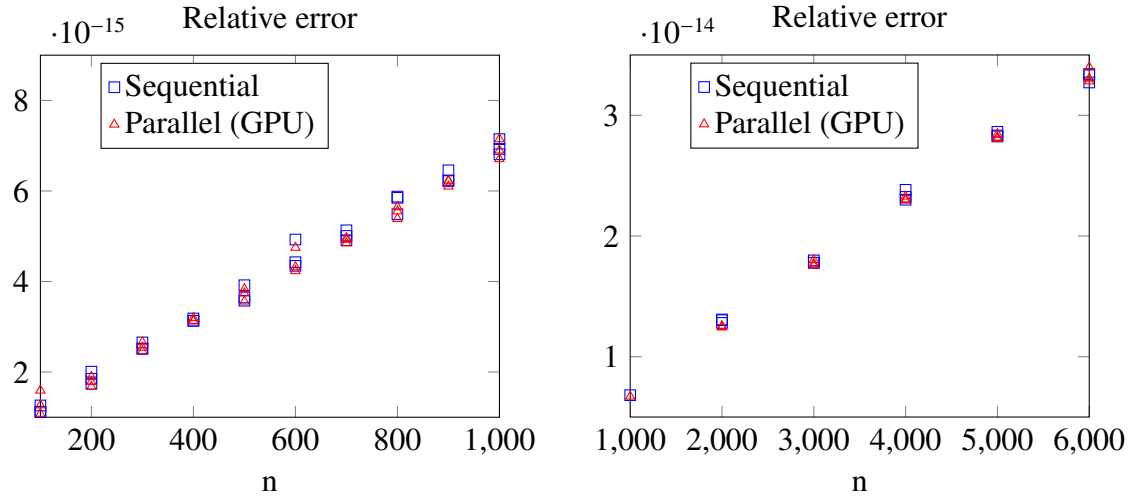


Figure 3.8: Relative error for random matrices of order n. For every n was taken three different matrices.

It is interesting to see the results of the factorizations that used only 1×1 or 2×2 pivots in every step. Implementation distinguishes these two cases and performs different actions, as was explained in the previous section. Since the case of the 1×1 pivot needs less computations, we expect that the algorithm will be faster on positive definite matrices (that is when the algorithm uses 1×1 pivot every time). However, by using the 2×2 pivot, we perform less steps (we use one 2×2 pivot instead of two 1×1 pivots).

As test matrices we used Hadamard and symmetric Clement (or Kac) matrices generated using MATLAB [12]. Hadamard matrix is a symmetric indefinite of 1's and -1's whose columns are orthogonal, and in algorithm it always takes 1×1 pivots.

Clement matrix is a tridiagonal matrix with zeros on its main diagonal and known eigenvalues, which include plus and minus the numbers $n-1, n-3, n-5, \ldots,$ (1 or 0). Because of its structure in algortihm it always takes 2×2 pivots.

In Figures 3.11 and 3.14 we have the factorization data and plots of the element growth $\max_{ij}|A_{ij}^{[n-k]}|/\max_{ij}|A_{ij}|$ on each stage, where A is the original matrix and $A^{[n-k]}$ is the reduced matrix on stage k. Factorization data includes order of the matrix, its rank, the total number of swaps separately for both 1×1 and 2×2 pivots, the total number of pivots, execution time and relative error. Pivots 1×1 and 2×2 on each stage are colored in blue and red, respectively.

Furthermore, it is interesting to see how the matrix is changing *throughout the stages*. We show the pictures of the Hadamard matrix on different stages. Large elements of the matrix are colored in red, and the smaller ones are blue. In Figure 3.15 we have a lower triangle of the Hadamard matrix of order 512 on 1st (original), 101st, 205th, 322nd, 421st

Hadamard matrix Desc.: matrix of 1's and −1's with orthogonal columns	
Order	1024
Rank	1024
1×1 swaps	0
2×2 swaps	0
1×1 pivots	1024
2×2 pivots	0
Time	0.34 sec
Rel. error	0

Figure 3.9: Factorization data

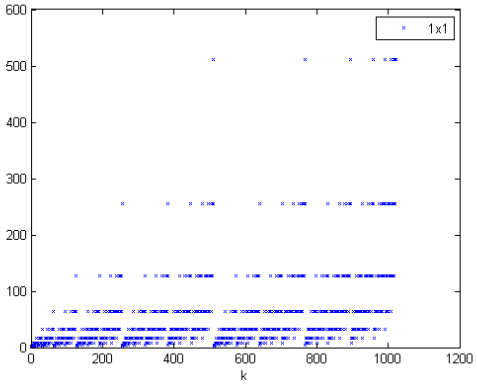


Figure 3.10: Elements growth

Figure 3.11: Hadamard matrix

Clement matrix Desc.: tridiagonal matrix with zeros on its main diagonal	
Order	1024
Rank	1024
1×1 swaps	0
2×2 swaps	930
1×1 pivots	0
2×2 pivots	512
Time	0.29 sec
Rel. error	6.14×10^{-16}

Figure 3.12: Factorization data

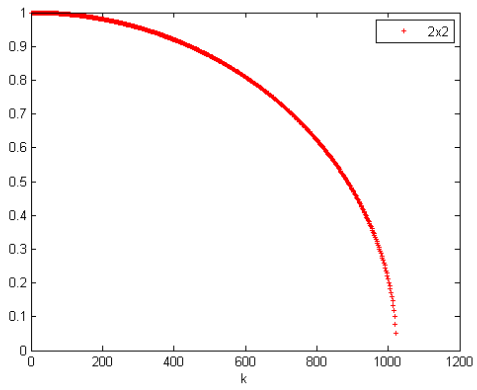
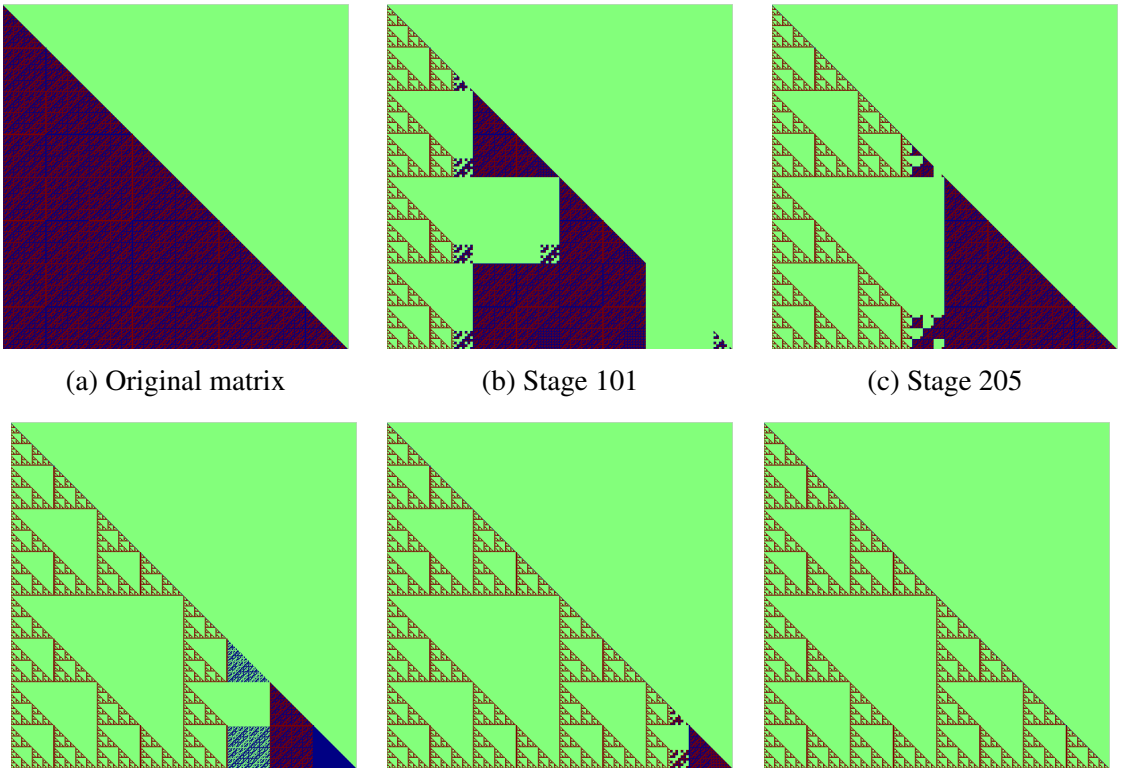


Figure 3.13: Elements growth

Figure 3.14: Clement matrix

and 512th (final) stage of the factorization. We can see the construction of matrices M and D (here D is diagonal because matrix A is positive definite) which overwrite the original matrix in each stage. At the end of algorithm, matrix M uncovers its amazing fractal structure, called *Sierpinski carpet*.

We tested our implementation on a number of matrices drawn from *The university of Florida sparse matrix collection* [6]. We used dense matrices from different applications to demonstrate a wide spectrum of problems where the symmetric indefinite matrices arise.



(a) Original matrix (b) Stage 101 (c) Stage 205

(d) Stage 322 (e) Stage 421 (f) Final matrix

Figure 3.15: Factorization of the Hadamard matrix (lower triangle)

We present the data of three of them in Figures 3.16, 3.18 and 3.20.

The plots 3.17, 3.19 and 3.21 show the elements growth for these matrices, where blue and red points indicate that 1×1 and 2×2 pivot was used in this stage, respectively. The names of matrices we used are the same as in the collection.

Singular matrices

Our code is designed mainly for dense non-singular matrices, but it can also be used for singular matrices. Factorization is regarded as complete if the absolute value of elements of the reduced matrix becomes less than a given tolerance. In that case, our program will return the computed rank of the original matrix. The tolerance depends on the magnitude of elements in the original matrix. More precisely, it is equal to the product of maximum by absolute value element in the original matrix and the machine epsilon. Note that rank can be underestimated or overestimated if we have an inappropriate tolerance.

Alemdar matrix	
Desc.: Finite-element matrix	
Order	6245
Rank	6245
1×1 swaps	3632
2×2 swaps	2581
1×1 pivots	3635
2×2 pivots	1305
Time	9.8 sec
Rel. error	4.689×10^{-14}

Figure 3.16: Factorization data

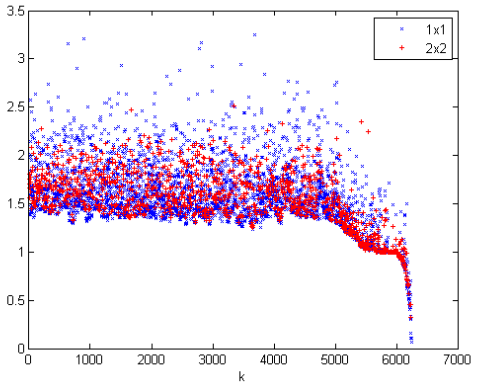


Figure 3.17: Elements growth

human_gene1 matrix	
Desc.: Human gene network	
Order	22283
Rank	22283
1×1 swaps	21970
2×2 swaps	278
1×1 pivots	22003
2×2 pivots	140
Time	7.6 min
Rel. error	4.620×10^{-15}

Figure 3.18: Factorization data

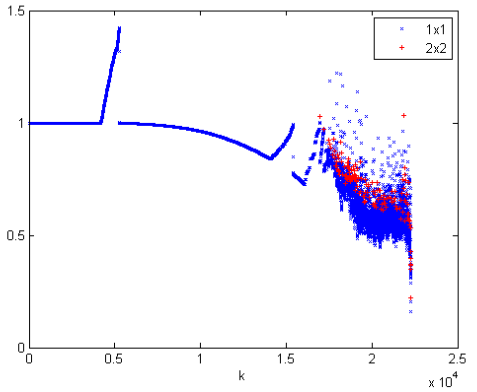


Figure 3.19: Elements growth

astro-ph matrix	
Desc.: Collaboration network	
Order	16706
Rank	15668
1×1 swaps	7003
2×2 swaps	8660
1×1 pivots	7006
2×2 pivots	4331
Time	2.7 min
Rel. error	2.977×10^{-15}

Figure 3.20: Factorization data

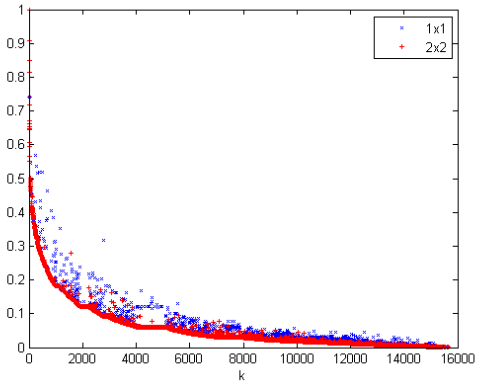


Figure 3.21: Elements growth

The famous Hilbert matrix has a full rank, but it is very ill-conditioned, and it is hard to compute its rank numerically. For Hilbert matrix of order 2000, MATLAB function `rank` gives us the result 25 and our implementation with the tolerance as explained above gives us the result 30. The example of a successful computation of rank is the `alemdar` matrix (see Figure 3.16).

Dependence on the parameter α

The important question that arises when dealing with symmetric indefinite factorization is when to take the 2×2 rather than 1×1 pivot. By complete pivoting we choose the 1×1 pivot when maximum by absolute value diagonal element μ_1 in the reduced matrix is not much smaller then maximum by absolute value off-diagonal element μ_0, i.e., when $\mu_1/\mu_0 > \alpha$ for some $0 < \alpha < 1$. In Chapter 2 we have shown that the parameter α which prevents the rapid growth of elements is equal to

$$\alpha_0 = \frac{1 + \sqrt{17}}{8}.$$

Others values of the parameter α are also possible, but then the respective bounds on elements will be larger or go to infinity. In Table 3.1 we give the relative errors of computed factorizations for three matrices (`astro-ph` and `alemdar` used before, and one random matrix of order 3000) when using different parameters α. Highlighted row corresponds to the parameter equal to the optimal α_0. We denote the more accurate results with the red colour. Note that some of these results are obtained for non-optimal parameters.

α	**astro-ph**	**3000**	**alemdar**
0.01	3.938794e-15	1.766888e-14	4.971599e-14
0.15	**2.757264e-15**	1.766888e-14	4.884935e-14
0.29	3.137581e-15	1.766888e-14	4.599203e-14
0.43	3.282157e-15	1.766888e-14	4.507927e-14
0.57	3.274952e-15	1.824500e-14	4.721938e-14
α_0	2.977507e-15	**1.721470e-14**	4.689740e-14
0.71	2.956007e-15	1.753908e-14	4.127681e-14
0.85	2.946825e-15	2.050335e-14	**4.011067e-14**
0.99	1.703524e-14	7.376952e-14	4.573181e-14

Table 3.1: Relative errors for different parameters α

GJG$^{\mathrm{T}}$ factorization

In the introduction we briefly explained the method for computing eigenpairs for symmetric indefinite matrices that uses the symmetric indefinite factorization. The first part of this method is computing the factorization $PAP^T = GJG^T$ of original matrix A (called GJG$^{\mathrm{T}}$ factorization), where G is a lower block-triangular and J is diagonal matrix with 1's, -1's and 0's on the diagonal. Authors in [13] used a Fortran routine for generation of the needed matrices G and J. Firstly, they computed symmetric indefinite factorization with complete pivoting and after that transformed obtained matrices M and D to the matrices G and J. We compared the results from their Fortran routine (written in a 128-bit extended precision) with results obtained with our CUDA C implementation. Diagonalization of matrix D and scaling of M was implemented in a *separated CUDA C program*. Obtained relative errors for matrices of order n are given in Figure 3.22.

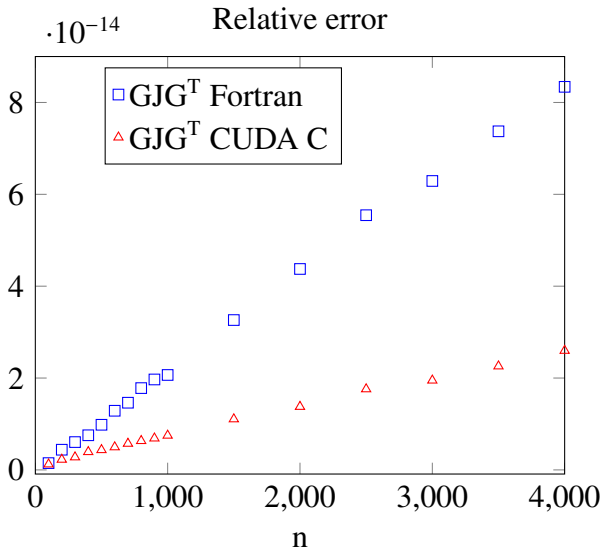


Figure 3.22: Relative errors for Fortran and CUDA C implementations for GJG$^{\mathrm{T}}$ factorization

Chapter 4

Conclusion and future outlook

In this final chapter we will briefly review the goals that we have achieved throughout this thesis. We will summarize and bring together the main results and ideas covered in this work, make final comments which include suggestions for improvement and future work.

The purpose of this thesis was to study and implement the symmetric indefinite factorization for dense matrices on GPU. In Chapter 1 we introduced the factorization and discussed the most popular pivot strategies. The complete pivoting is the most stable of them and it can be efficiently implemented for GPU. Therefore, for our implementation we chose the complete pivoting. In Chapter 2 we have presented an analysis of complete pivoting following the results of Bunch [2] and have showed that it is backward stable and provides bound for the multipliers. The latter is important for several algorithms, such as modified Cholesky factorization. In Chapter 3 we provided the details of our implementation and obtained numerical results. These results have shown that the factorization with complete pivoting can be efficiently implemented on GPU and our implementation gave very accurate results.

In [13] it was pointed out that the implementation of symmetric indefinite factorization is the essential missing part of the complete GPU-based symmetric indefinite Jacobi-type eigensolver. It would be interesting to replace the missing part with our implementation and compare obtained results with the already existing ones.

Implementation of the symmetric indefinite factorization was developed for dense matrices and cannot be used effectively for sparse matrices. Ashcraft, Grimes and Lewis [1] address the *sparse problem* and discuss the constraints imposed by the goal of preserving sparsity. They have shown that sparsity prevents the application of the techniques that were used to solve the dense problem, and have developed different approaches. The next step could also be the parallelization of that suggested algorithms on GPU.

In applications can arise very large matrices that cannot fit within the global memory. For example, the finite element matrix in practical applications often has the dimension of

order of tens or hundreds of thousands. Tesla S2050 provides 3 GB of global memory. Matrices of order 30 thousands or more cannot be computed with our algorithm using this GPU. If we need to compute the factorization for a larger matrix, we have two choices: to find a machine with larger memory or use the so called *out-of-core algorithm* where factors are stored on the disk. The disk-to-memory bandwidth is usually about two orders of magnitude lower than memory-to-processor bandwidth. Therefore, the out-of-core algorithm needs to take care about the disk input/output accesses. Therefore, the possible direction of a similar study is the out-of-core implementation.

Bibliography

[1] C. Ashcraft, R. G. Grimes, and J. G. Lewis, *Accurate symmetric indefinite linear equation solvers*, SIAM Journal on Matrix Analysis and Applications **20** (1999), 513–561.

[2] J. R. Bunch, *Analysis of the diagonal pivoting method*, SIAM Journal on Numerical Analysis **8** (1971), 656–680.

[3] J. R. Bunch and L. Kaufman, *Some stable methods for calculating inertia and solving symmetric linear systems*, Math. Comp. **31** (1977), 163–179.

[4] J. R. Bunch and B. N. Parlett, *Direct methods for solving symmetric indefinite systems of linear equations*, SIAM Journal on Numerical Analysis **8** (1971), 639–655.

[5] S. H. Cheng and N. J. Higham, *A modified Cholesky algorithm based on a symmetric indefinite factorization*, SIAM Journal on Matrix Analysis and Applications **19** (1998), 1097–1110.

[6] T. A. Davis and Y. Hu, *The university of florida sparse matrix collection*, ACM Transactions on Mathematical Software **38** (2011), 1:1–1:25.

[7] V. Hari, S. Singer, and S. Singer, *Full block J-Jacobi method for Hermitian matrices*, Techn. rep., Dept. of Mathematics, Univ. of Zagreb, 2010, submitted for publication.

[8] N. J. Higham, *Stability of the diagonal pivoting method with partial pivoting*, SIAM Journal on Matrix Analysis and Applications **18** (1997), 52–65.

[9] ———, *Accuracy and stability of numerical algorithms*, 2nd ed., Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2002.

[10] *IEEE 754-2008, standard for floating-point arithmetic*, August 2008.

[11] D. B. Kirk and W. W. Hwu, *Programming massively parallel processors: A hands-on approach*, 2nd ed., Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2013.

[12] *MATLAB, The Mathworks Inc., Natick, Massachusetts*, 2010.

[13] V. Novaković and S. Singer, *A GPU-based hyperbolic SVD algorithm*, BIT **51** (2011), 1009–1030.

[14] *NVIDIA CUDA C Programming guide*, June 2011.

[15] K. Veselić and V. Hari, *A note on one-sided Jacobi algorithm*, Numer. Math. **56** (1989), 627–633.

[16] N. Whitehead and A. Fit-Florea, *Precision & performance: Floating point and IEEE 754 compliance for NVIDIA GPUs*, Nvidia technical white paper, NVidia, 2011.

Sažetak

Simetrična indefinitna faktorizacija je ekvivalent faktorizaciji Choleskog za matrice koje su simetrične, ali nisu pozitivno definitne. U takvoj faktorizaciji uvijek se primjenjuje pivotiranje i pivoti su 1×1 ili 2×2 matrice. Algoritam se može iskoristiti za rješavanje indefinitnih linearnih sustava i kao prva faza algoritma za traženje svojstvenih vrijednosti simetrične indefinitne matrice. Cilj ovog diplomskog rada je paralelizacija algoritma za simetričnu indefinitnu faktorizaciju s potpunim pivotiranjem na grafičkoj kartici (GPU). Numerička testiranja pokazuju ubrzanje i dobru točnost u usporedbi sa sekvencijalnom verzijom.

Summary

The symmetric indefinite factorization is an equivalent to the Cholesky factorization for matrices that are symmetric, but not positive definite. This factorization uses some kind of pivoting, and takes 1×1 or 2×2 matrices as pivot elements. This algorithm can be used for solving symmetric indefinite linear systems and as the first stage of solving symmetric indefinite eigenvalue problem. The goal of this thesis is a parallelization of the algorithm for symmetric indefinite factorization with complete pivoting on a graphics processing unit (GPU). Numerical testing shows improvement in speed and good accuracy compared to the sequential version.

Biography

I was born on the 18th of December 1991 in Orehovo-Zuevo (Russian Federation). After finishing school, I enrolled at the Faculty of Applied Mathematics and Control Processes of Saint-Petersburg State University. After moving to Croatia, I continued my studies at the University of Zagreb's Faculty of Science, at the Department of Mathematics. After successfully completing my undergraduate degree programme in Mathematics, I enrolled at the University's graduate programme in Computer Science and Mathematics.