FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Task-based implementation of Cholesky decomposition
Student:	Vít Břichňáč
Supervisor:	Ing. Jakub Šístek, Ph.D.
Study program:	Informatics
Branch / specialization:	Knowledge Engineering
Department:	Department of Applied Mathematics
Validity:	until the end of summer semester 2023/2024

Instructions

Cholesky matrix decomposition, that is, factorization of a symmetric positive definite matrix to A = LL^T is one of the basic algorithms of numerical linear algebra and a crucial step in solving systems of linear equations with such matrices. The block version of this algorithm has a variable amount of parallelism. A suitable tool expressing this algorithm is a graph of tasks with data dependencies, the so-called Directed Acyclic Graph (DAG), in which other tools (called runtime systems) discover the parallelism automatically and optimally employ cores of a multicore processor.

The aim of the work is to create own implementation of the algorithm of Cholesky decomposition using a task-based programming framework.

1) The student will learn basic algorithms of numerical linear algebra and for operations with matrices [1].
2) He will get familiar with tools for task-based programming on multicore processors with shared memory using OpenMP [2] and StarPU [3].
3) The student will develop an implementation of the block Cholesky decomposition in C programming language using these tools.
4) He will evaluate the efficiency of the developed implementation by measuring time of the computation or performance on a multicore processor and he will compare it with available numerical libraries, such as the Intel Math Kernel Library (MKL).

References:

Electronically approved by Ing. Magda Friedjungová, Ph.D. on 13 October 2022 in Prague.

[1] Trefethen, L. N., Bau, D. III. Numerical Linear Algebra, SIAM, 1997.
[2] OpenMP, http://www.openmp.org
[3] StarPU, http://starpu.gforge.inria.fr

Bachelor's thesis

TASK-BASED IMPLEMENTATION OF CHOLESKY DECOMPOSITION

Vít Břichňáč

Faculty of Information Technology
Department of Applied Mathematics
Supervisor: Ing. Jakub Šístek, Ph.D.
May 11, 2023

Citation of this thesis: Břichňáč Vít. *Task-based implementation of Cholesky decomposition.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

Contents

An immense amount of my gratitude goes towards the supervisor of this thesis, Ing. Jakub Šístek, Ph.D., for his cooperation, his will to help with every issue that has arisen during the development process of this thesis, and last but not least for his admirable amount of patience. I would also like to thank my family and my friends for supporting me throughout my studies at FIT CTU.

Declaration

In Prague on May 11, 2023 Vít Břichňáč

Abstract

The aim of this thesis is to present the blocked algorithm for finding a Cholesky decomposition of a symmetric positive definite matrix and to provide an implementation of the algorithm using the StarPU and OpenMP task-based runtime systems. The implementation is tested using test routines from the LAPACK library and its performance is compared to several available libraries for numerical linear algebra on three different compute nodes. On all three of them, the implementation delivers a competitive performance, especially for larger matrices. In the theoretical part of the thesis, custom proofs of correctness are formulated for both the blocked and the unblocked algorithms for Cholesky decomposition. The rest of the thesis explores matrix storage schemes, the BLAS interface and basic concepts of task-based programming.

Keywords Cholesky decomposition, runtime systems, task-based programming, numerical linear algebra, StarPU, OpenMP

Abstrakt

Cílem této práce je představit blokový algoritmus pro nalezení Choleského rozkladu symetrické pozitivně definitní matice a implementovat tento algoritmus pomocí task-based runtime systémů StarPU a OpenMP. Tato implementace je otestována za pomoci testovacích rutin z knihovny LAPACK a její výkon je porovnán s několika volně dostupnými knihovnami pro numerickou lineární algebru na třech různých výpočetních uzlech. Výkon implementace na všech třech uzlech vykazuje konkurenceschopnost, především pak u větších matic. V teoretické části práce jsou zformulovány vlastní důkazy korektnosti jak blokového, tak neblokového algoritmu pro Choleského rozklad. Ve zbytku práce jsou prezentována schémata pro ukládání matic, rozhraní BLAS a základní koncepty task-based programování.

Klíčová slova Choleského rozklad, runtime systémy, task-based programování, numerická lineární algebra, StarPU, OpenMP

Introduction

One of the most thoroughly examined problems in mathematics is solving systems of linear equations. Systems of linear equations often arise as a discretization of systems of differential equations that describe a certain model in physics, economics, and many more scientific areas.

As the size of the systems of equations describing these models grows steadily, so does the demand for high-efficiency parallelized algorithms for solving those systems. The subject of research in this thesis is Cholesky decomposition, which is used to solve linear equation systems with symmetric positive definite matrices.

When computing the decomposition in parallel on several CPU cores, the matrix is split into blocks, and elementary matrix operations are performed on those blocks. Some of the operations, however, require other blocks to be already processed. To simplify the implementation of these algorithms, the routine is split into several tasks that are performed on the individual blocks, and the dependencies between these tasks are described to a task-based runtime system, which then takes care of allocating computing power to the individual tasks and ensuring that no task is run before all of its dependencies are satisfied.

The main goal of this thesis is to implement a blocked algorithm for the Cholesky decomposition using the StarPU (and possibly OpenMP) runtime systems, then to measure the performance of this implementation and compare it to several well-known implementations on different CPU platforms (with various instruction set architectures, if possible).

As there are many available highly optimized implementations of the blocked algorithm for Cholesky decomposition (many of them being developed by major hardware vendors), the author is not expecting to develop the new fastest algorithm implementation that could be used in production runs. Instead, the main reason for selecting this topic was an interest in seeing a performance comparison between the available optimized implementations and a relatively unoptimized custom implementation that uses a modern task-based runtime system with an optimized scheduler.

The first chapter presents elementary mathematical concepts regarding matrices, vectors, symmetric matrices, and their definiteness. The second chapter discusses several approaches for storing matrices in computer memory. In the third chapter, we describe three of the elementary operations used in the blocked algorithm and the interface standard associated with them (BLAS). The content of the fourth chapter comprises a description of the Cholesky decomposition and the blocked and unblocked algorithms used for its computation. In the fifth chapter, we examine existing implementations of the decomposition in software libraries for numerical linear algebra. The sixth chapter describes elementary concepts of task-based runtime systems and presents two widely used examples of such systems, OpenMP and StarPU.

The last two chapters describe the implementation, subsequent testing, and performance evaluation.

Chapter 1

Properties of symmetric matrices

1.1 Matrices and vectors

Hefferon [1] defines an $m \times n$ **matrix** as a rectangular array of numbers with m rows and n columns. Matrices are typically named with capital letters.

Each number in a matrix is called an **entry**. The entry in the ith row and the jth column of a matrix A is denoted A_{ij}.

The set of all $m \times n$ matrices whose entries are real numbers is labeled $\mathbb{R}^{m,n}$.

The **transpose** of a matrix A is a matrix obtained by interchanging the corresponding rows and columns of A [1]:

▶ **Definition 1.1.** *A transpose of a matrix $A \in \mathbb{R}^{m,n}$ is a matrix $A^T \in \mathbb{R}^{n,m}$ such that $A_{ij} = A^T_{ji}$ for all $i \in \{1, \dots, m\}$ and $j \in \{1, \dots, n\}$.*

A sum of two matrices of the same size and the scalar multiple of a matrix are both defined element-wise [1]:

▶ **Definition 1.2.** *A **sum** of two $m \times n$ matrices A and B is a matrix $A + B \in \mathbb{R}^{m,n}$ such that*

$$(A + B)_{ij} = A_{ij} + B_{ij}$$

for each $i \in \{1, \dots, m\}$ and $j \in \{1, \dots, n\}$.

▶ **Definition 1.3.** *Let $\alpha \in \mathbb{R}$. A **scalar multiple** of an $m \times n$ matrix A is a matrix $\alpha A \in \mathbb{R}^{m,n}$ such that*

$$(\alpha A)_{ij} = \alpha A_{ij}$$

for each $i \in \{1, \dots, m\}$ and $j \in \{1, \dots, n\}$.

A matrix product is defined as follows:

▶ **Definition 1.4** ([1], Definition 2.3)**.** *Let A be an $m \times r$ matrix and B an $r \times n$ matrix. A **product** of A and B is a matrix $AB \in \mathbb{R}^{m,n}$ such that*

$$(AB)_{ij} = \sum_{k=1}^{r} A_{ik} B_{kj}$$

for each $i \in \{1, \dots, m\}$ and $j \in \{1, \dots, n\}$.

The product of matrices A and B can also be denoted by $A \cdot B$.

▶ **Theorem 1.5.** *Let $A \in \mathbb{R}^{m,n}, B, D \in \mathbb{R}^{n,o}, C \in \mathbb{R}^{o,p}$ and $\alpha \in \mathbb{R}$. Then:*

1. $(AB)C = A(BC)$
2. $A(\alpha B) = (\alpha A)B = \alpha(AB)$
3. $A(B+D) = AB + AD$
4. $(B+D)C = BC + DC$
5. $A^T B^T = (BA)^T$

Proof.

1. $$\big((AB)C\big)_{ij} = \sum_{l=1}^{o} (AB)_{il} C_{lj} = \sum_{l=1}^{o}\sum_{k=1}^{n} (A_{ik}B_{kl})C_{lj} = \sum_{k=1}^{n}\sum_{l=1}^{o} A_{ik}(B_{kl}C_{lj}) = \\ = \sum_{k=1}^{n} A_{ik}(BC)_{kj} = \big(A(BC)\big)_{ij}$$

2. $$\big(A(\alpha B)\big)_{ij} = \sum_{k=1}^{n} A_{ik}(\alpha B)_{kj} = \sum_{k=1}^{n} \alpha A_{ik}B_{kj} = \sum_{k=1}^{n} (\alpha A)_{ik}B_{kj} = \big((\alpha A)B\big)_{ij} = \\ = \alpha\sum_{k=1}^{n} A_{ik}B_{kj} = \big(\alpha(AB)\big)_{ij}$$

3. $$\big(A(B+D)\big)_{ij} = \sum_{k=1}^{n} A_{ik}(B+D)_{kj} = \sum_{k=1}^{n} A_{ik}(B_{kj}+D_{kj}) = \sum_{k=1}^{n} A_{ik}B_{kj}+A_{ik}D_{kj} = (AB+AD)_{ij}$$

4. $$\big((B+D)C\big)_{ij} = \sum_{k=1}^{n} (B+D)_{ik}C_{kj} = \sum_{k=1}^{n} (B_{ik}+D_{ik})C_{kj} = \sum_{k=1}^{n} B_{ik}C_{kj}+D_{ik}C_{kj} = (BC+DC)_{ij}$$

5. $$(A^TB^T)_{ij} = \sum_{k=1}^{n} (A^T)_{ik}(B^T)_{kj} = \sum_{k=1}^{n} A_{ki}B_{jk} = \sum_{k=1}^{n} B_{jk}A_{ki} = (BA)_{ji} = \big((BA)^T\big)_{ij}$$

◀

1.1.1 Square matrices

An $n \times n$ matrix is called a **square matrix** of size n.

A diagonal matrix is a square matrix whose non-diagonal entries are zero [2], or more precisely:

▶ **Definition 1.6.** *A square matrix $A \in \mathbb{R}^{n,n}$ is called **diagonal** if $A_{ij} = 0$ for all $i, j \in \{1, \dots, n\}, i \neq j$.*

By relaxing the restriction so that only the entries above or below the main diagonal need to be zero, we obtain a triangular matrix:

▶ **Definition 1.7.** *A square matrix $A \in \mathbb{R}^{n,n}$ is:*

- ***lower triangular**, if $A_{ij} = 0$ for all $i, j \in \{1, \dots, n\}, i < j$*
- ***upper triangular**, if $A_{ij} = 0$ for all $i, j \in \{1, \dots, n\}, i > j$*

A special case of a diagonal matrix is an identity matrix:

▶ **Definition 1.8** ([2]). *An **identity matrix** of size n is a square matrix $I \in \mathbb{R}^{n,n}$, where*

$$\forall i, j \in \{1, \dots, n\}: \quad I_{ij} = \begin{cases} 1, & \text{if } i = j, \\ 0, & \text{if } i \neq j. \end{cases}$$

For the rest of this thesis, the identity matrix will always be labeled I. The identity matrix acts as the identity element with respect to matrix multiplication [1]:

▶ **Theorem 1.9.** *Let $A \in \mathbb{R}^{m,n}, B \in \mathbb{R}^{n,m}$ and let I be the $n \times n$ identity matrix. Then:*

$$AI = A \quad \text{and} \quad IB = B.$$

Proof.

$$(AI)_{ij} = \sum_{k=1}^{n} A_{ik} I_{kj} = \sum_{\substack{k\in\{1,\dots,n\}\\ k\neq j}} A_{ik} I_{kj} + A_{ij} I_{jj} = \sum_{\substack{k\in\{1,\dots,n\}\\ k\neq j}} A_{ik} \cdot 0 + A_{ij} \cdot 1 = A_{ij}$$

$$(IB)_{ij} = \sum_{k=1}^{n} I_{ik} B_{kj} = \sum_{\substack{k\in\{1,\dots,n\}\\ k\neq i}} I_{ik} B_{kj} + I_{ii} B_{ij} = \sum_{\substack{k\in\{1,\dots,n\}\\ k\neq i}} 0 \cdot B_{kj} + 1 \cdot B_{ij} = B_{ij}$$

◀

An important property of square matrices that has many different (but equivalent) definitions across different linear algebra texts or textbooks, is nonsingularity:

▶ **Definition 1.10** ([2]). *A square matrix $A \in \mathbb{R}^{n,n}$ is called **nonsingular** (or **invertible**) if there exists a matrix $C \in \mathbb{R}^{n,n}$ such that*

$$AC = I \quad \text{and} \quad CA = I.$$

*C is then called an **inverse** of A. A matrix that is not invertible is called **singular**.*

▶ **Theorem 1.11.** *The inverse of a nonsingular matrix $A \in \mathbb{R}^{n,n}$ is determined uniquely, that is, if there are two $n \times n$ matrices B and C such that $AB = BA = AC = CA = I$, then $B = C$.*

Proof. Consider $B, C \in \mathbb{R}^{n,n}$ such that $AB = BA = AC = CA = I$. Then:

$$\begin{aligned} AB &= AC \\ B(AB) &= B(AC) \\ (BA)B &= (BA)C \\ IB &= IC \\ B &= C \end{aligned}$$

◀

The (unique) inverse of a matrix A is denoted by A^{-1} [2].

▶ **Theorem 1.12.** *$A \in \mathbb{R}^{n,n}$ is a nonsingular matrix if and only if A^T is nonsingular. Moreover, if A is nonsingular, then $(A^T)^{-1} = (A^{-1})^T$.*

Proof. If A is nonsingular, we can show that $(A^{-1})^T$ is the inverse of A^T using Theorem 1.5:

$$\begin{aligned} A^T(A^{-1})^T &= (A^{-1}A)^T = I^T = I \\ (A^{-1})^T A^T &= (AA^{-1})^T = I^T = I \end{aligned}$$

Furthermore, if A^T is nonsingular, then $A^{-1} = \left((A^T)^{-1}\right)^T$ since $A = (A^T)^T$. ◀

▶ **Theorem 1.13.** *Let $A, B \in \mathbb{R}^{n,n}$ be two nonsingular matrices. Then AB is nonsingular with the inverse $(AB)^{-1} = B^{-1}A^{-1}$.*

Proof.

$$(B^{-1}A^{-1})AB = B^{-1}(A^{-1}A)B = B^{-1}B = I$$
$$AB(B^{-1}A^{-1}) = A(BB^{-1})A^{-1} = AA^{-1} = I$$

◀

A **determinant** of a square matrix A (labeled $\det A$) is a number that fully determines the singularity of A – more precisely, $\det A \neq 0$ if and only if A is nonsingular. There are multiple equivalent definitions of the determinant [1, 2], but for the purposes of this thesis, we will present a determinant formula for lower triangular matrices only:

▶ **Theorem 1.14** ([2], Theorem 3.1.2)**.** *Let $A \in \mathbb{R}^{n,n}$ be a lower triangular matrix. Then the determinant of A is equal to*

$$\det A = \prod_{i=1}^{n} A_{ii}.$$

▶ **Theorem 1.15** ([2], Theorem 3.2.4)**.** *A lower triangular matrix $A \in \mathbb{R}^{n,n}$ is invertible if and only if* $\det A \neq 0$.

1.1.2 Vectors

Starting from this subsection, we will consider a real number x and a 1×1 matrix whose only entry is x to be equal, or, in other words, $x = \big(x\big)$ for all $x \in \mathbb{R}$. Note that this convention does not cause any conflict with basic matrix operations as defined above, since $\big(x\big) + \big(y\big) = \big(x + y\big)$ and $\big(x\big) \cdot \big(y\big) = x \cdot \big(y\big) = \big(x \cdot y\big)$ for all $x, y \in \mathbb{R}$.

▶ **Definition 1.16** ([1], Definition I.2.8)**.** *An $n \times 1$ matrix (that is, a matrix with a single column) is called a **vector**.*

The entries $x_{11}, x_{21}, \ldots, x_{n1}$ of a vector x can be also represented by $x_1, x_2, \ldots, x_n$.

▶ **Definition 1.17** ([1], Definition I.2.8)**.** *A **zero vector** is a vector whose entries are all equal to zero.*

Zero vectors will be further labeled θ.

▶ **Definition 1.18** ([1], Definition II.2.1)**.** *The **length** (or more formally, the **Euclidean norm**) of a vector $x \in \mathbb{R}^{n,1}$ is defined as*

$$\|x\| = \sqrt{\sum_{i=1}^{n} x_i^2}.$$

It is important to note (and easy to see) that a length of every vector is non-negative and that a vector has a length of zero if and only if it is a zero vector.

▶ **Lemma 1.19.** *Let $x \in \mathbb{R}^{n,1}$ be a vector. Then*

$$\|x\|^2 = x^T x.$$

Proof.

$$\|x\|^2 = \left(\sqrt{\sum_{i=1}^{n} x_i^2}\right)^2 = \sum_{i=1}^{n} x_i^2 = \sum_{i=1}^{n} (x^T)_{1i} x_{i1} = x^T x$$

◀

The following theorem is provided without a proof, as the proof usually consists of proving a larger chain of implications.

▶ **Theorem 1.20** ([2], Theorem 2.8)**.** *Consider a square matrix $A \in \mathbb{R}^{n,n}$. Then*

$$A \textit{ is nonsingular} \quad \Leftrightarrow \quad \forall x \in \mathbb{R}^{n,1}, x \neq \theta : \|Ax\| > 0.$$

1.2 Symmetric matrices

A symmetric matrix is defined by Trefethen and Bau [3] as a square matrix that is equal to its transpose. An equivalent definition without using matrix transposition can be formed:

▶ **Definition 1.21** ([3])**.** *A square matrix $A \in \mathbb{R}^{n,n}$ is* ***symmetric*** *if and only if $A_{ij} = A_{ji}$ for all $i, j \in \{1, \dots, n\}$.*

▶ **Lemma 1.22.** *Symmetric matrices have the following properties:*

1. *For a general matrix $X \in \mathbb{R}^{m,n}$, X^TX and XX^T are symmetric*
2. *A sum of two symmetric matrices is symmetric*
3. *A scalar multiple of a symmetric matrix is symmetric*
4. *Every diagonal matrix is symmetric*

Proof. For a general matrix $X \in \mathbb{R}^{m,n}$, symmetric matrices $A, B \in \mathbb{R}^{n,n}$ and a diagonal matrix $D \in \mathbb{R}^{n,n}$:

1. $(X^TX)_{ij} = \sum_{k=1}^{m} (X^T)_{ik} \cdot X_{kj} = \sum_{k=1}^{m} X_{ki} \cdot (X^T)_{jk} = \sum_{k=1}^{m} (X^T)_{jk} \cdot X_{ki} = (X^TX)_{ji}$
   $(XX^T)_{ij} = \sum_{k=1}^{m} X_{ik} \cdot (X^T)_{kj} = \sum_{k=1}^{m} (X^T)_{ki} \cdot X_{jk} = \sum_{k=1}^{m} X_{jk} \cdot (X^T)_{ki} = (XX^T)_{ji}$
2. $(A + B)_{ij} = A_{ij} + B_{ij} = A_{ji} + B_{ji} = (A + B)_{ji}$
3. $(kA)_{ij} = k \cdot A_{ij} = k \cdot A_{ji} = (kA)_{ji}$
4. By definition, $D_{ij} = D_{ji} = 0$ for all $i \neq j$

◀

1.3 Matrix definiteness

▶ **Definition 1.23** ([3])**.** *A symmetric matrix A is said to be* ***positive definite*** *if, for every non-zero vector $x \in \mathbb{R}^{n,1}$, it holds that*

$$x^TAx > 0.$$

The term x^TAx for a square matrix $A \in^{n,n}$ is called a **quadratic form** determined by matrix A [2].

▶ **Lemma 1.24.** *Let $A \in \mathbb{R}^{n,n}$ be a nonsingular matrix. Then A^TA and AA^T are symmetric positive definite matrices.*

Proof. We already know that A^TA and AA^T are symmetric from Lemma 1.22. Following Theorems 1.12 and 1.20, we obtain for all non-zero vectors $x \in \mathbb{R}^{n,1}$:

$$x^T(A^TA)x = (x^TA^T)(Ax) = (Ax)^TAx = \|Ax\|^2 > 0$$
$$x^T(AA^T)x = (x^TA)(A^Tx) = (A^Tx)^TAx = \|A^Tx\|^2 > 0$$

◀

A fundamental property of symmetric positive definite matrices that we will use later is the positive definiteness of their principal submatrices [3]. First, we will define what a principal submatrix is.

▶ **Definition 1.25.** *Let $A \in \mathbb{R}^{n,n}$ and $S \subsetneq \{1,\dots,n\}$. A submatrix of A obtained by removing rows with indices S and columns with indices S from A is called a **principal submatrix** of A. Furthermore, if $S = \{i, i+1, \dots, n\}$ for some $i \in \{1,\dots,n\}$, we call the submatrix a **leading principal submatrix**.*

▶ **Theorem 1.26.** *Let $A \in \mathbb{R}^{n,n}$ be a symmetric positive definite matrix. Then its principal submatrix A' obtained by removing the rows and columns with indices $S \subsetneq \{1,\dots,n\}$ is also symmetric positive definite.*

Proof. Symmetry of A' is trivial. Let $R_k = \{1,\dots,n\} \setminus S$ and let R_k be the kth smallest element from R. For every $x' \in \mathbb{R}^{|R|,1}, x' \neq \theta$, we can define $x \in \mathbb{R}^{n,1}$ entry-wise:

$$x_i = \begin{cases} 0, & \text{if } i \in S \\ x'_k, & \text{if } i = R_k \text{ for some } k \in \{1,\dots,|R|\} \end{cases}$$

Then:

$$(x')^TA'x' = \sum_{k=1}^{|R|} x'_k A'_{kk} x'_k = \sum_{k\in S} 0 \cdot A_{ii} \cdot 0 + \sum_{k \in R} x_k A_{kk} x_k = \sum_{k=1}^{n} x_k A_{kk} x_k = x^TAx > 0$$

The last inequality follows from the positive definiteness of A. ◀

1.3.1 Generating symmetric positive definite matrices

We can obtain a symmetric positive definite matrix A from a given nonsingular matrix R using the following formula (see Lemma 1.24):

$$A = R^TR \tag{1.1}$$

However, a randomly generated square matrix is not guaranteed to be invertible. Furthermore, even in the case of R being nonsingular, problems with numerical stability can arise for matrices R that are "nearly singular" (i. e., they have a very large condition number).

To prevent R from being singular or nearly singular, we can add a positive scalar multiple of an identity matrix to the right-hand side of eq. (1.1):

$$A = R^TR + \lambda I \tag{1.2}$$

where I is an identity matrix of size n and $\lambda > 0$. Lemma 1.22 can be used to show that the matrix A obtained from eq. (1.2) is symmetric. The quadratic form determined by matrix A is equal to:

$$x^TAx = x^T(R^TR + \lambda I)x = x^TR^TRx + x^T\lambda Ix = (Rx)^TRx + \lambda x^Tx = \|Rx\|^2 + \lambda\|x\|^2 \tag{1.3}$$

which is positive for every non-zero vector $x \in \mathbb{R}^n$ due to both λ and $\|x\|$ being positive.

Thus, every square matrix A generated using eq. (1.2) is symmetric positive definite.

In summary, we have obtained the following algorithm for generating symmetric positive definite matrices:

```
  Data: $\lambda > 0$
  Result: a symmetric positive definite matrix $A$
1 Generate a random matrix $R \in \mathbb{R}^{n,n}$
2 $A \leftarrow R^T R$
  // this can be done using the GEMM procedure described in Section 3.2
3 for $i \leftarrow 1; i \leq n; i \leftarrow i + 1$ do
4     $A_{ii} \leftarrow A_{ii} + \lambda$
5 end
6 return A
```

Algorithm 1: Generation of symmetric positive definite matrices

1.4 Block matrices

In this thesis, we will often work with **block matrices** (or **partitioned matrices**), that is, matrices obtained by joining smaller matrices (called blocks) in a particular way.

▶ **Definition 1.27** (reformulation of Definition 2.10 in [4]).
Let $A_{11}, \dots, A_{1n}, A_{21}, \dots, A_{2n}, \dots, A_{m1}, \dots, A_{mn}$ be matrices where:

- *for all $i \in \{1, \dots, m\}$, matrices A_{i1} to A_{in} have m_i rows*
- *for all $j \in \{1, \dots, n\}$, matrices A_{1j} to A_{mj} have n_j columns*

We define the block matrix $A = \begin{pmatrix} A_{11} & \dots & A_{1n} \\ \vdots & \ddots & \vdots \\ A_{mn} & \dots & A_{mn} \end{pmatrix}$ such that its x, yth entry is equal to the u, vth entry of A_{ij}, where $x = u + \sum_{k=1}^{i-1} m_k$ and $y = v + \sum_{k=1}^{j-1} n_k$.

*The matrices A_{11} to A_{mn} are called the **blocks** of A.*

In other words, we obtain the matrix A by “joining” the entries of the blocks where:

- The entries of A_{ij} are directly above the entries of $A_{(i+1)j}$ for all $i \in \{1, \dots, m-1\}$ and $j \in \{1, \dots, n\}$
- The entries of A_{ij} are directly to the left of the entries of $A_{i(j+1)}$ for all $i \in \{1, \dots, m\}$ and $j \in \{1, \dots, n-1\}$

We consider a block matrix comprised of a single block to be equivalent to that block, i.e., $\begin{pmatrix} B \end{pmatrix} = B$ for all matrices $B \in \mathbb{R}^{m,n}$.

Multiplication of block matrices works in the same way as standard entry-wise matrix multiplication if we consider the blocks to be the entries of those matrices and if the pairs of blocks being multiplied together are compatible for multiplication (that is, every left-hand side block has the same number of columns as the number of rows of the corresponding right-hand side block).

▶ **Theorem 1.28** (reformulation of Theorem 2.3.4 in [4]).
Let $A = \begin{pmatrix} A_{11} & \dots & A_{1p} \\ \vdots & \ddots & \vdots \\ A_{m1} & \dots & A_{mp} \end{pmatrix}$ *and* $B = \begin{pmatrix} B_{11} & \dots & B_{1n} \\ \vdots & \ddots & \vdots \\ B_{p1} & \dots & B_{pn} \end{pmatrix}$ *be two block matrices for some* $m, p, n \in \mathbb{N}$ *where the number of columns of* A_{ik} *is equal to the number of rows of* B_{kj} *for all* $i \in \{1, \dots, m\}$, $j \in \{1, \dots, n\}$ *and* $k \in \{1, \dots, p\}$. *Then*

$$AB = \begin{pmatrix} \sum_{k=1}^{p} A_{1k}B_{k1} & \dots & \sum_{k=1}^{p} A_{1k}B_{kn} \\ \vdots & \ddots & \vdots \\ \sum_{k=1}^{p} A_{mk}B_{k1} & \dots & \sum_{k=1}^{p} A_{mk}B_{kn} \end{pmatrix}.$$

Transposition of blocked matrices is performed by reordering the transposes of the blocks in the same way as we reorder the entries when transposing matrices entry-wise:

▶ **Theorem 1.29.** *Let* $A = \begin{pmatrix} A_{11} & \dots & A_{1n} \\ \vdots & \ddots & \vdots \\ A_{m1} & \dots & A_{mn} \end{pmatrix}$ *be a block matrix.*

Then $A^T = \begin{pmatrix} A_{11}^T & \dots & A_{m1}^T \\ \vdots & \ddots & \vdots \\ A_{1n}^T & \dots & A_{mn}^T \end{pmatrix}$.

Proof. By Definitions 1.1 and 1.27, the following entries are equal:

- The y, xth entry of A^T
- The x, yth entry of A
- The u, vth entry of A_{ij}, where $x = u + \sum_{k=1}^{i-1} m_k$ and $y = v + \sum_{k=1}^{j-1} n_k$
- The v, uth entry of A_{ij}^T, where $x = u + \sum_{k=1}^{i-1} m_k$ and $y = v + \sum_{k=1}^{j-1} n_k$

Considering that all of the matrices A_{i1}^T to A_{in}^T have n_i columns and the matrices A_{1j}^T to A_{mj}^T have m_j rows, all of the entries stated above are also equal to the y, xth entry of $\begin{pmatrix} A_{11}^T & \dots & A_{m1}^T \\ \vdots & \ddots & \vdots \\ A_{1n}^T & \dots & A_{mn}^T \end{pmatrix}$ and we thus have $A^T = \begin{pmatrix} A_{11}^T & \dots & A_{m1}^T \\ \vdots & \ddots & \vdots \\ A_{1n}^T & \dots & A_{mn}^T \end{pmatrix}$. ◀

Chapter 2

Matrix representation in computer memory

There are many ways in which programs can represent a matrix in memory. Arguably the most natural way to store matrices is the **row major full storage** layout [5], where all entries are stored as a one-dimensional array of numbers in a contiguous area in memory, ordered primarily by their row index and secondarily by their column index. As this storage scheme can be used for all matrices, the reader may ask themselves why other storage schemes are needed. The main reason is that some matrices are known to have special properties which can be taken advantage of in order to:

- Use less memory space (for example, the packed storage scheme for upper triangular matrices only stores entries on/above the main diagonal [6])
- Improve the efficiency of matrix algorithms (for example, the compressed diagonal storage makes matrix-vector products more efficient for some matrices [7])

Unless specified otherwise, row and column entries will be addressed starting from 1 for the rest of this chapter. Also, storage schemes will be sometimes referred to as storage formats.

2.1 Sparse matrix representation

The content of this section follows the structure of [7].

Sparse matrices are matrices with a sufficiently large amount of zero entries, for which storing all entries proves inefficient. Many matrices that arise as approximations of systems of differential equations are sparse, and sparse matrices can be found in various other areas of mathematics or other scientific disciplines. Some sparse storage schemes (e.g., the compressed row/column storage scheme) are designed to be used for any sparse matrices without particular assumptions about the properties of those matrices, while others (for example the block compressed storage scheme) are designed for specific matrix subclasses with certain properties.

The **compressed row storage** (CRS) **scheme** stores the value and the column index of every non-zero entry. There is an array named `values` storing the values and a separate array for the column indices named `column_indices`, both ordered first by the row indices and then by the column indices of the saved entries. A pair of values on the same position in both of those arrays always corresponds to the same matrix entry (that is, if `values`$(k) = a_{ij}$, then `column_indices`$(k) = j$). However, using only those two arrays would make the stored matrices ambiguous. For instance, see Figure 2.1, where both of the 2×2 matrices would have an identical representation.

Figure 2.1 Ambiguity of representation using only the `values` and `column_indices` arrays.

$$\begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}$$

`values`	1
`column_indices`	2

$$\begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$$

`values`	1
`column_indices`	2

Due to this ambiguity, we introduce a third array that stores the index in the `values` and `column_indices` arrays of the first non-zero entry in every row. We name it `row_ptrs`. If there are no non-zero entries in the ith row, the `row_ptrs`(i) value is determined as follows:

- If $i = 1$, then `row_ptrs`$(i) = 0$
- If $i > 1$, then `row_ptrs`$(i) =$ `row_ptrs`$(i-1)$

We denote the number of non-zero entries in the matrix by nnz. Sometimes, $(nnz + 1)$ is appended to the end of `row_ptrs` to make matrix traversal easier to implement. In total, using the CRS scheme reduces the amount of stored numbers from n^2 to $2nnz + n + 1$ [7].

▶ **Example 2.1.** Compressed row storage example.

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 5 & 0 & -2 & 0 \\ 3 & 0 & 0 & 1 & 6 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & -4 & 0 & 0 & 7 \end{pmatrix}$$

`values`	5	-2	3	1	6	-4	7
`column_indices`	2	4	1	4	5	2	5

`row_ptrs`	0	1	3	3	6	8

The **compressed column storage** (CCS) **scheme** operates in a similar way as the compressed row storage scheme, but the matrix is transposed before saving, and the arrays are renamed to `row_indices` and `column_ptrs`. Effectively, the original matrix is traversed by columns, unlike in CRS, where it is traversed by rows.

The **block compressed row storage scheme** is used to store matrices where non-zero entries arise only in square submatrices (blocks) with a regular pattern. Those matrices are usually a result of discretization of a system of partial differential equations. In the block compressed row storage scheme, the blocks are handled as dense matrices. This approach leads to $nnzb \times n_b$ numbers being stored, where $nnzb$ is the number of blocks and n_b is the size of each block [7].

2.2 Dense matrix representation

In this section, we will present matrix storage schemes which are used in dense numerical linear algebra libraries, mainly BLAS libraries (see Chapter 3) and LAPACK (see Section 5.1). Some of the matrix classes that these storage schemes are designed for may be considered sparse, since not all of the entries of those matrices are stored. However, in this text, they will be labeled as dense.

This section mainly follows the structure of [5] and [6].

2.2.1 Full storage scheme

The **full storage scheme** (or **conventional storage**) is ideal for storing matrices that we have no initial assumptions about. In this storage format, every matrix entry is stored, including the zero ones. There are several methods to map the entries from the two-dimensional matrix into the one-dimensional computer memory, which will be described in the following subsections.

2.2.1.1 Row major storage

The **row major full storage scheme** stores every matrix row in a contiguous memory area, but these areas may not have consecutive addresses. In those areas, the order of the matrix entries is determined by their column index in the original matrix. The spacing between the first bytes of those areas is defined by a parameter called the **leading dimension** of the matrix, which we will label `LDA` in this section. Naturally, `LDA` may not be smaller than the number of columns, as that would make those memory areas overlap, meaning that the computer would have to store more than 1 byte at a single memory address.

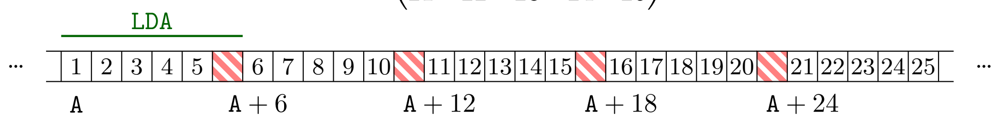
The element A_{ij} of a matrix A is thus stored at address

$$\mathtt{A} + \big((i-1)\cdot\mathtt{LDA} + (j-1)\big)\cdot\mathtt{size} \tag{2.1}$$

where `A` is the address of the first byte available for storing A and `size` is the number of bytes that a single entry of A occupies.

▶ **Example 2.2.** Example of row major full storage of a 5×5 matrix with $\mathtt{LDA}=6$.

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 & 25 \end{pmatrix}$$



2.2.1.2 Column major storage

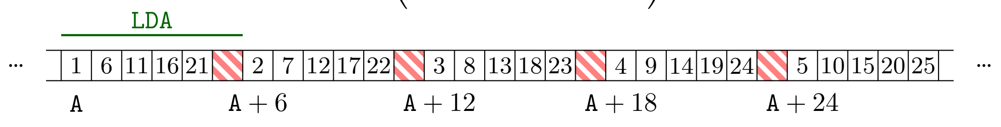
The matrix A is stored using the **column major full storage scheme** in the same way as A^T is stored using the row major full storage scheme. That is, the element A_{ij} is stored at address

$$\mathtt{A} + \big((j-1)\cdot\mathtt{LDA} + (i-1)\big)\cdot\mathtt{size} \tag{2.2}$$

Unlike in row major storage, `LDA` may be lesser than the number of columns, but it has to be greater than or equal to the number of rows.

▶ **Example 2.3.** Example of column major full storage of a 5×5 matrix with $\mathtt{LDA}=6$.

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 & 25 \end{pmatrix}$$



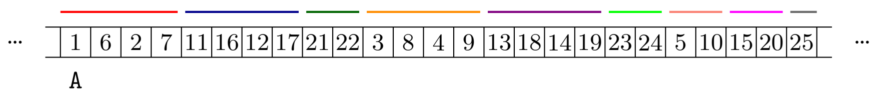
2.2.1.3 Block storage

When running blocked algorithms on matrices, it can be beneficial to store the entries of a single block in a compact way. If all blocks not containing entries from the last row or the last column of the matrix (we call those the main blocks) have the same size, and the other blocks do not have a larger width or height than the main blocks, we can utilize the **block storage scheme**. It works by storing the individual blocks as matrices using the row major or column major full storage scheme in contiguous memory areas (that is, $LDA = n$ or $LDA = m$, respectively). The blocks are stored consecutively, and their order is determined by the same column major or row major full storage scheme. It is also possible to use row major storage for the individual blocks and the column major scheme for the block order, or vice versa.

The indices of the block that a particular entry belongs to can be determined as floor values of the entry indices divided by the main block width or height (both the block and the entry indices are assumed to start from 0 in this case).

▶ **Example 2.4.** Example of column major block storage of a 5×5 matrix with 2×2 main blocks.

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 & 25 \end{pmatrix}$$



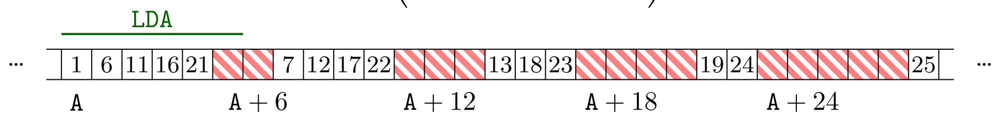
2.2.2 Triangular and symmetric matrix representation

There are two main options for storing a triangular matrix.

Firstly, we can modify the full storage scheme to only store and access the elements above and on the main diagonal (for upper triangular matrices) or below and on the main diagonal (for lower triangular matrices). That leads to less space being used compared to full storage, although the extra free space is created in non-contiguous chunks of bytes. It usually improves the performance of matrix operations due to fewer cache misses. The formula used to determine the address of the i,jth element of matrix A is identical to (2.1) or (2.2), depending on whether the row major or the column major modified full storage scheme is used.

▶ **Example 2.5.** Example of modified column major full storage of a 5×5 lower triangular matrix with `LDA` $= 6$.

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 6 & 7 & 0 & 0 & 0 \\ 11 & 12 & 13 & 0 & 0 \\ 16 & 17 & 18 & 19 & 0 \\ 21 & 22 & 23 & 24 & 25 \end{pmatrix}$$



The second option is to use the **packed storage scheme**. In this thesis, only the **column major packed storage scheme** will be introduced, although there is a row major variant of this storage format. For upper triangular matrices, we store elements above or on the main diagonal from every column in a contiguous vector ordered by their row index. Those vectors are then stored consecutively in an ascending order determined by size.

The i, jth element of A, where $i \leq j$, is therefore stored on the address

$$\texttt{A} + \left((i-1) + \frac{j \cdot (j-1)}{2}\right) \cdot \texttt{size} \tag{2.3}$$

where `A` and `size` have the same meaning as in the previous section.

For lower triangular matrices, we only store the part of each column containing the elements below or on the main diagonal. Those vectors are then stored consecutively from largest to smallest. For $i \geq j$, we store the entry A_{ij} of a lower triangular matrix $A \in \mathbb{R}^{n,n}$ on address

$$\texttt{A} + \left((i-1) + \frac{(j-1) \cdot (2 \cdot n - j)}{2}\right) \cdot \texttt{size} \tag{2.4}$$

▶ **Example 2.6.** Example of column major packed storage of a 5×5 lower triangular matrix.

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 6 & 7 & 0 & 0 & 0 \\ 11 & 12 & 13 & 0 & 0 \\ 16 & 17 & 18 & 19 & 0 \\ 21 & 22 & 23 & 24 & 25 \end{pmatrix}$$

...	1	6	11	16	21	7	12	17	22	13	18	23	19	24	25	...
	A															

Symmetric and hermitian matrices can utilize the same storage schemes as triangular matrices, as the entries below the diagonal are fully defined by the corresponding entries above the diagonal, so only one of these parts of a matrix needs to be stored. However, matrix operations on symmetric and Hermitian matrices have to be implemented separately from triangular matrix operations.

2.2.3 Band matrix representation

▶ **Definition 2.7.** *A band matrix with kl subdiagonals and ku superdiagonals is a matrix where for each $i, j \in \{1, \ldots, n\}$:*

$$j - i > ku \Rightarrow A_{ij} = 0 \quad \textit{and} \quad i - j > kl \Rightarrow A_{ij} = 0$$

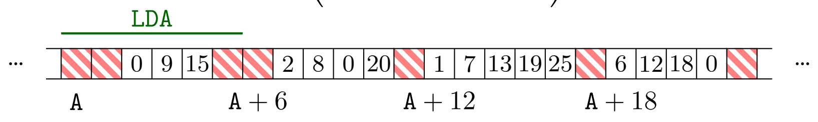
In the **band storage scheme**, we store a band matrix A with ku superdiagonals and kl subdiagonals as a $(kl + ku + 1) \times n$ matrix AB using the column major full storage scheme with $\texttt{LDAB} \geq ku + kl + 1$, where the entries $j - ku$ to $j + kl$ of the jth column of A are stored as the jth column of AB (the entries whose indices are out of bounds are left out).

In summary, the elements A_{ij} for $j - i \leq ku$ and $i - j \leq kl$ is stored in the address

$$\texttt{A} + \big(i + ku - j + (j+1) \cdot \texttt{LDAB}\big) \cdot \texttt{size} \tag{2.5}$$

▶ **Example 2.8.** Example of band storage of a 5×5 band triangular matrix with $ku = 2, kl = 1, \texttt{LDAB} = 6$.

$$A = \begin{pmatrix} 1 & 2 & 0 & 0 & 0 \\ 6 & 7 & 8 & 9 & 0 \\ 0 & 12 & 13 & 0 & 15 \\ 0 & 0 & 18 & 19 & 20 \\ 0 & 0 & 0 & 0 & 25 \end{pmatrix}$$



Chapter 3
Description of used BLAS routines

BLAS (Basic Linear Algebra Subprograms) is a set of routines that perform basic operations in linear algebra [8]. A BLAS library is a library providing implementations for all BLAS routines.

A so-called Reference BLAS library is available at [8]. This implementation can be used in various applications where basic linear algebra operations need to be performed, though it is not meant to be used for production runs. The main reason is its performance, which is suboptimal in comparison to highly optimized (usually platform-specific) BLAS libraries.

The more common way to utilize BLAS is to use machine-specific BLAS libraries, usually provided by the computer vendor (such as Intel OneApi MKL for Intel x86 processors) or independent software vendors (such as BLIS for AMD x86 processors).

The Reference BLAS library has, for the most part, two use cases:

- A last resort option for users
- A working example (and API reference) for testing other BLAS libraries

3.1 BLAS routine subsets

BLAS routines are divided into three categories [8]:

- **Level 1 BLAS** – routines performing scalar-vector or vector-vector operations of complexity $\mathcal{O}(n)$
- **Level 2 BLAS** – routines performing matrix-vector operations of complexity $\mathcal{O}(n^2)$
- **Level 3 BLAS** – routines performing matrix-matrix operations of complexity $\mathcal{O}(n^3)$

As we can see in the following three subsections, this categorization also has a historical context. The content of these subsections will closely follow [9], [10] and [11], respectively.

3.1.1 Level 1 BLAS

The historically first BLAS package, published by C. L. Lawson, R. J. Hanson, D. R. Kincaid and F. T. Krogh in 1979, was comprised of the set of routines that we now call Level 1 BLAS [9]. It was written in Fortran, and it included 38 routines for working with single-precision, double-precision, complex or extended-precision vectors. Examples of these routines are:

- Dot products (`DDOT`, `SDSDOT`, `CDOTU`)
- Givens rotation construction and application (`DROTG`, `DROT`)
- Euclidean and Manhattan norm calculation (`DNRM2`, `DASUM`)

3.1.2 Level 2 BLAS

In 1988, an extended set of BLAS routines was published by J. J. Dongarra, J. Du Croz, S. Hammarling and R. J. Hanson [10]. The newly developed routines performed matrix-vector operations of three types:

- Matrix-vector products (`GEMV`, `SYMV`, `TRMV`)
- Solution of triangular linear systems of equations (`TRSV`)
- Rank-1 or rank-2 updates (`GER`, `SYR`, `SYR2`)

The paper [10] also introduced the Level 1 BLAS and Level 2 BLAS terminology, noting that the newly-introduced routines could be implemented by series of calls to the original routine set from 1979, but it was inefficient (and therefore not recommended) to use such an implementation.

As the new routine set included matrix operations, unlike its predecessor, it had to account for various types of matrices (general, symmetric, hermitian, triangular, band matrices) and various forms of storage (packed matrices) that were used in different applications. That is why it introduced a naming convention with two prefixes – one denoting the scalar data type and the other denoting the matrix type. This naming convention would later be used in Level 3 BLAS [11] as well.

▶ **Example 3.1** (Level 2 BLAS routine naming convention example)**.**

$$\underbrace{\mathbf{D}}_{\substack{\text{Scalar data type}\\ \textbf{D}\text{ouble precision}}}\qquad\underbrace{\mathbf{SB}}_{\substack{\text{Matrix type}\\ \textbf{S}\text{ymmetric }\textbf{B}\text{and matrix}}}\qquad\underbrace{\mathbf{MV}}_{\substack{\text{Operation type}\\ \textbf{M}\text{atrix-}\textbf{V}\text{ector product}}}$$

Some newly introduced argument conventions (e.g., the `TRANS` and `UPLO` arguments) would also later make their way to Level 3 BLAS.

3.1.3 Level 3 BLAS

Level 2 BLAS routines offered efficient matrix-vector computations suited mostly for vector processing machines [11]. Many matrix-matrix operations could be performed by sequences of calls to Level 2 BLAS routines (e.g., matrix-matrix multiplication could be carried out by calling the matrix-vector multiplication routine `DGEMV` on every column of the right-hand side matrix). That approach, however, proved to be inefficient for computers with a hierarchical structure of memory (i.e., global memory, several levels of cache and vector registers), where better performance could be obtained by splitting the matrices into blocks and performing those operations on the blocks.

For that reason, no more than 2 years after Level 2 BLAS was made public, a new article describing Level 3 BLAS routines was published along with a reference implementation [11]. It followed naming conventions from Level 2 BLAS (see Example 3.1), and it introduced four new types of operations:

- Matrix-matrix products (GEMM, SYMM, HEMM)
- Rank-k or rank-2k updates of symmetric/hermitian matrices (SYRK, HERK, SYR2K)
- Products of general matrices and triangular matrices (TRMM)
- Solution of triangular linear systems of equations with a right-hand side matrix (TRSM)

The scope of the new set was intentionally limited – for instance, no routines for matrix decomposition were proposed, as they would later become part of a library that was in development at the time, LAPACK (see Section 5.1).

Three Level 3 BLAS routines are used in the blocked Cholesky decomposition algorithm, as proposed by [11]. Implementation of these three routines was also a part of this thesis. On that account, they will be described in more detail in the following sections of this chapter.

3.2 General matrix-matrix product (GEMM)

The GEMM routine performs the following computation:

$$C \leftarrow \alpha \mathrm{op}_A(A)\mathrm{op}_B(B) + \beta C \tag{3.1}$$

The values of α and β are passed to GEMM as parameters ALPHA and BETA. Assuming that A, B and C are real matrices, the meaning of op_A and op_B is determined by the TRANSA and TRANSB parameter values as follows:

- If TRANSA=N, then A is an $M \times K$ matrix and $\mathrm{op}_A(A) = A$
- If TRANSA=T or TRANSA=C, then A is a $K \times M$ matrix and $\mathrm{op}_A(A) = A^T$
- If TRANSB=N, then B is a $K \times N$ matrix and $\mathrm{op}_B(B) = B$
- If TRANSB=T or TRANSB=C, then B is an $N \times K$ matrix and $\mathrm{op}_B(B) = B^T$

The C value of these parameters marks a Hermitian transpose, which, in the case of real matrices, is equivalent to a transpose as specified in Definition 1.1 [11].

The other parameters of GEMM determine the matrix sizes (integer-valued parameters M, N and K), the pointers to the matrices (parameters A, B and C) and the leading dimensions (parameters LDA, LDB and LDC).

A naive (simple but unoptimized) implementation of the GEMM routine could be described with the following pseudocode (we consider TRANSA=N and TRANSB=N and ignore leading dimensions for simplicity):

```
Data: Positive integers $M, N, K$, matrices $A \in \mathbb{R}^{M,K}, B \in \mathbb{R}^{K,N}, C \in \mathbb{R}^{M,N}$
Result: The updated matrix $C$
for $i \leftarrow 1; i \leq M; i \leftarrow i + 1$ do
    for $j \leftarrow 1; j \leq N; j \leftarrow j + 1$ do
        $C_{ij} \leftarrow \beta C_{ij}$
        for $k \leftarrow 1; k \leq K; k \leftarrow k + 1$ do
            $C_{ij} \leftarrow C_{ij} + \alpha A_{ik} B_{kj}$
        end
    end
end
return $C$
```

Algorithm 2: Naive GEMM for TRANSA=N and TRANSB=N without leading dimensions

The number of floating point operations performed is:

- MN multiplications on line 3
- $2MNK$ multiplications on line 5
- MNK additions on line 5

In total, we have $3MNK + MN$ floating point operations. If the values of α and β are both equal to one, which is a common case when benchmarking `GEMM` implementations, we can omit MNK multiplications on line 5 and remove line 3 altogether. That leaves us with $2MNK$ floating point operations.

It is also worth noting that the total number of floating point operations is the same for all values of `TRANSA`, `TRANSB` and leading dimensions [12].

3.3 Symmetric matrix rank-k update (`SYRK`)

The `SYRK` routine calculates

$$C \leftarrow \alpha A A^T + \beta C \tag{3.2}$$

for an $N \times K$ matrix A if `TRANSA=N` and

$$C \leftarrow \alpha A^T A + \beta C \tag{3.3}$$

for a $K \times N$ matrix A if `TRANSA=T`. The matrix C is symmetric with the size $N \times N$ in both cases.

The other parameters are:

- `UPLO` – determines whether the upper (value `U`) or lower (value `L`) part of C is stored (see Subsection 2.2.2)
- `N` and `K` – the matrix sizes
- `LDA` and `LDC` – the matrix leading dimensions
- `A` and `C` – the matrices
- `ALPHA` and `BETA` – the α and β values used in equations (3.2) and (3.3)

For values `TRANSA=N UPLO=U`, `SYRK` could be naively implemented as follows (again, ignoring leading dimensions):

```
   Data: Positive integers $N, K$, matrices $A \in \mathbb{R}^{N,K}$ and $C \in \mathbb{R}^{N,N}$
   Result: The updated matrix $C$
1  for $i \leftarrow 1; i \leq N; i \leftarrow i + 1$ do
2      for $j \leftarrow i; j \leq N; j \leftarrow j + 1$ do
3          $C_{ij} \leftarrow \beta C_{ij}$
4          for $k \leftarrow 1; k \leq K; k \leftarrow k + 1$ do
5              $C_{ij} \leftarrow C_{ij} + \alpha A_{ik} A_{jk}$
6          end
7      end
8  end
9  return $C$
```

Algorithm 3: Naive `SYRK` for `TRANSA=N` and `UPLO=U` without leading dimensions

The number of times line 3 is executed is

$$\sum_{i=1}^{N}\sum_{j=i}^{N} 1 = \sum_{i=1}^{N}(N-i+1) = N^2 - \sum_{i=1}^{N} i + N = N^2 - \frac{N(N-1)}{2} + N = \frac{1}{2}N^2 + \frac{3}{2}N.$$

Line 5 is then executed $K\big(\frac{1}{2}N^2 + \frac{3}{2}N\big)$ times.

That leaves us with $K\big(\frac{1}{2}N^2 + \frac{3}{2}N\big)$ additive and $\frac{1}{2}N^2 + \frac{3}{2}N + 2K\big(\frac{1}{2}N^2 + \frac{3}{2}N\big) = (2K+1)\big(\frac{1}{2}N^2 + \frac{3}{2}N\big)$ multiplicative floating point operations, or

$$K\Big(\frac{1}{2}N^2 + \frac{3}{2}N\Big) + (2K+1)\Big(\frac{1}{2}N^2 + \frac{3}{2}N\Big) = (3K+1)\Big(\frac{1}{2}N^2 + \frac{3}{2}N\Big)$$

floating point operations in total.

In case of both α and β being equal to one, we end up with $K\big(\frac{1}{2}N^2 + \frac{3}{2}N\big)$ additive and $K\big(\frac{1}{2}N^2 + \frac{3}{2}N\big)$ multiplicative floating point operations, totaling $K\big(N^2 + 3N\big)$ floating point operations.

According to [12], the analysis yields the same result for other values of `TRANSA` and `UPLO`.

3.4 Solution of triangular systems of equations with multiple right-hand sides (TRSM)

The `TRSM` routine receives a nonsingular triangular matrix A and an $M \times N$ matrix B. It calculates

$$B \leftarrow \alpha \mathrm{op}(A)^{-1} B \tag{3.4}$$

where $A \in \mathbb{R}^{M\times M}$ if `SIDE` $=$ `L` and

$$B \leftarrow \alpha B \mathrm{op}(A)^{-1} \tag{3.5}$$

where $A \in \mathbb{R}^{N\times N}$ if `SIDE` $=$ `R`.

If the value of the `DIAG` parameter is `U`, all of the entries on the main diagonal of A are assumed to equal one, allowing us to skip one floating point division for every entry of B (see Algorithm 4). If the entries on the main diagonal are not known to be one, the user should use the `N` value of `DIAG`.

The value of α is determined by the parameter `ALPHA` and op_A is determined by `TRANSA` in the same way as described in Section 3.2. The `UPLO` parameter defines the stored part of A (see Subsection 2.2.2). The rest of the parameters represent the matrices (`A` and `B`), their dimensions (`M` and `N`) and their leading dimensions (`LDA` and `LDB`).

As stated above, A is assumed to be nonsingular. The `TRSM` routine, however, does not explicitly check for its nonsingularity. As a consequence, if a singular matrix (or a "nearly singular" matrix[1]) matrix A is used, the resulting matrix may vastly differ from the matrix specified in equations (3.4) or (3.5) due to numerical stability issues.

For the rest of this section, we name the newly calculated matrix X. When `SIDE` $=$ `L` and `TRANSA` $=$ `N`, we can rewrite the calculation (3.5) as an equation and then rearrange it in the following way:

$$\begin{aligned}
X &= \alpha A^{-1} B \\
AX &= A(\alpha A^{-1})B \\
AX &= \alpha(AA^{-1})B \\
AX &= \alpha I B \\
AX &= \alpha B
\end{aligned}$$

[1]By "nearly singular", we mean a matrix whose condition number approaches positive infinity.

We further suppose that `UPLO` = `L`. The matrix X can then be computed in an entry-wise manner:

$$\begin{aligned}
(AX)_{ij} &= (\alpha B)_{ij} \\
\sum_{k=1}^{N} A_{ik} X_{kj} &= \alpha B_{ij} \\
\sum_{k=1}^{i} A_{ik} X_{kj} &= \alpha B_{ij} \\
A_{ii} X_{ij} + \sum_{k=1}^{i-1} A_{ik} X_{kj} &= \alpha B_{ij} \\
A_{ii} X_{ij} &= \alpha B_{ij} - \sum_{k=1}^{i-1} A_{ik} X_{kj} \\
X_{ij} &= \frac{\alpha B_{ij} - \sum_{k=1}^{i-1} A_{ik} X_{kj}}{A_{ii}}
\end{aligned}$$

When transforming the sum in the left-hand side of the equation from line 2 to line 3, we eliminated all summands containing the term A_{ik} where $k > i$. Those terms are all equal to zero due to A being lower triangular.

When calculating X_{ij} using the last equality, all entries in the same column with a smaller row index must have already been calculated. Thus, if we choose any entry calculation order which satisfies those dependencies, such as a row major order where rows are calculated top to bottom, the matrix X can be calculated in the place of the original matrix B.

Assuming that we also know that all of the entries on the main diagonal are equal to one (that is, `DIAG` = `U`), we can skip the division by A_{ii}. All of these findings lead us to the following naive `TRSM` algorithm:

```
   Data: Positive integers $M, N$, matrices $A \in \mathbb{R}^{M,M}$ and $B \in \mathbb{R}^{M,N}$
   Result: The updated matrix $B$
 1 for $i \leftarrow 1; i \leq M; i \leftarrow i + 1$ do
 2     for $j \leftarrow 1; j \leq N; j \leftarrow j + 1$ do
 3         $B_{ij} \leftarrow \alpha B_{ij}$
 4         for $k \leftarrow 1; k < i; k \leftarrow k + 1$ do
 5             $B_{ij} \leftarrow B_{ij} - A_{ik} B_{kj}$
 6         end
 7         if DIAG = N then
 8             $B_{ij} \leftarrow B_{ij} / A_{ii}$
 9         end
10     end
11 end
12 return $B$
```

Algorithm 4: Naive `TRSM` for `SIDE` = `L` and `TRANSA` = `N` without leading dimensions

In this case (`SIDE` = `L` and `TRANSA` = `N`), we can see that the algorithm performs:

- MN floating point operations on line 3,
- $2 \cdot \left(\frac{1}{2} M(M-1)N\right) = M(M+1)N$ operations on line 5,
- MN operations on line 8 (only if `DIAG` = `N`)

Summed up, that is $M\big(2N + (M-1)N\big)$ floating point operations. If $\alpha = 1$, all operations on line 3 are left out, leaving us with $MN + M(M-1)N = MN(1 + M - 1) = M^2N$ operations.

That number of operations stays the same for both values of `TRANSA` and both values of `UPLO`. If `DIAG` $=$ `U`, it further reduces to $M^2N - MN$. The floating point operation count also changes for `SIDE` $=$ `R` to MN^2 and $MN^2 - MN$ for values of `DIAG` being equal to `N` and `U`, respectively. [12]

3.5 The CBLAS interface

The content of this section closely follows [13].

The original BLAS specifications were made primarily with Fortran in mind. As Fortran is a compiled language, the routines can, by all means, be called from C/C++ as well. Furthermore, there is no functionality of BLAS that C/C++ programmers may be unable to access via the Fortran interface. However, differences in the design of the C and Fortran languages lead to inconveniences for C/C++ programmers who use the Fortran BLAS interface, such as the necessity to perform additional argument preprocessing and return value postprocessing or the increased difficulty of debugging errors due to fewer compile-time checks. Examples of those inconveniences are:

- Arguments have to be passed by reference, not by value
- No compile-time checking of argument values is performed
- No C include files are available
- Only column major matrix storage is supported, as it is the default storage scheme in Fortran
- Vector indices start with 1 (in C/C++, arrays are indexed starting from 0)

As a result, a new C interface for BLAS routines, named CBLAS, was proposed [13]. The routines were newly named using non-capital letters and a `cblas_` prefix (e.g., the `DGEMM` routine became `cblas_dgemm` in CBLAS). The vector index numbering was shifted from $1, 2, \dots, N$ to $0, 1, \dots, N - 1$ and arguments were made to be passed by value. An include file named `cblas.h` was made newly available so that the C compiler could check the routine calls against the specified signatures during compile time, leading to fewer errors.

The CBLAS interface also started using enumerated types for arguments that were characters in the original Fortran interface. That allowed for tighter error checking – for instance, passing `CblasNoTrans` as a value of the `CBLAS_SIDE` argument raises a compile-time error, but in the original BLAS interface, passing the character `N` as a value of the `SIDE` argument only produces a run-time error, which is harder to debug for the programmer.

Another feature introduced by the CBLAS interface is row major storage support. By convention, the column major storage scheme is used for matrices in Fortran, so the Fortran BLAS interface does not support row major matrices at all. The CBLAS interface adds a new parameter `CBLAS_ORDER` to all routines working with matrices. Using this parameter, programmers may specify the storage scheme (column major or row major) used for all matrices passed to the routine. Reference [13] notes that for Level 1 and Level 3 BLAS routines, no extra storage is required to support row major BLAS operations. However, some Level 2 BLAS routines may require up to $\mathcal{O}(n)$ extra storage and $\mathcal{O}(n)$ extra floating point operations.

The paper also mentions discussions about support for two-dimensional arrays in C (i.e., arrays of pointers), which was not added in the end, as conversions between two-dimensional C arrays and Fortran-style one-dimensional arrays would lead to significant performance losses.

Chapter 4

Cholesky decomposition

Lay, Lay and McDonald [2] define a **matrix factorization** (also called a **matrix decomposition**) as an equation that represents a matrix as a product of two or more matrices, which we call **factors**. We typically want the factors to have specific properties, such as being upper or lower triangular, diagonal or orthogonal [14]. Optimized routines that perform those factorizations can then be used to solve other problems efficiently, for example:

- The LU, Cholesky and QR factorizations are used to solve systems of linear equations
- The QR factorization is used in analysis of variance (ANOVA) methods
- The singular value decomposition (SVD) is used in principal component analysis[1]
- The spectral decomposition is used in locally linear embedding[2]

One of the most important factorizations is the Cholesky factorization, which is defined for symmetric positive definite matrices. Along with the LU and QR factorizations, it is considered to be one of the three elementary matrix factorizations (also called the “Three Amigos”) used in numerical linear algebra.

▶ **Definition 4.1** ([3])**.** *A **Cholesky factorization** (or **Cholesky decomposition**) of a symmetric positive definite matrix $A \in \mathbb{R}^{n,n}$ is an equation that has the form*

$$A = LL^T \tag{4.1}$$

where L is a lower triangular matrix with $L_{ii} > 0$ for all $i \in \{1, \dots, n\}$.

By finding the Cholesky factorization of a symmetric positive definite matrix $A \in \mathbb{R}^{n,n}$, we typically mean finding the factor L, as the other factor L^T can be constructed from L trivially.

In general, the Cholesky decomposition can be defined for Hermitian matrices (matrices with complex entries that are equal to the transpose of their complex conjugate). In this thesis, however, we limit ourselves to matrices with real-valued entries, for which the complex conjugate transpose equals the transpose.

In some definitions of the Cholesky factorization (including the definition in Reference [3]), the defining equation has the form

$$A = U^T U, \tag{4.2}$$

[1]A linear method of dimensionality reduction used in machine learning.
[2]A non-linear method of dimensionality reduction used in machine learning.

where U is an upper triangular matrix with positive entries on the main diagonal. Both of these definitions are equivalent – if we know the matrix U used in (4.2), we can obtain the matrix L from (4.1) as $L = U^T$, since

$$A = A^T = (U^T U)^T = U^T (U^T)^T = LL^T.$$

We will now show that every symmetric positive definite matrix has a Cholesky decomposition:

▶ **Lemma 4.2.** *Let $A \in \mathbb{R}^{n,n}$ be a symmetric positive definite matrix and $L \in \mathbb{R}^{n,n}$ a lower triangular matrix such that $A = LL^T$. Then $L_{ii} \neq 0$ for all $i \in \{1, \dots, n\}$.*

Proof. Suppose that $L_{ii} = 0$ for some $i \in \{1, \dots, n\}$. Then L is singular as per Theorem 1.15 since $\det L = 0$. Following Theorem 1.12, L^T is also singular. Thus, according to Theorem 1.20, there is a non-zero vector $x \in \mathbb{R}^{n,1}$ such that:

$$\begin{aligned} \|L^T x\| &= 0 \\ L^T x &= \theta \\ LL^T x &= \theta \\ Ax &= \theta \\ x^T A x &= 0 \end{aligned}$$

which is contradictory to the assumption that A is positive definite. ◀

▶ **Theorem 4.3.** *Let $A \in \mathbb{R}^{n,n}$ be a symmetric positive definite matrix. Then A has a unique Cholesky decomposition, i.e., there exists precisely one lower triangular matrix $L \in \mathbb{R}^{n,n}$ with positive entries on the main diagonal such that $A = LL^T$.*

Proof. We will show both the existence and the uniqueness of L by finding an explicit formula for all of its entries.

For all $i, j \in \{1, \dots, n\}, i \geq j$, we have:

$$A_{ij} = \sum_{k=1}^{n} L_{ik} (L^T)_{kj} = \sum_{k=1}^{n} L_{ik} L_{jk}$$

Since L is lower triangular and $i \geq j$, we have $L_{ik} L_{jk} = 0$ for all $k > j$:

$$A_{ij} = \sum_{k=1}^{j} L_{ik} L_{jk}. \tag{4.3}$$

If $i = j$, we then obtain:

$$\begin{aligned} A_{ii} &= \sum_{k=1}^{i} L_{ik}^2 \\ A_{ii} &= L_{ii}^2 + \sum_{k=1}^{i-1} L_{ik}^2 \\ L_{ii}^2 &= A_{ii} - \sum_{k=1}^{i-1} L_{ik}^2 \\ |L_{ii}| &= \sqrt{A_{ii} - \sum_{k=1}^{i-1} L_{ik}^2} \end{aligned}$$

Since the diagonal entries of L are required by definition to be positive, we set:

$$L_{ii} = \sqrt{A_{ii} - \sum_{k=1}^{i-1} L_{ik}^2} \tag{4.4}$$

We can see that $L_{ii} > 0$, since a square root of a non-negative number is non-negative and we know that $L_{ii} \neq 0$ from Lemma 4.2.

Following (4.3) for $i > j$, we obtain:

$$A_{ij} = \sum_{k=1}^{j} L_{ik} L_{jk}$$

$$A_{ij} = L_{ij} L_{jj} + \sum_{k=1}^{j-1} L_{ik} L_{jk}$$

$$L_{ij} L_{jj} = A_{ij} - \sum_{k=1}^{j-1} L_{ik} L_{jk}$$

$$L_{ij} = \frac{A_{ij} - \sum_{k=1}^{j-1} L_{ik} L_{jk}}{L_{jj}} \tag{4.5}$$

The correctness of the last line follows from Lemma 4.2.

◀

4.1 Unblocked algorithm

In the proof of Theorem 4.3, we found the explicit formulas for all entries of the matrix L. Both of the formulas (4.4) and (4.5) only require the entries in columns 1 to $j-1$ to be known when calculating an entry in the jth column. Thus, we can calculate the Cholesky decomposition using formulas (4.4) and (4.5) in a column-wise order:

```
**Data:** Symmetric positive definite matrix $A \in \mathbb{R}^{n,n}$
**Result:** Lower triangular matrix $L$ with positive diagonal entries such that $A = LL^T$
1 $L \leftarrow \Theta$
2 **for** $j \leftarrow 1; j \leq n; j \leftarrow j+1$ **do**
3     $L_{jj} \leftarrow \sqrt{A_{jj} - \sum_{k=1}^{j-1} L_{jk}^2}$
4     **for** $i \leftarrow j+1; j \leq n; i \leftarrow i+1$ **do**
5         $L_{ij} \leftarrow (A_{ij} - \sum_{k=1}^{j-1} L_{ik} L_{jk})/L_{jj}$
6     **end**
7 **end**
8 **return** L
```

Algorithm 5: Unblocked Cholesky decomposition (variant of [3], Algorithm 23.1)

Note that when calculating the entry L_{jj} on line 3, we only use A_{jj} and the entries of L that are already calculated. Similarly, we only access A_{ij} and the already calculated entries of L when calculating L_{ij} on line 5. Thus, if the programmer does not require the matrix A to remain unchanged, we can perform all of the computations in place – that is, remove line 1 and change all memory accesses to L to memory accesses to A instead. Since we do not overwrite the entries above the diagonal, the returned matrix will likely not be lower triangular. However, if we consider the entries of the returned matrix above the main diagonal to be zero, the correctness of the algorithm still holds.

4.1.1 Time complexity of the unblocked algorithm

On line 3, we perform $j-1$ multiplications, $j-1$ additions/subtractions and a square root computation. The overall amount of floating point operations executed on line 3 is thus

$$\sum_{j=1}^{n} j-1+j-1+1=\sum_{j=1}^{n} 2j-1=2\frac{n(n+1)}{2}-n=n(n+1)-n=n^2$$

On line 5, $j-1$ multiplications and $j-1$ additions/subtractions are performed as well as a single floating point division. The number of floating point operations performed on line 5 is

$$\begin{aligned}\sum_{j=1}^{n}\sum_{i=j+1}^{n} j-1+j-1+1 &= \sum_{j=1}^{n}(n-j)(2j-1)=2n\Big(\sum_{j=1}^{n} j\Big)-2\Big(\sum_{j=1}^{n} j^2\Big)-n^2+\Big(\sum_{j=1}^{n} j\Big)=\\ &=2n\frac{n(n+1)}{2}-2\frac{n(n+1)(2n+1)}{6}+n^2-\frac{n(n+1)}{2}=\\ &=n^3+n^2-\frac{2n^3+3n^2+n}{3}-n^2+\frac{n^2+n}{2}=\\ &=\frac{1}{3}n^3-\frac{1}{2}n^2+\frac{1}{6}n\end{aligned}$$

In total, the number of floating point operations performed during the algorithm execution is $\frac{1}{3}n^3+\frac{1}{2}n^2+\frac{1}{6}n$.

In the following sections, we will denote the first factor in the Cholesky decomposition of a symmetric positive definite matrix A by POTF2(A) (that is, if $A=LL^T$, then we represent L by POTF2(A)). This comes from the fact that the routine implementing the unblocked Cholesky decomposition algorithm in the LAPACK library (see Section 5.1) is named POTF2.

4.2 Blocked algorithm

In this section, we split the matrices A and L into blocks. We first fix a number $nb \in \{1,\dots,n\}$ and we define:

- A_{ij} for $i,j \in \{1,\dots,\lceil n/nb\rceil\}$ as a submatrix of A obtained by removing the first $i\dots nb$ rows and the first $i\dots nb$ columns, and then removing all but the first nb rows and columns
- L_{ij} for $i,j \in \{1,\dots,\lceil n/nb\rceil\}$ as a submatrix of L obtained by removing the first $i\dots nb$ rows and the first $i\dots nb$ columns, and then removing all but the first nb rows and columns

In this section, A_{ij} will not denote the i,jth entry of A, but the i,jth block of A, unless stated otherwise.

```
Data: Symmetric positive definite matrix $A \in \mathbb{R}^{n,n}$
Result: Lower triangular matrix $L$ with positive diagonal entries such that $A = LL^T$
1  $L \leftarrow \Theta$
2  for $j \leftarrow 1; j \leq nb; j \leftarrow j+1$ do
3      $L_{jj} \leftarrow$ POTF2($A_{jj}$)
4      for $i \leftarrow j+1; i \leq nb; i \leftarrow i+1$ do
5          $L_{ij} \leftarrow A_{ij} \cdot \left((L_{jj})^T\right)^{-1}$   // this can be calculated using TRSM
6      end
7      for $k \leftarrow j+1; k \leq nb; k \leftarrow k+1$ do
8          for $i \leftarrow k+1; i \leq nb; i \leftarrow i+1$ do
9              if $i = k$ then
10                 $A_{kk} \leftarrow A_{kk} - L_{kj} \cdot (L_{kj})^T$   // this can be calculated using SYRK
11             else
12                 $A_{ik} \leftarrow A_{ik} - L_{ij} \cdot (L_{kj})^T$   // this can be calculated using GEMM
13             end
14         end
15     end
16 end
17 return L
```

Algorithm 6: Blocked Cholesky decomposition ([15], Algorithm 3)

For the purposes of proving the correctness of Algorithm 6, we denote the submatrix of A at the start of the $(j+1)$th iteration of the outer loop on line 2 obtained by removing the first $j \cdot nb$ rows and the first $j \cdot nb$ columns by B_j. Trivially, we can see that $A = B_0$.

For a particular $j \in \{1, \dots, \lceil \frac{n}{nb} \rceil - 1\}$, we partition $B_j = \begin{pmatrix} D & W^T \\ W & K \end{pmatrix}$, where $D \in \mathbb{R}^{nb,nb}$, $W \in \mathbb{R}^{n-(j+1)\cdot nb, nb}$ and $K \in \mathbb{R}^{n-(j+1)\cdot nb, n-(j+1)\cdot nb}$. Then by definition of B_j:

- D has the same value as $A_{(j+1)(j+1)}$ at the start of the $(j+1)$th iteration of the outer loop
- W has the same value as the block matrix $\begin{pmatrix} A_{(j+2)(j+1)} \\ \vdots \\ A_{\lceil \frac{n}{nb} \rceil (j+1)} \end{pmatrix}$ at the start of the $(j+1)$th iteration of the outer loop
- K has the same value as the block matrix $\begin{pmatrix} A_{(j+2)(j+2)} & \dots & A_{(j+2)\lceil \frac{n}{nb} \rceil} \\ \vdots & \ddots & \vdots \\ A_{\lceil \frac{n}{nb} \rceil (j+2)} & \dots & A_{\lceil \frac{n}{nb} \rceil \lceil \frac{n}{nb} \rceil} \end{pmatrix}$ at the start of the $(j+1)$th iteration of the outer loop

We further denote the submatrix of L obtained by removing the first $j \cdot nb$ rows and the first $j \cdot nb$ columns by L_j. The first column of the matrix L_j is thus comprised solely of the blocks of L that are modified during the $(j+1)$th iteration of the outer loop.

▶ **Lemma 4.4.** *Fix $j \in \{1, \dots, \lceil \frac{n}{nb} \rceil - 1\}$ and partition $B_j = \begin{pmatrix} D & W^T \\ W & K \end{pmatrix}$ as described above. If B_j is symmetric positive definite, then $B_{j+1} = K - WD^{-1}W^T$.*

Proof. Since $D = (B_j)_{11}$ is a leading principal submatrix of B_j, it is symmetric positive definite following Theorem 1.26. We denote $L_D =$ POTF2(D).

During the $(j+1)$th iteration of the outer loop, L_{jj} is set to POTF2(A_{jj}) and L_{ij} to $A_{ij} \cdot (L_{jj}^T)^{-1}$ for all $i \in \{j+1, \dots, nb\}$. The first column of L_j thus becomes

$$\begin{pmatrix}(L_j)11\\(L_j)21\\\vdots\\(L_j)n1\end{pmatrix}=\begin{pmatrix}\texttt{POTF2}\big((B_j)_{11}\big)\\(B_j)_{21}\big((L_j)_{11}^T\big)^{-1}\\\vdots\\(B_j)_{(\lceil\frac{n}{nb}\rceil-j)1}\big((L_j)_{11}^T\big)^{-1}\end{pmatrix}=\begin{pmatrix}L_D\\(B_j)_{21}\big(L_D^T\big)^{-1}\\\vdots\\(B_j)_{(\lceil\frac{n}{nb}\rceil-j)1}\big(L_D^T\big)^{-1}\end{pmatrix}.$$

For all $i,k\in\{j+1,\dots,\lceil\frac{n}{nb}\rceil\}$ where $i\neq k$:

- A_{kk} is set to $A_{kk}-L_{kj}\cdot(L_{kj})^T$ on line 10, and since no other modifications of A_{kj} and L_{kj} are performed during the $(j+1)$th iteration, we can conclude that

$$\begin{aligned}(B_{j+1})_{(k-j)(k-j)}&=(B_j)_{(k-j+1)(k-j+1)}-L_{kj}\cdot(L_{kj})^T=\\&=(B_j)_{(k-j+1)(k-j+1)}-(L_j)_{(k-j+1)1}\cdot(L_j)_{(k-j+1)1}^T\end{aligned}$$

- A_{ik} is set to $A_{ik}-L_{ij}\cdot(L_{kj})^T$ on line 12, and because the only lines where A_{ik}, L_{ij} and L_{kj} are modified during the $(j+1)$th iteration are 5 and 12, we observe that

$$\begin{aligned}(B_{j+1})_{(i-j)(k-j)}&=(B_j)_{(i-j+1)(k-j+1)}-L_{ij}\cdot(L_{kj})^T=\\&=(B_j)_{(i-j+1)(k-j+1)}-(L_j)_{(i-j+1)1}\cdot(L_j)_{(k-j+1)1}^T\end{aligned}$$

We thus obtain:

$$\begin{aligned}B_{j+1}&=\begin{pmatrix}(B_{j+1})_{11}&\dots&(B_{j+1})_{1(\lceil\frac{n}{nb}\rceil-j-1)}\\\vdots&\ddots&\vdots\\(B_{j+1})_{(\lceil\frac{n}{nb}\rceil-j-1)1}&\dots&(B_{j+1})_{(\lceil\frac{n}{nb}\rceil-j-1)(\lceil\frac{n}{nb}\rceil-j-1)}\end{pmatrix}=\\
&=K-\begin{pmatrix}(L_j)_{21}\cdot\big((L_j)_{21}\big)^T&\dots&(L_j)_{21}\cdot\big((L_j)_{(\lceil\frac{n}{nb}\rceil-j)1}\big)^T\\\vdots&\ddots&\vdots\\(L_j)_{(\lceil\frac{n}{nb}\rceil-j)1}\cdot\big((L_j)_{21}\big)^T&\dots&(L_j)_{(\lceil\frac{n}{nb}\rceil-j)1}\cdot\big((L_j)_{(\lceil\frac{n}{nb}\rceil-j)1}\big)^T\end{pmatrix}=\\
&=K-\begin{pmatrix}(L_j)_{21}\\\vdots\\(L_j)_{(\lceil\frac{n}{nb}\rceil-j)1}\end{pmatrix}\begin{pmatrix}(L_j)_{21}\\\vdots\\(L_j)_{(\lceil\frac{n}{nb}\rceil-j)1}\end{pmatrix}^T=\\
&=K-\begin{pmatrix}(B_j)_{21}(L_D^T)^{-1}\\\vdots\\(B_j)_{(\lceil\frac{n}{nb}\rceil-j)1}(L_D^T)^{-1}\end{pmatrix}\begin{pmatrix}(B_j)_{21}(L_D^T)^{-1}\\\vdots\\(B_j)_{(\lceil\frac{n}{nb}\rceil-j)1}(L_D^T)^{-1}\end{pmatrix}^T=\\
&=K-\begin{pmatrix}(B_j)_{21}(L_D^T)^{-1}\\\vdots\\(B_j)_{(\lceil\frac{n}{nb}\rceil-j)1}(L_D^T)^{-1}\end{pmatrix}\begin{pmatrix}(B_j)_{21}(L_D^{-1})^T\\\vdots\\(B_j)_{(\lceil\frac{n}{nb}\rceil-j)1}(L_D^{-1})^T\end{pmatrix}^T=\\
&=K-\begin{pmatrix}(B_j)_{21}(L_D^T)^{-1}\\\vdots\\(B_j)_{(\lceil\frac{n}{nb}\rceil-j)1}(L_D^T)^{-1}\end{pmatrix}\begin{pmatrix}L_D^{-1}(B_j)_{21}^T&\dots&L_D^{-1}(B_j)_{(\lceil\frac{n}{nb}\rceil-j)1}^T\end{pmatrix}=\\
&=K-W(L_D^T)^{-1}L_D^{-1}W^T=K-W(L_DL_D^T)^{-1}W^T=K-WD^{-1}W^T\end{aligned}$$

◀

▶ **Lemma 4.5.** *For a symmetric positive definite matrix $A\in\mathbb{R}^{n,n}$, the matrix A_{jj} on line 3 is symmetric positive definite in all iterations of the outer loop.*

Proof (inspired by the proof in [3], lecture 23)**.**

We will first prove by induction that all B_j are symmetric positive definite:

Base step $B_0 = A$ is symmetric positive definite.

Inductive step Let B_j be symmetric positive definite. Then D is also symmetric positive definite following Theorem 1.26.

We set $L_D = \texttt{POTF2(D)}$, $\tilde{L} = \begin{pmatrix} L_D & \theta^T \\ W(L_D^T)^{-1} & I \end{pmatrix}$ and $\tilde{B} = \begin{pmatrix} I & \theta^T \\ \theta & B_{j+1} \end{pmatrix}$ such that $\tilde{L}, \tilde{B} \in \mathbb{R}^{n-j\cdot nb, n-j\cdot nb}$, then:

$$\begin{aligned}
\tilde{L}\tilde{B}\tilde{L}^T &= \begin{pmatrix} L_D & \theta^T \\ W(L_D^T)^{-1} & I \end{pmatrix} \begin{pmatrix} I & \theta^T \\ \theta & B_{j+1} \end{pmatrix} \begin{pmatrix} L_D^T & \left(W(L_D^T)^{-1}\right)^T \\ \theta & I \end{pmatrix} = \\
&= \begin{pmatrix} L_D & \theta^T \\ W(L_D^T)^{-1} & B_{j+1} \end{pmatrix} \begin{pmatrix} L_D^T & \left(W(L_D^T)^{-1}\right)^T \\ \theta & I \end{pmatrix} = \\
&= \begin{pmatrix} L_D L_D^T & L_D\left(W(L_D^T)^{-1}\right)^T \\ W(L_D^T)^{-1}L_D^T & W(L_D^T)^{-1}\left(W(L_D^T)^{-1}\right)^T + B_{j+1} \end{pmatrix} = \\
&= \begin{pmatrix} L_D L_D^T & L_D\left(W(L_D^{-1})^T\right)^T \\ W(L_D^T)^{-1}L_D^T & W(L_D^T)^{-1}\left(W(L_D^{-1})^T\right)^T + B_{j+1} \end{pmatrix} = \\
&= \begin{pmatrix} L_D L_D^T & L_D L_D^{-1} W^T \\ W(L_D^T)^{-1}L_D^T & W(L_D^T)^{-1}L_D^{-1}W^T + B_{j+1} \end{pmatrix} = \\
&= \begin{pmatrix} D & W^T \\ W & W(L_D L_D^T)^{-1}W^T + B_{j+1} \end{pmatrix}
\end{aligned}$$

Following Lemma 4.4,

$$\begin{pmatrix} D & W^T \\ W & W(L_D L_D^T)^{-1}W^T + B_{j+1} \end{pmatrix} = \begin{pmatrix} D & W^T \\ W & WD^{-1}W^T + K - WD^{-1}W^T \end{pmatrix} = \begin{pmatrix} D & W^T \\ W & K \end{pmatrix} = \\ = B_j$$

Following Theorem 1.15 and Lemma 4.2, we can observe that $\tilde{L}$ is nonsingular since all of its diagonal entries are non-zero. $\tilde{L}^T$ and $(\tilde{L}^T)^{-1}$ are then also nonsingular thanks to Theorem 1.12. Given our supposition that B_j is positive definite, we have $x^T B_j x > 0$ for all non-zero $x \in \mathbb{R}^{n-j\cdot nb,1}$. We also know from Theorem 1.20 that $(\tilde{L}^T)^{-1}x$ is non-zero. We then obtain

$$x^T \tilde{B} x = x^T\left(\tilde{L}^{-1}B_j(\tilde{L}^T)^{-1}\right)x = \left((\tilde{L}^T)^{-1}x\right)^T B_j\left((\tilde{L}^T)^{-1}x\right).$$

which is positive due to the positive definiteness of B_j. As such, $\tilde{B}$ is positive definite and so is its principal submatrix B_{j+1}.

Finally, we can observe that at the start of the jth iteration of the outer loop, the matrix A_{jj} is positive definite, since it is a leading principal submatrix of B_{j-1}. ◀

▶ **Theorem 4.6.** *For a symmetric positive definite matrix $A \in \mathbb{R}^{n,n}$, Algorithm 6 returns the matrix* `POTF2`(A).

Proof. Using backward induction, we will prove that $L_j = \texttt{POTF2}(B_j)$ for all $j \in \{1, \dots, \lceil \frac{n}{nb} \rceil\}$:

Base step $L_{\lceil \frac{n}{nb} \rceil \lceil \frac{n}{nb} \rceil}$ is set to $\texttt{POTF2}(A_{\lceil \frac{n}{nb} \rceil \lceil \frac{n}{nb} \rceil})$ on line 3 during the last iteration of the outer loop.

Inductive step Suppose that $L_{j+1} = \texttt{POTF2}(B_{j+1})$ for any $j \in \{1, \dots, \lceil \frac{n}{nb} \rceil - 1\}$.

Using observations from the proof of Lemma 4.4, we have

$$\begin{pmatrix} (L_j)_{11} \\ (L_j)_{21} \\ \vdots \\ (L_j)_{n1} \end{pmatrix} = \begin{pmatrix} L_D \\ (B_j)_{21}\left(L_D^T\right)^{-1} \\ \vdots \\ (B_j)_{(\lceil \frac{n}{nb} \rceil - j)1}\left(L_D^T\right)^{-1} \end{pmatrix} = \begin{pmatrix} L_D \\ W\left(L_D^T\right)^{-1} \end{pmatrix}.$$

Thus:

$$\begin{aligned} L_j L_j^T &= \begin{pmatrix} L_D & \Theta \\ W\left(L_D^T\right)^{-1} & \texttt{POTF2}(B_{j+1}) \end{pmatrix} \begin{pmatrix} L_D & \Theta \\ W\left(L_D^T\right)^{-1} & \texttt{POTF2}(B_{j+1}) \end{pmatrix}^T = \\ &= \begin{pmatrix} L_D & \Theta \\ W\left(L_D^T\right)^{-1} & \texttt{POTF2}(B_{j+1}) \end{pmatrix} \begin{pmatrix} L_D^T & \left(W\left(L_D^T\right)^{-1}\right)^T \\ \Theta^T & \texttt{POTF2}(B_{j+1})^T \end{pmatrix} = \\ &= \begin{pmatrix} L_D & \Theta \\ W\left(L_D^T\right)^{-1} & \texttt{POTF2}(B_{j+1}) \end{pmatrix} \begin{pmatrix} L_D^T & L_D^{-1} W^T \\ \Theta^T & \texttt{POTF2}(B_{j+1})^T \end{pmatrix} = \\ &= \begin{pmatrix} L_D L_D^T & L_D L_D^{-1} W^T \\ W\left(L_D^T\right)^{-1} L_D^T & W\left(L_D^T\right)^{-1} L_D^{-1} W^T + \texttt{POTF2}(B_{j+1})\texttt{POTF2}(B_{j+1})^T \end{pmatrix} = \\ &= \begin{pmatrix} D & W^T \\ W & WD^{-1}W^T + B_{j+1} \end{pmatrix} \end{aligned}$$

We know that $K = WD^{-1}W^T + B_{j+1}$ from Lemma 4.4, and as such, we have

$$\begin{pmatrix} D & W^T \\ W & WD^{-1}W^T + B_{j+1} \end{pmatrix} = \begin{pmatrix} D & W^T \\ W & K \end{pmatrix} = B_j.$$

We have obtained the identity $L_j = \texttt{POTF2}(B_j)$ for all $j \in \{1, \dots, \lceil \frac{n}{nb} \rceil\}$. Specifically for $j = 0$, we have $L = L_0 = \texttt{POTF2}(B_0) = \texttt{POTF2}(A)$. ◀

In Algorithm 6, we can notice that the blocks A_{ik} for $i < k$ are only accessed in a single iteration of line 12, where they are modified. As such, we can eliminate some redundant floating point operations by rewriting the algorithm in the following form:

```
   Data: Symmetric positive definite matrix $A \in \mathbb{R}^{n,n}$
   Result: Lower triangular matrix $L$ with positive diagonal entries such that $A = LL^T$
 1 $L \leftarrow \Theta$
 2 for $j \leftarrow 1; j \leq nb; j \leftarrow j+1$ do
 3     $L_{jj} \leftarrow$ POTF2($A_{jj}$)
 4     for $i \leftarrow j+1; i \leq nb; i \leftarrow i+1$ do
 5         $L_{ij} \leftarrow A_{ij} \cdot \left((L_{jj})^T\right)^{-1}$   // this can be calculated using TRSM
 6     end
 7     for $k \leftarrow j+1; k \leq nb; k \leftarrow k+1$ do
 8         $A_{kk} \leftarrow A_{kk} - L_{kj} \cdot (L_{kj})^T$   // this can be calculated using SYRK
 9         for $i \leftarrow k+1; i \leq nb; i \leftarrow i+1$ do
10             $A_{ik} \leftarrow A_{ik} - L_{ij} \cdot (L_{kj})^T$   // this can be calculated using GEMM
11         end
12     end
13 end
14 return L
```

Algorithm 7: Improved blocked Cholesky decomposition (variant of [15], Algorithm 3)

Note that the reason why we first introduced Algorithm 6 was a more straightforward proof of correctness – the matrices B_j would not always be symmetric if we used Algorithm 7, but the matrices obtained by "mirroring" the lower triangular part of B_j along the main diagonal would.

A second improvement is to perform the calculation in place – that would be achieved by removing line 1 and replacing all references to L with references to A in Algorithm 7.

4.2.1 Parallelization of the blocked Cholesky decomposition algorithm

In Algorithm 7, we can notice that on lines 5, 8 and 10, we do not need to perform all of the computations in series. On line 5, we only have to know the value of the block L_{jj} in order to compute the blocks $L_{j(j+1)}$ to L_{jn}, but not the other blocks with the column index j. As such, the computations of the blocks $L_{j(j+1)}$ to L_{jn} are independent, and can thus be performed in parallel, because the only memory accessed by all computing threads is the block L_{jj}, which is only read, but not overwritten.

In a similar fashion, we can perform all of the computations on line 8 in a single iteration of 2 simultaneously, since we do not need to access any of the diagonal blocks $A_{(j+1)(j+1)}, \ldots, A_{nn}$ except for A_{kk} to be able to update the block A_{kk}. The block L_{kj} can be accessed by all of the threads running on line 8 at once due to the fact that L_{ij} is only read, but not written to.

Using the same reasoning, we can also perform all of the computations on line 10 in one iteration of 2 in parallel.

In summary, the parallelized algorithm can be expressed as follows:

```
    **Data:** Symmetric positive definite matrix $A \in \mathbb{R}^{n,n}$
    **Result:** Lower triangular matrix $L$ with positive diagonal entries such that $A = LL^T$
1 $L \leftarrow \Theta$
2 **for** $j \leftarrow 1; j \leq nb; j \leftarrow j+1$ **do**
3     $L_{jj} \leftarrow \texttt{POTF2}(A_{jj})$
4     Compute $L_{ij} \leftarrow A_{ij} \cdot \left((L_{jj})^T\right)^{-1}$ using TRSM for $i \in \{j+1, \dots, nb\}$ in parallel
5     Compute $A_{kk} \leftarrow A_{kk} - L_{kj} \cdot (L_{kj})^T$ using SYRK for $k \in \{j+1, \dots, nb\}$ in parallel
6     Compute $A_{ik} \leftarrow A_{ik} - L_{ij} \cdot (L_{kj})^T$ using GEMM for $k \in \{j+1, \dots, nb\}$ and
       $i \in \{k+1, \dots, nb\}$ in parallel
7 **end**
8 **return** L
```

Algorithm 8: Parallelized version of blocked Cholesky decomposition

4.3 Solving systems of linear equations using Cholesky decomposition

Cholesky decomposition can be used to solve systems of linear equations $Ax = b$ with a symmetric positive definite matrix A in the following way:

```
    **Data:** Symmetric positive definite matrix $A \in \mathbb{R}^{n,n}$
    **Data:** Vector $b \in \mathbb{R}^{n,1}$
    **Result:** Vector $x \in \mathbb{R}^{n,1}$ such that $Ax = b$
1 $L \leftarrow \texttt{POTF2}(A)$
2 $y \leftarrow L^{-1}b$
3 $x \leftarrow (L^T)^{-1}y$
4 **return** $x$
```

Algorithm 9: Solution of a system of linear equations $Ax = b$ with a symmetric positive definite matrix A using Cholesky decomposition

The calculations on lines 2 and 3 can be performed using the `TRSV` Level 2 BLAS routine.

▶ **Theorem 4.7.** *For a symmetric positive definite matrix $A \in \mathbb{R}^{n,n}$ and a vector $b \in \mathbb{R}^{n,1}$, Algorithm 9 returns a vector $\tilde{x}$ that is the only solution of the equation system $Ax = b$.*

Proof. We know that both L and L^T are nonsingular from Lemma 4.2 and Theorem 1.15. We then have

$$A\tilde{x} = A(L^T)^{-1}y = A(L^T)^{-1}L^{-1}b = \left(LL^T\right)(L^T)^{-1}L^{-1}b = LL^T(L^T)^{-1}L^{-1}b = LL^{-1}b = b.$$

Following Theorem 1.13, A is also nonsingular. By multiplying both sides of $Ax = b$ by A^{-1} from the left, we obtain

$$x = A^{-1}b.$$

Since both A^{-1} and b are determined uniquely, x is also unique. ◀

If A is not positive definite, other decomposition algorithms are used to solve the system of linear equations (typically the LU factorization) [14]. The reason why solution using the Cholesky decomposition algorithm is attempted first is that it performs approximately $\frac{1}{3}n^3$ floating point operations, whereas algorithms for solving systems of linear equations with general matrices require roughly $\frac{2}{3}n^3$ operations [3].

We can show that if Algorithm 9 fails to return a solution x, then the input matrix A was not positive definite, regardless of the algorithm used on line 1:

▶ **Theorem 4.8.** *A square matrix $A \in \mathbb{R}^{n,n}$ has a Cholesky decomposition if and only if it is symmetric positive definite.*

Proof. The converse implication ($\Leftarrow$) has already been proven in Theorem 4.3.

A matrix $A \in \mathbb{R}^{n,n}$ with a Cholesky decomposition $A = LL^T$ is symmetric positive definite following Lemmas 1.22 and 1.24, since L has a non-zero determinant and is therefore nonsingular. ◀

In summary, if the algorithm used on line 1 fails to produce the matrix L, then A is not symmetric positive definite and thus some other algorithm for solving systems of linear equations must be used.

Chapter 5

Numerical libraries for linear algebra

In this chapter, we will present existing libraries for dense numerical linear algebra that are used in this thesis for performance comparison.

5.1 LAPACK

We will start with the oldest library described in this chapter – LAPACK. The content of this section closely follows [16].

In 1987, six authors, three of whom were co-authors of the original articles that introduced the Level 1, Level 2 and Level 3 BLAS routine sets (articles [9], [10] and [11]), pledged to design and develop a new numerical library that would serve as an alternative to the LINPACK and EISPACK libraries [16]. Consequently, the scope of this new library would mainly consist of the following:

- Routines for solving systems of linear equations
- Routines for eigenvalue problems
- Routines for linear least squares problems
- Routines for performing associated matrix factorizations
- Routines for estimating condition numbers for the above problems

The main reason for this proposal was the insufficient performance of the LINPACK and EISPACK libraries, which were constructed only using Level 1 BLAS routines. Using optimized implementations of those routines, LINPACK and EISPACK could perform well on scalar processors, but these libraries were not very well suited for vector processing machines. Thus, many of the algorithms used in LINPACK and EISPACK were restructured to employ the Level 2 and Level 3 BLAS so that users could gain increased performance by linking against optimized BLAS libraries.

Many new algorithms (especially the ones computing matrix factorizations) took a similar approach to utilize Level 3 BLAS routines – partition the matrices into blocks and then express the computation using basic matrix operations on those blocks. Those algorithms were called **blocked algorithms**; an example is given in Section 4.2. The basic block operations often involved the same computation being performed on one of the blocks (such as the `POTF2` routine used in Algorithm 7), which required the **unblocked algorithm** to be implemented in the library, too. The unblocked algorithms used Level 2 BLAS routines more frequently.

Like the Reference BLAS libraries, LAPACK was implemented in Fortran 77. Many routines also followed the naming convention showcased in Example 3.1, with the exception that the operation type could exceed the two letter limit. The routines were divided into two groups – top-level routines that were intended to solve complete problems and lower-level routines which represented the individual steps of those solutions.

Apart from improving the performance of LAPACK and EISPACK routines, the library also intended to increase portability (mainly through making calls to BLAS) and serve as a benchmark to evaluate the performance of various supercomputers.

5.2 Intel OneApi MKL

In an effort to optimize performance with respect to the given CPU architecture and its features, hardware vendors often provide their own implementations of numerical routines. An example of this phenomenon is the Math Kernel Library (MKL), which is freely provided by Intel as a part of their OneApi interface. The rest of this section follows [17]. The functionalities of MKL include:

- Dense linear algebra routines – all BLAS and selected LAPACK routines
- Selected sparse BLAS routines
- Fast Fourier Transform routines
- Random number generators
- Selected Vector Math routines

MKL is parallelized using the Intel Threading Building Blocks (TBB) runtime model but also supports OpenMP GPU offloading. Programs that use the MKL are suggested to be compiled using Intel's `icc` compiler (which is also part of the OneApi interface), though other compilers, such as GNU `gcc`, are supported as well.

5.3 OpenBLAS

OpenBLAS is an open-source numerical library which supports many different CPU architectures, such as x86-64, AArch64 and RISC-V. As a successor to GotoBLAS, it was initially strictly a BLAS library. Today, it implements many LAPACK routines as well (including the `POTRF` Cholesky decomposition routine), though the optimization efforts are mainly focused on the BLAS routines. The leading maintainer of the project is Dr Zhang Xianyi [18].

For many BLAS routines, the authors of the project use a *“template-based optimization framework”* [19] called AUGEM. The framework allows them to select *“the best configurations based on performance feedback of the optimized code”* [19].

5.4 AMD Optimizing CPU Libraries

This section follows [20].

As Intel MKL became more focused on Intel CPUs, AMD introduced their own set of numerical libraries named AMD Optimizing CPU Libraries (AOCL). Unlike Intel MKL, AOCL is fully open-source. Most of the libraries that the AOCL set contains are forks of other open-source numerical libraries, such as:

- AOCL-BLIS – an optimized BLAS library (a fork of BLIS)
- AOCL-libFLAME – a parallel numerical library providing the functionality of LAPACK (a fork of libFLAME)
- AOCL-ScaLAPACK – a numerical library for *"parallel distributed memory machines"* [20]

AOCL also implements routines for sparse linear algebra, Fast Fourier Transform, cryptographically secure random and pseudo-random number generation, cryptography, data compression, optimized math, string and memory functions. All of the AOCL libraries are available on GitHub.

5.5 Arm performance libraries

Arm developed a set of free-to-use (though not open-source) libraries named the Arm Performance Libraries. It is optimized for Arm CPUs, and it implements the following functionalities [21]:

- The BLAS routines
- Higher level numerical linear algebra routines using the LAPACK interface
- Routines for Fast Fourier Transform using the FFTW[1] interface
- Sparse numerical linear algebra functionalities
- Elementary math functions optimized for the AArch64 architecture as defined in `math.h`
- String and memory functions optimized for the AArch64 architecture as defined in `string.h`

Many of the numerical routines are parallelized using OpenMP [21].

[1]Fastest Fourier Transform in the West, an open-source library implementing Fast Fourier Transform routines.

Chapter 6

Task-based runtime systems

In scientific computations, programs are rarely ever sequential. Most often, calculations are performed on several CPU cores, multiple different CPUs or even multiple processing units with different architectures. As such, numerical programs have to be **parallelized**.

One way to parallelize existing programs is to utilize standard threading libraries, such as the `pthread` library on UNIX systems. The disadvantage to this approach is that the programmer is solely responsible for managing dependencies between sections of code (e.g., making sure that a function is run only after another function is already finished). Such code is thus very prone to race conditions – program errors which arise when the same memory section is either written to by multiple threads simultaneously or read by one or more threads while it is written to by another thread. These libraries usually provide tools for synchronization, such as mutual exclusion locks, semaphores, atomic data types or barriers, which can theoretically be used even for programs with high amounts of dependencies. That approach, however, leads to code that is difficult to understand and therefore difficult to debug.

For many programs with large amounts of dependencies, we can split the code into sections and describe how each of those code sections accesses shared memory areas (whether it reads the memory, writes to the memory, or both) and what other code sections need to be finished before this code section can be run. An object that holds this information about a particular code section is called a **task**. The memory sections that the code section of a certain task accesses, along with the corresponding data access modes, are called its **data dependencies** (or **memory dependencies**) and the tasks which need to be executed before a certain task is run are called its **task dependencies** [22].

The programmer can either deduce the task dependencies from the data dependencies and the sequential code themselves and express them explicitly, or they can describe the data dependencies to a **task-based runtime system**, which then implicitly derives the task dependencies from the data dependencies and the order that the tasks were added in [22].

6.1 Task-based Cholesky decomposition algorithm

The parallel Algorithm 8 can be further improved using task-based programming. For example, if the value $A_{(j+1)(j+1)}$ is already updated on line 6 but some of the other values are still being recalculated, the algorithm waits for those computations to finish. The computation of the value $L_{(j+1)(j+1)}$ could, however, be launched on the core that $A_{(j+1)(j+1)}$ was recalculated on. As such, we can achieve better parallelization by describing the data dependencies and letting the runtime system implicitly construct the task dependencies:

```
Data: Symmetric positive definite matrix $A \in \mathbb{R}^{n,n}$
Result: Lower triangular matrix $L$ with positive diagonal entries such that $A = LL^T$
1  for $j \leftarrow 1; j \leq nb; j \leftarrow j+1$ do
2      Insert task $A_{jj} \leftarrow$ POTF2($A_{jj}$) with data dependency $A_{jj}$ (R + W)
3      for $i \leftarrow j+1; i \leq nb; i \leftarrow i+1$ do
4          Insert task $A_{ij} \leftarrow A_{ij} \cdot \left((A_{jj})^T\right)^{-1}$ with data deps. $L_{jj}$ (R) and $A_{ij}$ (R + W)
5      end
6      for $k \leftarrow j+1; k \leq nb; k \leftarrow k+1$ do
7          Insert task $A_{kk} \leftarrow A_{kk} - A_{kj} \cdot (A_{kj})^T$ with data deps. $A_{kj}$ (R) and $A_{kk}$ (R + W)
8          for $i \leftarrow k+1; i \leq nb; i \leftarrow i+1$ do
9              Insert task $A_{ik} \leftarrow A_{ik} - A_{ij} \cdot (A_{kj})^T$ with data dependencies $A_{ij}$, $A_{kj}$ (R) and
                 $A_{ik}$ (R + W)
10         end
11     end
12 end
13 return the lower triangular part of $A$
```

Algorithm 10: Task-based Cholesky decomposition (variant of [15], Algorithm 3)
This algorithm was derived from the in-place variant of Algorithm 7.

6.2 Task graphs

By considering tasks as vertices of an oriented graph and the task dependencies as edges, we can characterize the task-based program by a **task graph**. Task graphs can also be called **directed acyclic graphs** or **DAGs** for short, meaning graphs that contain no cycles. In the context of task-based runtime systems, we define a cycle in a task graph of a program P as a finite sequence of tasks $a_1, a_2, \dots, a_n$ in P where a_i is a task dependency of a_{i+1} for all $i \in \{1, \dots, n-1\}$ and a_n is a task dependency of a_1. The following theorem shows that the task graph of a finite program is always acyclic:

▶ **Theorem 6.1.** *Let P be a task-based program. If all tasks of P finish in a finite amount of time, then the task graph of P is acyclic.*

Proof. We will prove the contraposition of the statement.

Suppose that a task-based program P contains a cycle, or equivalently, there are tasks $a_1, a_2, \dots, a_n$ in P such that a_i is a task dependency of a_{i+1} for all $i \in \{1, \dots, n-1\}$ and a_n is a task dependency of a_1. Using induction, we can prove that the task a_1 has to be finished before the runtime system executes the task a_n:

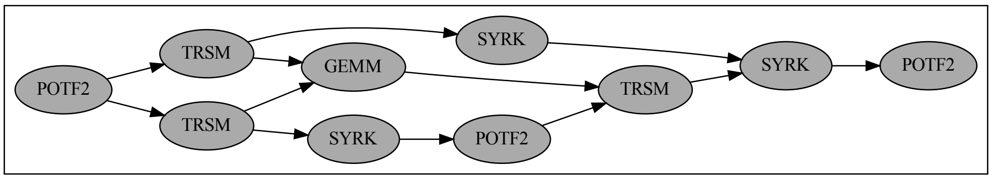
Base step a_1 is a task dependency of a_2, so a_1 has to be finished before the system starts a_2

Inductive step Suppose that a_i can be started only after a_1 is finished for any $i \in \{2, \dots, n-1\}$. Due to a_i being a task dependency of a_{i+1}, a_{i+1} can be run only after a_i is finished, which, following our supposition, is only after a_1 is finished.

We have thus established using the induction principle that a_n can only be executed after a_1 is finished. However, since a_n is a task dependency of a_1, a_1 can only be run after a_n is finished. Running either a_1 or a_n (or both at once) would break at least one of those two conditions, therefore a_1 and a_n will not be launched at all. Hence, P never finishes. ◀

To provide an example, we present the task graph of Algorithm 10 for an input matrix with 3×3 blocks:

▶ **Example 6.2.** Task graph of Algorithm 10 for an input matrix with 3×3 blocks.



6.3 Task states and scheduling

This section follows [22].

At any moment when running a task-based program, each task can be in exactly one of the following states:

Submitted One or more task dependencies of this task are not finished yet

Ready All of the task dependencies are finished, but the task has not been executed yet

Active The task is being executed at the moment

Completed The execution of the task has been finished.

The runtime system stores a list of all tasks that are ready called a **ready pool**. Once an active task a is complete, the system iterates through all submitted tasks that have a as a dependency and marks the dependency as satisfied – this may be implemented as simply removing the dependency from the task dependency list. If any of those submitted tasks have no unsatisfied dependencies, they are moved to the ready pool. Then, the system selects a task from the ready pool and executes it on a core that is currently free. The component of the runtime system that selects tasks for execution is called the **scheduler**. Most schedulers allow programmers to influence the task selection process by setting **task priorities**.

6.4 Task granularity

When designing parallel programs, it is crucial for the programmer to choose the right granularity of the tasks – i.e., how many tasks will the problem be split into and how long will those tasks take to execute.

If we have a small amount of tasks that take long to execute, the CPU may not be utilizing all of its cores at all times, and so the program might take longer to finish. On the contrary, if we have a large amount of short tasks, it is going to be more difficult (and more time consuming) for the runtime system to manage all of those tasks, their dependencies and priorities. So while the workload may be better distributed among the CPU cores, the system is going to require more time to manage the tasks, so the overall time may increase as well [22].

To address these issues, we usually run the task-based program for various different task granularities and select the case with the lowest overall execution time. In our case of the blocked Cholesky decomposition algorithm, task granularity is controlled by the block size – smaller blocks lead to a higher amount of smaller tasks and vice versa.

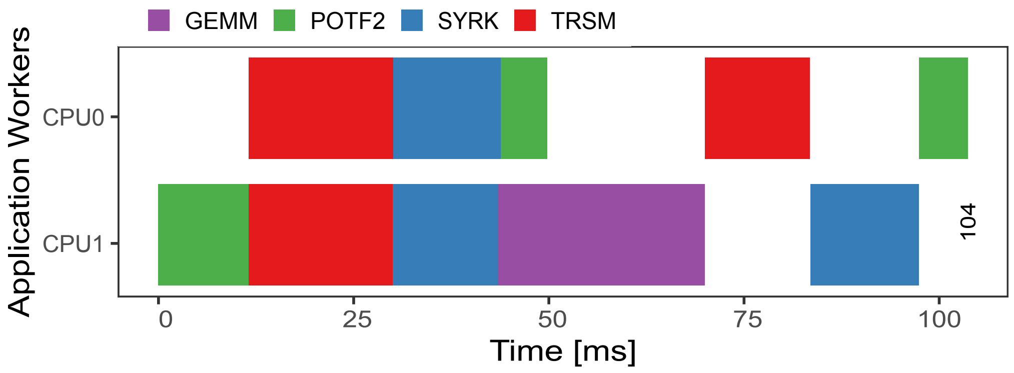
6.5 Trace

A trace of a runtime program is a visualization of the individual tasks' lifecycles. It contains a horizontal timeline for each CPU core, and each of the colored areas on one of the timelines represents the time that a particular task was active. Tasks may be colored according to their name.

A trace may help us spot unwanted dependencies or tune the task granularity:

- If there is a long time period at the end of the program where not all cores are being used, it may be better to split the program into more granular tasks
- If there are many tasks that are switching rapidly, it may be better to join the tasks into less granular ones

▶ **Example 6.3.** A trace of Algorithm 10 for an input matrix with 3 × 3 blocks on a machine with two cores.



6.6 OpenMP

OpenMP is an interface for programming shared-memory parallel systems published in 1997 [23]. Recently, extensions have been released to also provide an interface for GPU computing functionalities. OpenMP can be used in C/C++ as a set of preprocessor pragmas, all of which have the form `#pragma omp <construct> [options]`. OpenMP also has a Fortran interface, but the syntax differs from the C/C++ interface.

In task-based programming, the most important OpenMP construct is the `task` construct. A `task` construct followed by a brace-enclosed code block creates a task, which is either executed immediately, or delayed until it is scheduled for execution. Data dependencies of the task can be specified using the `depend` clause and the priority of a task can be set using the `priority` clause. The variable access rules can be specified using the following clauses [24]:

private Creates a new variable for each thread and ignores the variable with the same name in the outer scope, if one exists.

firstprivate Creates a new variable for each thread with the same value as the eponymous variable in the outer scope.

shared Shares the variable in the outer scope across all threads.

Another important construct is `barrier`, which waits for all previously created tasks to finish. In case we want to wait for child tasks only (i.e., tasks that were created inside another task), the `taskwait` construct can be used [24].

OpenMP also offers many constructs outside of the task-based programming domain. A code block can be run in the fork-join model by multiple threads using the `parallel` construct, though there is no option to specify the dependencies. The `single` construct may then be used to tell OpenMP to execute a code block inside a `parallel` region by only one thread. If we want to execute a code block by a single thread at a time, we can use the `critical` construct. A for loop in C/C++ can be parallelized using a `loop` construct [24].

As OpenMP is an interface, it cannot be distributed as a standard C/C++ or Fortran library, but compiler vendors may choose to implement the OpenMP preprocessor pragmas in their compiler. Examples of widely used compilers that implement the OpenMP interface are the `gcc` compiler developed by the GNU initiative, the `icc` compiler by Intel Corporation and the `clang` compiler, which is a part of the LLVM project [25].

6.7 StarPU

StarPU is a task-based runtime system developed at the University of Bordeaux and released in 2009. The main motivation behind its development was to design a high-level, easy to use, portable runtime system for heterogeneous architecture systems. Reference [26] states that the most successful interface for programming heterogeneous architectures prior to the development of StarPU was OpenCL, but it was too low level to be considered a runtime system.

Data used by StarPU tasks has to be registered in the system using a function, which then returns a **handle**. A handle is a structure that contains information about a data block that may be specified as a data dependency of a task. Data blocks have different types in StarPU, for example, a **vector** data block represents a contiguous array of elements and the **matrix** data block represents a matrix stored in the row major full storage format (see Subsection 2.2.1.1). There are also data block types for some of the sparse storage formats described in Section 2.1 [27]. The ability to run tasks across devices with different architectures is accomplished by using **codelets**, which can be thought of as a structure similar to tasks, with the difference that multiple implementations can be provided for the same task – usually one implementation per device architecture. That allows programmers to utilize different libraries or even different programming languages for each architecture that the task may be run on. Every codelet also specifies the data dependencies, that is, the data blocks that the code accesses along with the corresponding data access modes (read, write or read + write). StarPU also manages the data transfers between devices in a way that all of the data blocks that a task is dependent on are present in the memory of the device that runs the task.

StarPU has an interface for the C/C++, Java, Python and Fortran programming languages. Using the KStar source-to-source compiler, programmers can seamlessly translate a program that uses OpenMP pragmas to a program that uses StarPU [28].

Unlike OpenMP, StarPU is also well suited for distributed memory systems, as it integrates very well with the Message Passing Interface (MPI). Since version 1.2, it also supports data offloading to disks. Several tools exist to visualize the task graphs of StarPU programs (Temanejo, ViTe) or to generate a trace (the FxT library) [28].

Chapter 7

Implementation

The blocked algorithm for Cholesky factorization was implemented in the C language, as both OpenMP and StarPU offer a C interface and the language is commonly used in many modern numerical libraries (e.g., libFLAME, BLIS and OpenBLAS). The implementation was tested on Linux systems with the `gcc` and `gfortran` compilers only (version 10.2.0 for both), although it may work on other UNIX-like systems with other compilers as well.

When compiled, the following artifacts are produced by default:

- A dynamic library – `libcholesky.so`
- A C/C++ header file for the library – `cholesky.h`
- Three test binaries – `blas_test`, `potf2_test`, `cholesky_test`
- A benchmarking utility – `cholesky_benchmark`

A static library build may be enabled in the configuration script as well (see Section 7.6).

The code is freely available on GitLab under a 2-clause BSD license.

7.1 Implementation of BLAS routines

The `GEMM`, `SYRK` and `TRSM` routines are implemented in the `src` directory. Only the double-precision real number variants are implemented, though the code would not be difficult to edit in order to implement the single-precision number routines `SGEMM`, `SSYRK`, and `STRSM` or the complex number routines prefixed with `Z` or `C`.

The implementations closely resemble Algorithms 2, 3 and 4, though the C implementations are split in four branches (for `SYRK`) or eight branches (for `GEMM` and `TRSM`) according to the different values of the `TRANS`, `TRANSA`, `TRANSB`, `UPLO` and `SIDE` parameters.

The interface to the BLAS routines is highly inspired by the original Fortran interface (see Section 3.1.3), but with the following changes:

- The function names consist of non-capital letters only (`dgemm`, `dsyrk` and `dtrsm`)
- The `TRANS`, `TRANSA`, `TRANSB`, `UPLO` and `SIDE` parameters are passed by value, not reference (the parameters are of type `char`, not `char*`, but the meaning of the values remains the same)
- The `ALPHA` and `BETA` parameters are passed by value, not reference (these parameters have the `double` type as opposed to `double*`)

- All matrix size and leading dimension parameters are passed by value, not reference (these parameters have the `int` type as opposed to `int*`)
- The error code is not passed by the `INFO` output variable, but by the return value

Similarly to the Fortran BLAS interface, the routines assume all matrices in column major full storage.

7.1.1 Cache access optimization

The only optimizations performed were the reordering of the nested for loops in order to avoid unnecessary cache misses, i.e., accesses to memory addresses that are not loaded in the cache memory. As cache memories are known to perform better when the memory is accessed sequentially, we try to reorder the loops in a way that minimizes the offsets between subsequent accesses.

For instance, in the `DGEMM` routine, the code in listing 7.1 is not very efficient, since most subsequent memory accesses in matrices A and B have an offset of `ldc` entries, which leads to large amounts of cache misses:

Code listing 7.1 Unoptimized code from the `DGEMM` implementation.

```
for (int i = 0; i < M; i++) {
  for (int k = 0; k < K; k++) {
    ...
    C[i + j * ldc] += alpha * A[i + k * ldc] * B[j + k * ldc];
    ...
  }
}
```

To the contrary, the code in listing 7.2 accesses the entries of A and C sequentially in an ascending order, and is therefore more efficient due to a lower amount of cache misses:

Code listing 7.2 Cache access optimized code from the `DGEMM` implementation.

```
for (int k = 0; k < K; k++) {
  for (int i = 0; i < M; i++) {
    ...
    C[i + j * ldc] += alpha * A[i + k * ldc] * B[j + k * ldc];
    ...
  }
}
```

The only branches that are optimized for cache accesses are the ones used in Algorithm 7. Those are:

- `TRANSA` = `N` and `TRANSB` = `T` for `GEMM`
- `UPLO` = `L` and `TRANS` = `N` for `SYRK`
- `SIDE` = `R`, `UPLO` = `L` and `TRANSA` = `T` for `TRSM`

7.1.2 Further possible optimizations

The BLAS routine implementations could be further optimized by utilizing SIMD vector register instructions, such as SSE or AVX instructions on certain Intel processors. These register have a larger size than standard registers, but allow the same operations to be performed on more parts of the data at once. This allows multiple floating point operations to be performed on multiple floating point numbers during one instruction cycle.

Optimized BLAS routines also often make use of the Fused-Multiply-Add (FMA) instruction, which combines the floating point addition and multiplication operations into one instruction. In combination with SIMD instructions, it can lead to a throughput may times higher than the one of unoptimized routines.

The BLAS routines could themselves be parallelized using a threading library or a runtime system, but that is outside of the scope of this thesis.

It should be noted that we only use optimized BLAS routines from numerical libraries in the final benchmarks.

7.2 Implementation of the Cholesky decomposition routines

The unblocked Cholesky routine is implemented in the `src/potf2` subdirectory using a modified variant of Algorithm 5. The interface is similar to the one of BLAS routines (described in the Section 7.1), with the difference that invalid parameter values are signaled with negative return codes, while positive return codes indicate an input matrix that is not positive definite. The return value then marks the index of the main loop iteration where a negative diagonal element was detected before computing the square root.

The blocked Cholesky decomposition algorithm implementation comprises two routines in the `src/cholesky` subdirectory, where one of them uses OpenMP tasks and the other uses the StarPU runtime system. Apart from the pointer to the memory area where the matrix is stored and the matrix size, the routines also retrieve the block size used for partitioning the matrix and the translation method as parameters.

The Cholesky routines are implemented using the task-based Algorithm 10. In both routines, we assume that the input matrix $A \in \mathbb{R}^{n,n}$ is stored in the column major full storage format with the leading dimension equal to n.

The routines start by partitioning the input matrix into blocks. The information about the blocks is stored in a matrix (the information about a particular block is stored in the `starpu_data_handle_t` structure for the StarPU variant and a custom structure `block_data_t` for the OpenMP variant). According to the value of the `translation_method` parameter, one of the following operations is performed on the matrix:

- The matrix is left as it is (the `NO_TRANSLATION` value)
- The matrix is translated into the block storage format sequentially
  (the `SEQUENTIAL_TRANSLATION` value)
- The matrix is translated into the block storage format using StarPU or OpenMP where the translation of each block is inserted as a task (the `PARALLEL_TRANSLATION` value)

At the end, the computed matrix is translated back into the column major format if necessary (again, in sequential or in parallel, according to the value of the `translation_method` parameter) and all memory blocks associated with matrices created in the routine are freed.

7.3 Testing of BLAS routines

The `DGEMM`, `DSYRK` and `DTRSM` routines were tested using a modified version of the `DBLAT3` program – a testing program for double-precision Level 3 BLAS routines from the reference BLAS library. Function calls that test other Level 3 BLAS routines were deleted and only the function calls to test routines DPOT1, DPOT3 and DPOT4 were left in, as they test the three aforementioned BLAS routines.

These test routines use the reference DMMCH function. It receives matrices A, B, CT and CC as input parameters and performs a computation of the general matrix product

$$CT \leftarrow \alpha \mathrm{op}(A)\mathrm{op}(B) + \beta CT \tag{7.1}$$

It then computes the maximum error ratio among all of the entries and reports failure if at least one of the results is less than half accurate [29].

Testing DGEMM using DMMCH is straightforward, as both of the routines perform the same computation. The computation that SYRK performs (equation (3.2) or (3.3)) is a special case of equation (7.1), so SYRK can be tested using DMMCH in a straightforward way as well. TRSM performs fundamentally different computations, but after B has been updated according to equations (3.4) or (3.5), the value of the original matrix B can be obtained as $\alpha^{-1}\mathrm{op}(A)B$ or $\alpha^{-1}B\mathrm{op}(A)$, respectively. Both of those formulas are specific cases of (7.1), so TRSM can be tested using the DMMCH routine, too.

7.4 Testing of the Cholesky decomposition routines

The main blocked Cholesky decomposition routine was tested using a modified version of the DCHKPO LAPACK routine rewritten to C. The tests performed in the routine were purposely limited to exclude testing features that are outside of the scope of this thesis. Those are:

- Testing error exits for invalid parameter values
- Testing error exits for input matrices that are not symmetric positive definite
- Testing the UPLO = U variant of the DPOTRF routine (the $A = U^T U$ decomposition)

All of the remaining tests are run for several block sizes.

For a single set of input parameters (matrix size, matrix type, block size, translation method and right-hand side count rhs), an $n \times rhs$ right-hand side matrix is generated and the following tests are performed [29]:

1. Reconstructing $\tilde{A} = LL^T$ and computing the matrix 1-norm $\|A - \tilde{A}\|$
2. Calculating the inverse $\tilde{A}^{-1} = (L^T)^{-1}L^{-1}$ and computing the matrix 1-norm $\|A\tilde{A}^{-1} - I\|$
3. Solving $Ax = b$ for each right hand side b and computing the vector 1-norm $\|A\tilde{x} - b\|$
4. Computing the difference between the solution obtained in the previous test and the true solution to $Ax = b$ (there is only a single true solution, see Theorem 4.7)
5. Iteratively refining the computed solution and computing the error bounds
6. Estimating the condition number of a matrix and comparing the estimate with the true value

The unblocked routine dpotf2 was tested using a stripped down version of the DCHKAA program, where all test routines except for DCHKPO (the routine testing symmetric positive definite matrix operations) were removed. In order to be able to call the xerbla error handling routine from LAPACK with strings of size that is unknown at compile time, a custom helper Fortran routine is provided in the file xerbla_helper.f.

The source code for all tests can be found in the test directory.

7.5 Implementation of the benchmarking utility

The benchmarking utility measures the performance of the blocked Cholesky routine. Its source code is found in the `src/benchmark` directory. It is a Linux command line executable with the following arguments:

- Matrix size: `-m`, `--matrix-size`
- Block size: `-b`, `--block-size`
- Number of threads used: `-t`, `--no-threads`
- The random seed for matrix generation: `-s`, `--seed`
- Number of iterations for each matrix size and block size: `-i`, `--iterations`
- Additional correctness testing: `-C`, `--check-correctness`
- Running the custom implementations only: `-c`, `--custom-only`
- Selection of the translation method: `-T`, `--translation-method`
- Ensuring correct trace generation using the FxT library and StarPU: `-f`, `--fxt-trace`

The `--matrix-size` and `--block-size` parameter values may be specified in the form of `start:end:step` ranges. If so, the benchmarking utility iterates through every matrix size and every block size specified in the ranges.

In every iteration, the utility first generates a symmetric positive definite matrix using Algorithm 1 with the entries of R being random uniform numbers from the interval $\langle 0, 1\rangle$ and $\lambda = 1$.

Then, the custom blocked Cholesky routine is run and the execution time is measured using the `clock_gettime(CLOCK_MONOTONIC, ...)` function from `time.h`. The Gflop/s metric (introduced in Section 8.1) is then calculated and the results are printed. The translation method used is determined by the `--translation-method` argument, the possible values are `no` for not performing scheme translation at all, `sequential` for sequential translation and `parallel` for parallelized translation using OpenMP or StarPU.

If additional correctness testing is enabled, the utility performs a test equivalent to test 1 from the list in Section 7.4 and prints the test ratio and the test result. If the `--custom-only` option is not enabled and if the program has been linked with one of the numerical libraries, the same steps are then taken for the library implementation of the `DPOTRF` routine, which computes the Cholesky factorization.

To make a fair comparison between the library implementation and the custom implementation, we need the driver routines (`DGEMM`, `DSYRK`, `DTRSM` and `DPOTRF`/`DPOTF2`) to be executed sequentially during the custom implementation, because the program is already parallelized by the runtime system and parallelizing the driver routines would lead to very granular "subtasks" with a big overhead. On the other hand, those routines need to be parallelized in the library implementation. We use builtin library functions (usually with the name `<libname>_set_num_threads`) to set the desired number of threads for both implementations.

Note that in order for the AOCL implementation to work properly, the libFLAME library has to be built with the LAPACK2FLAME option enabled. Similarly, the OpenBLAS implementation requires the usage of the `USE_THREAD=1` and `USE_OPENMP=0` options when compiling.

7.6 The configuration script

The implementation contains a configuration script which automatically detects the `gcc` and `gfortran` compilers, StarPU and OpenMP. Using the `--numerical-lib` option, one of the following numerical libraries/library sets can be linked:

- Intel MKL – value `MKL`
- AOCL (both libFLAME and BLIS have to be available) – value `AOCL`
- OpenBLAS – value `OpenBLAS`
- Arm Performance libraries – value `ArmPL`

When using the `--numerical-lib` option with one of these values, the script tries to detect the selected library. If the library files are not present in one of the directories specified in the `LIBRARY_PATH` environment variable or if the include files are not present in a location specified in the `C_INCLUDE_PATH` variable, the script returns an error. If the library is found, its implementations of the driver routines are then used instead of the builtin ones and its `DPOTRF` implementation is run in the benchmarking utility (see Section 7.5). If the option is omitted, the builtin implementations of the driver routines are used and only the custom implementations are run in the benchmarking utility.

The used runtime systems can be specified with the `--use-openmp` and `--use-starpu` options. Both options may be used at once, but at least one of them has to be used, otherwise the script returns an error. The script tries to detect the associated libraries and exits with an error in case of a failure.

The user can also opt to build a static library by using the `--build-static` option. After building, a file named `libcholesky.a` is produced as an artifact.

The directory where the artifacts will be installed can be specified with the `--prefix` option, with the default directory being `/usr/local`.

7.6.1 Makefile

The implementation is built with the GNU `make` utility using the Makefile in the top-level directory.

Available targets include:

all builds all artifacts (default target)

check runs the tests and prints their results

install installs the compiled libraries, header file and benchmarking utility into the directory specified by `--prefix` (or `/usr/local` by default)

uninstall uninstalls the installed artifacts

clean removes the artifacts produced by building

distclean removes the artifacts produced by building and the files generated by the configuration script

The installation procedure thus consists of running the configuration script with the desired options and running `make` and subsequently `make install`. Optionally, `make check` may be run to test the library before installation.

Chapter 8

Evaluation of results

8.1 Measuring performance in GFlop/s

When comparing the performance of several implementations of the same numerical routine, the overall execution time (sometimes called the **wall time**) may be used as a performance metric – the lower the wall time, the better the implementation (assuming that all of the implementations are tested with the same set of inputs). It is, however, impractical to compare implementations of different numerical operations using the execution time, because those operations may perform a different number of floating point operations.

To account for that difference, we may use the **Flop/s** metric, which is computed as follows:

$$\text{Flop/s} = \frac{\text{floating point operations required to perform the numerical computation}}{\text{wall time}}$$

Note that the number of floating point operations used in the numerator does not depend on the algorithm used for the computation. Instead, we use the number of floating point operations that have to be carried out in order to perform the computation – i.e., the number of operations of the best known algorithm for the computation. For instance, we use $\frac{1}{3}n^3 + \frac{1}{2}n^2 + \frac{1}{6}n$ for computations of the Cholesky decomposition, where n is the matrix size (see Subsection 4.1.1).

Using this metric, we may compare implementations of different numerical computations, where higher values mean better performance. The values of Flop/s obtained on modern multi-core processors are usually in the order of billions, which is why the **Gflop/s** metric calculated as

$$\text{Gflop/s} = \frac{\text{Flop/s}}{10^9}$$

is used more often.

8.2 Performance evaluation

In this section, we examine the performance of the main Cholesky decomposition routine using the benchmarking utility described in Section 7.5. All plots presented in this section were generated using the ggplot2 library of the R programming language. Each data point in a plot represents an average value across 5 iterations. All tests apart from the ones in Subsection 8.2.4 have been run on all available CPU cores. Only the custom implementations using StarPU are presented in this chapter, as the OpenMP custom implementations showed inferior performance.

8.2.1 Used hardware

Performance of the Cholesky decomposition routine was evaluated on three types of multicore compute nodes with different CPU architectures. All of the nodes are part of the IT4Innovations National Supercomputing Center, which belongs to the Technical University of Ostrava.

The used compute nodes have the following specifications [30]:

1. Node of the Karolina supercomputer (without accelerators)
   - Processors: 2×AMD EPYC 7H12
   - Processor architecture: x86-64
   - Processor cores: 64 per processor (128 in total)
   - CPU frequency: 2.6GHz
   - Instruction set extensions: Streaming SIMD Extensions 4.2 (SSE4.2), Advanced Vector Extensions 2 (AVX2)
   - Theoretical peak performance: 5324.8 Gflop/s
2. Node of the Barbora supercomputer (without accelerators)
   - Processors: 2×Intel Cascade Lake 6240
   - Processor architecture: x86-64
   - Processor cores: 18 per processor (36 in total)
   - CPU frequency: 2.6GHz
   - Instruction set extensions: SSE4.2, AVX2
   - Theoretical peak performance: 2995.2 Gflop/s
3. Complementary systems partition 1 node (Arm node)
   - Processors: 1×Fujitsu A64FX
   - Processor architecture: ARMv8.2-A
   - Processor cores: 48
   - CPU frequency: 2GHz
   - Instruction set extensions: Scalable Vector Extension (SVE)
   - Theoretical peak performance: 3072 Gflop/s (according to [31])

8.2.2 Block size benchmarking results

Figure 8.1 shows the performance of the custom blocked Cholesky decomposition routine for different block sizes. All results were generated using the StarPU version of the routine on all 128 cores of the Karolina compute node with 10000×10000 input matrices. Note that the Arm Performance Libraries were left out, as they are only available on Arm-based processors [21].

Figure 8.2 examines the relationship between performance and block size for the custom implementations using Arm Performance Libraries and OpenBLAS on the A64FX CPU, also with input matrices of size 10000×10000.

Both figures were generated with the `PARALLEL_TRANSLATION` translation method. The optimal block size is thus influenced by line sizes of various level cache memories on the processor, among other factors.

The plots show the effects described in Section 6.4 – large block sizes lead to a low number of long tasks, which means less parallelization near the end (sometimes also the start) of the program. On the other hand, small block sizes lead to many short tasks, which increases the overhead and, as a result, the overall execution time. We can see from Figure 8.1 that for this matrix size, the tasks hit the right level of granularity for block sizes approximately between 400 and 600.

We will now focus on the implementation using MKL driver routines (the orange line in Figure 8.1). The optimal block size for 10000×10000 matrices will likely be smaller than the optimal block size for 20000×20000 matrices, as the overall number of blocks is proportional to the square of the matrix size (assuming a fixed block size). In other words, smaller block sizes tend to yield better performance for smaller matrices and larger block sizes usually perform better on larger matrices. This phenomenon can be demonstrated on Figure 8.3, where we compare the performance of custom implementation using StarPU and MKL driver routines across varying matrix sizes (along the x axis) for different block sizes (represented by the colored lines).

The effect that varying task granularities have on performance can also be shown on the traces. The trace in Figure 8.4 was generated on one of the Karolina nodes using StarPU and MKL with a 10000×10000 matrix with 200×200 blocks, limited to 36 CPU cores for readability. This combination of input parameters leads to many short, rapidly changing tasks. We can compare this behavior with the trace in Figure 8.5, which was generated with the same matrix size and core count, but with 800×800 blocks. As we can see, the individual tasks have considerably longer execution times, which causes the underutilization of CPU cores towards the end of the program.

Figure 8.1 Performance vs. block size plot for different custom implementations using StarPU and routines from several numerical libraries. The performance was measured on a Karolina compute node with 10000×10000 input matrices. The `PARALLEL_TRANSLATION` translation method was used.

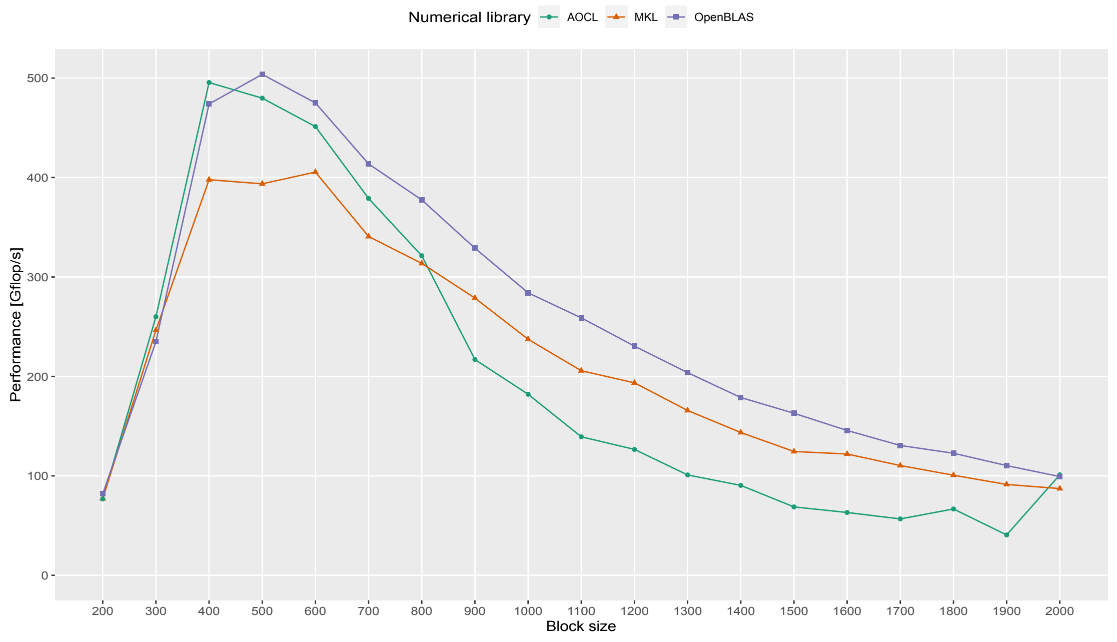


Figure 8.2 Performance vs. block size plot for different custom implementations using StarPU and routines from several numerical libraries. The performance was measured on the Arm node with 10000 × 10000 input matrices. The `PARALLEL_TRANSLATION` translation method was used.

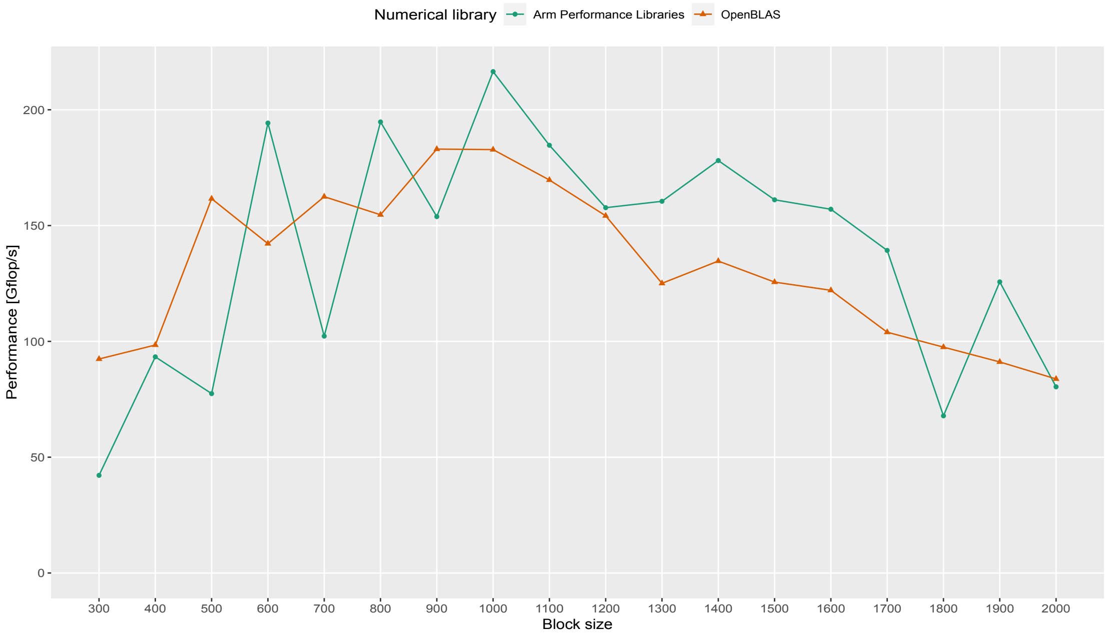


Figure 8.3 Performance vs. matrix size plot for the custom implementation using StarPU and MKL routines. The performance was measured on a Karolina compute node. The `NO_TRANSLATION` translation method was used.

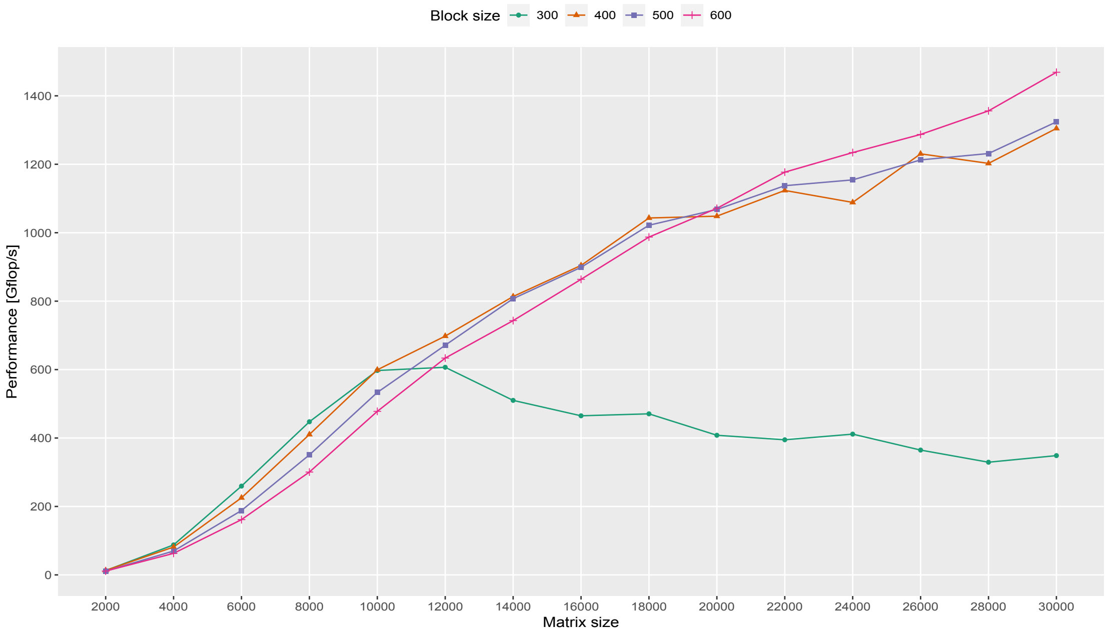


Figure 8.4 Trace of the custom implementation using StarPU and MKL routines generated on a Karolina compute node limited to 36 cores with a 10000 × 10000 input matrix and blocks of size 200 × 200. The `NO_TRANSLATION` translation method was used.

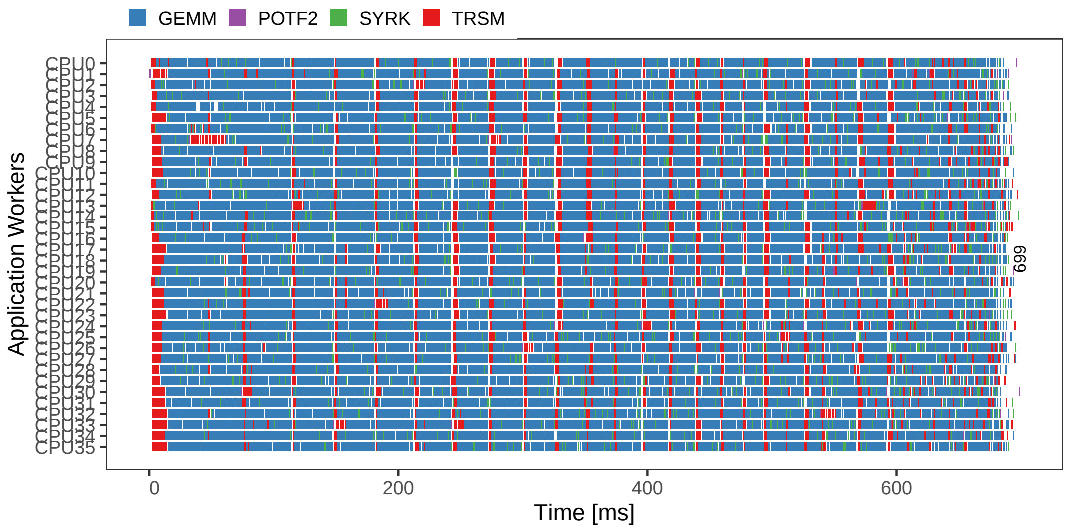
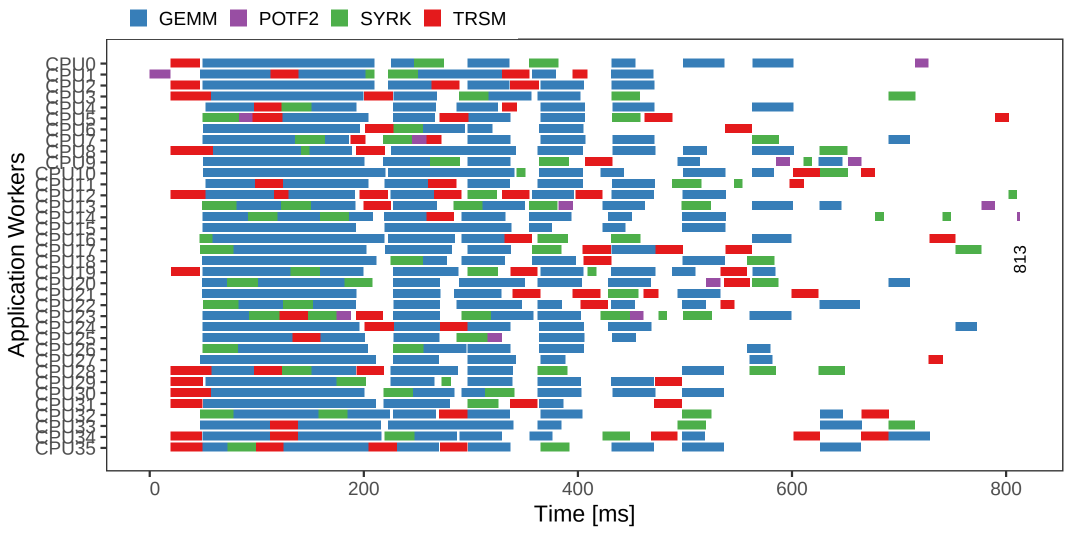


Figure 8.5 Trace of the custom implementation using StarPU and MKL routines generated on a Karolina compute node limited to 36 cores with a 10000 × 10000 input matrix and blocks of size 800 × 800. The `NO_TRANSLATION` translation method was used.



8.2.3 Performance benchmarking results

In this section, we will compare the performance of individual implementations. Figure 8.6 shows the performance on the Karolina compute nodes (with block sizes set to 500 for all three custom implementations) and Figure 8.7 presents the same comparison of the Barbora compute nodes, also with 500 × 500 blocks.

In these benchmarks, we compare our StarPU implementation with calls to optimized driver routines used by tasks with the parallel Cholesky routines from the same libraries.

We can see that on both processors with the x86-64 architecture, the custom implementation has the best results when linked with MKL routines, which is consistent with the fact that the MKL implementation of `DPOTRF` yields the overall best performance on both processors. The custom implementation reaches more than 95% of the performance of MKL `DPOTRF` for some block sizes and on the Karolina node, it seems to perform well even on smaller matrices.

On the Complementary systems' A64FX processor, the custom implementation using Arm Performance Libraries driver routines shows a competitive performance for larger matrices, even surpassing the Arm Performance Libraries implementation for 20000×20000 matrices. The custom implementation does not yield a good performance for smaller matrices, but the performance may be improved by using a smaller block size.

An interesting observation is that the maximum measured performances (in Gflop/s) are noticeably lower than the theoretical peak performances on all three compute nodes. On the Barbora and Karolina nodes, this might be partially explained by the relatively high-latency communication between the two CPUs that make up the compute node, as examined in Section 8.2.4.

Figure 8.9 compares the performance of the three different translation methods on a Karolina compute node, using the StarPU custom implementation with MKL driver routines. It shows that the presumed performance gain from localizing the blocks by scheme translation (which may lead to a lower amount of cache misses with the use of an appropriate block size) has been outweighed by the time it takes to perform the translation.

Figure 8.6 Performance vs. matrix size plot for several custom and library implementations. The performance was measured on a Karolina compute node with blocks of size 500×500. The `NO_TRANSLATION` translation method was used.

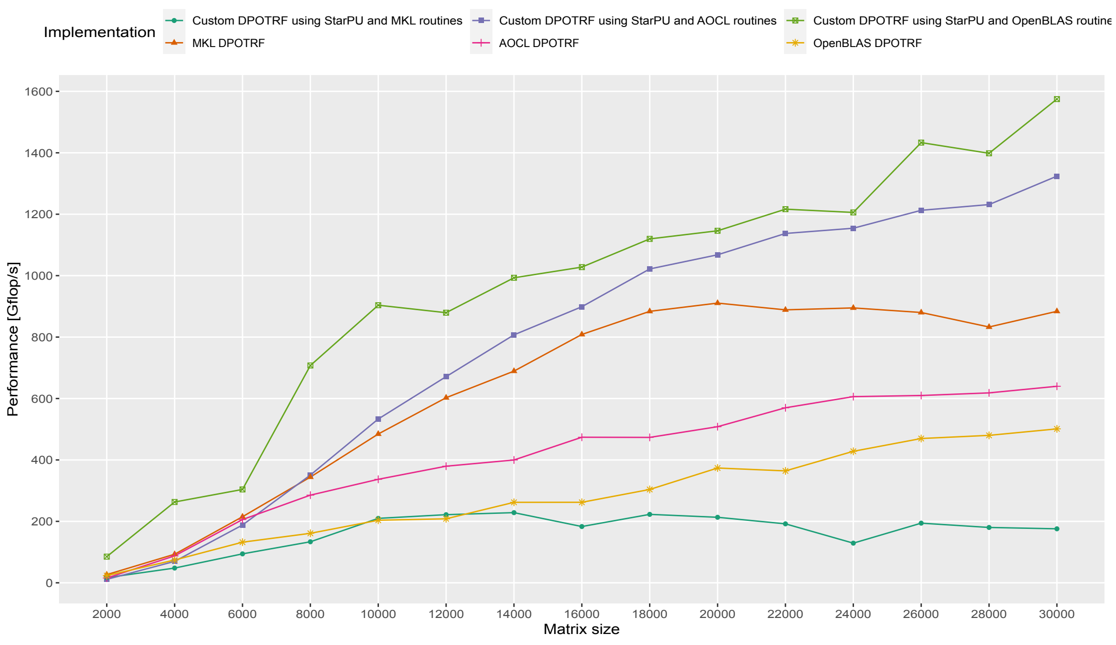


Figure 8.7 Performance vs. matrix size plot for several custom and library implementations. The performance was measured on a Barbora compute node with blocks of size 500 × 500. The `NO_TRANSLATION` translation method was used.

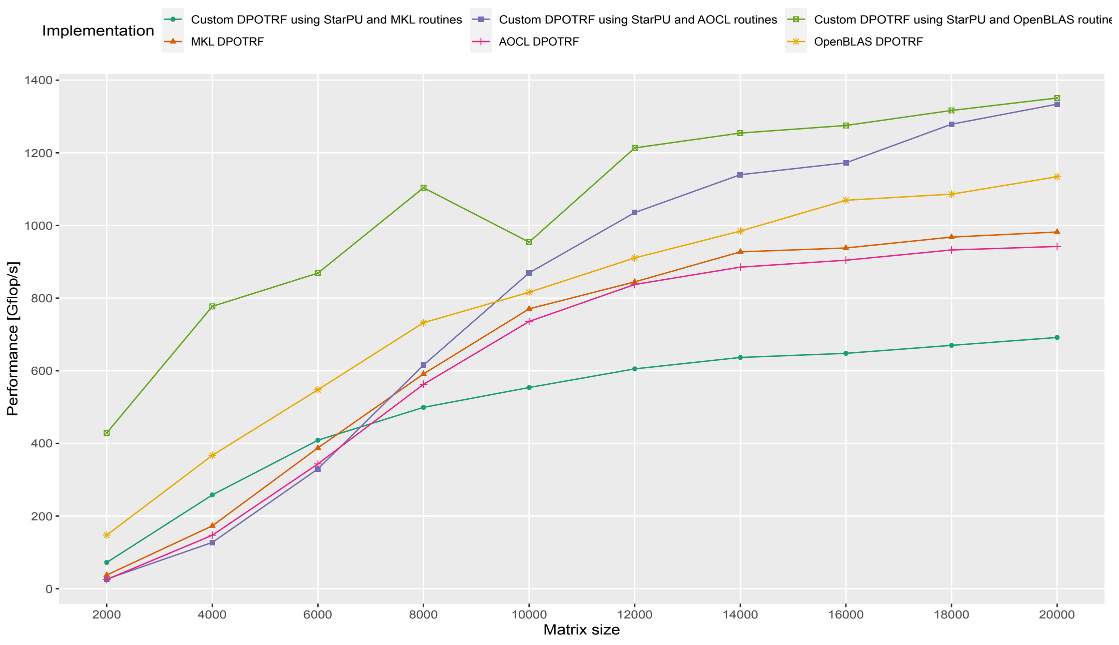


Figure 8.8 Performance vs. matrix size plot for several custom and library implementations. The performance was measured on the Arm node with blocks of size 1000×1000. The `NO_TRANSLATION` translation method was used.

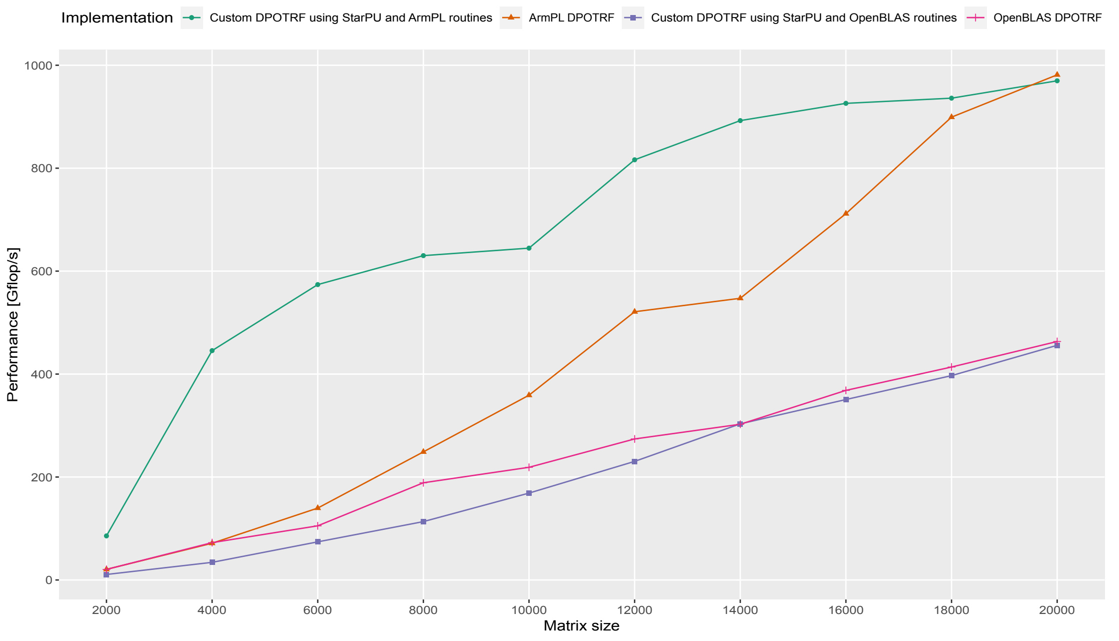
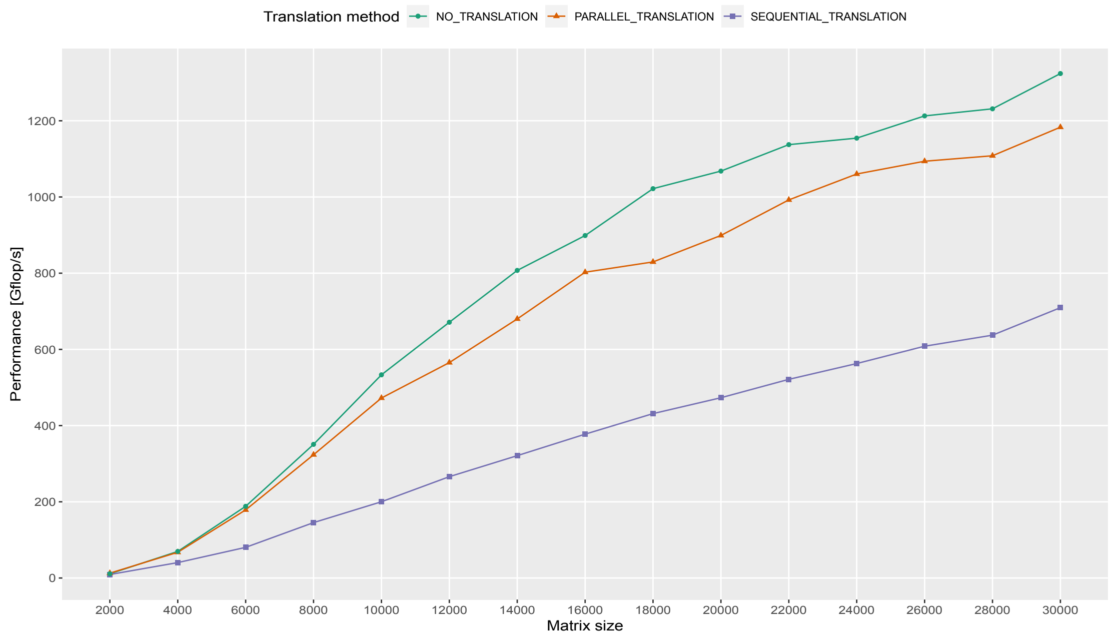


Figure 8.9 Performance vs. matrix size plot for all translation methods for the custom implementation using StarPU MKL routines. The performance was measured on a Karolina compute node with blocks of size 500 × 500.

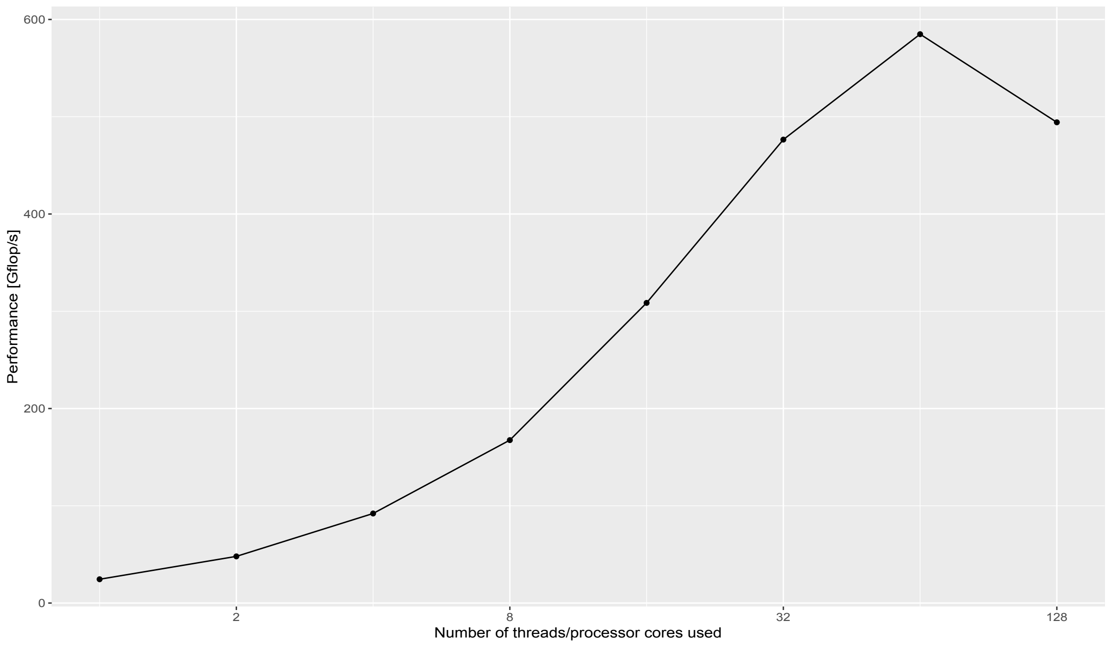


8.2.4 Parallelization benchmarking results

Figure 8.10 shows the relationship between performance and the number of processor cores that we allow the runtime system to use (which is equivalent to the number of threads). Note that the x axis of the figure is in logarithmic scale.

We can notice that the performance improves with a higher number of cores, up until the 64 core threshold. That number is precisely the number of cores of one of the two AMD EPYC 7H12 processors which make up the compute node, so the slowdown from 64 to 128 cores might be explained by the need for the CPUs to communicate with each other. Inter-processor communication naturally has a higher latency than communication between cores on the same CPU.

Figure 8.10 Performance vs. matrix size plot for the custom implementation using StarPU and MKL routines. The performance was measured on a Karolina compute node with 10000×10000 input matrices and blocks of size 500×500. The `PARALLEL_TRANSLATION` translation method was used.



Conclusions

The goal of this thesis was to implement the blocked Cholesky decomposition algorithm using a task-based runtime system, test the implementation and evaluate its performance compared to other well known numerical libraries for linear algebra. Along the main algorithm, the implementation also consisted of three Level 3 BLAS routines (`DGEMM`, `DSYRK` and `DTRSM`) and the low-level LAPACK routine `POTF2`, which implements the unblocked algorithm for Cholesky decomposition.

In the implementation, users are able to select the utilized runtime systems (OpenMP and/or StarPU) via a configuration script. They can then also select a numerical library that provides the optimized implementations for the four subroutines mentioned above. In case that the user does not have access to any of the libraries or they choose not to use them, they can also opt for the algorithm to utilize the builtin implementations of those subroutines. They are, however, not optimized, and their performance will thus not compare well with their optimized counterparts.

All of the variants for both runtime systems and all numerical libraries were tested using slightly modified LAPACK test routines.

The performance of the implementation was evaluated in comparison to the Arm Performance Libraries, MKL, AOCL and OpenBLAS libraries/library sets on three different compute nodes – two of them with x86-64 processors (AMD EPYC 7H12 and Intel Cascade Lake 6240) and the third one with an ARMv8 processor (Fujitsu A64FX). On both of the x86-64 processors, the StarPU implementation performs best with MKL driver routines, where it reaches more than 95% of MKL's performance for certain matrix sizes. On the A64FX processor, the StarPU implementation using driver routines from Arm Performance Libraries performs well for large matrices, even outperforming the Arm Performance Libraries `DPOTRF` implementation for one specific matrix size.

In the thesis, a custom proof of correctness was provided for the unblocked algorithm and a custom proof inspired by [3] was formulated for the correctness of the blocked algorithm as well.

The main area of possible improvement is the implementation of the four driver routines. The implementation could be rewritten in order to utilize features of modern processors (such as SSE/AVX vector register instructions) or some amount of parallelization. The behavior of the implementation could also be studied more thoroughly so that the algorithm could select the optimal block size automatically depending on the matrix size, the CPU type/architecture and the available number of cores.

The main Cholesky decomposition routines could be extended to a full parallelized alternative to the LAPACK routine `DPOTRF`. To accomplish that, the routines would have to:

1. Provide the functionality to calculate the upper triangular variant of the Cholesky decomposition ($A = U^T U$)

2. Support input matrices whose leading dimension is greater than their size

3. Indicate invalid parameter values by returning non-zero integer values
4. Indicate that the input matrix is not symmetric positive definite, also by returning non-zero integer values

The performance of the blocked Cholesky decomposition routines could also be improved by experimenting with different schedulers or task priorities.

The knowledge of the concepts described in this thesis (block algorithm formulation, working with the LAPACK library and the BLAS interface, fundamentals of task-based programming, testing and evaluation of numerical routines) could be utilized to provide implementations of other blocked numerical algorithms, such as the LU and QR factorizations or the SVD.

Bibliography

1. HEFFERON, Jim. *Linear Algebra.* 4th edition. Ann Arbor, MI, US: Orthogonal Publishing L3C, 2002. ISBN 978-1944325114.
2. LAY, David C.; LAY, Steven R.; MCDONALD, Judi J. *Linear Algebra and Its Applications.* 5th Edition. London, UK: Pearson, 2014. ISBN 978-0321982384.
3. TREFETHEN, Lloyd N.; BAU, David. *Numerical Linear Algebra.* Twenty-Fifth Anniversary Edition. Philadelphia, PA, US: SIAM-Society for Industrial and Applied Mathematics, 2022. ISBN 978-1611977158.
4. NICHOLSON, W. Keith. *Linear Algebra with Applications* [online]. Open Edition. Calgary, Canada: Lyryx Learning Inc., 2020. 2021A [visited on 2023-04-23]. Available from: `https://lyryx.com/linear-algebra-applications`.
5. INTEL CORPORATION. *Intel Developer Reference - Matrix Storage Schemes for LAPACK Routines* [online]. © 2023 [visited on 2023-03-08]. Available from: `https://www.intel.com/content/www/us/en/develop/documentation/onemkl-developer-reference-c/top/lapack-routines/matrix-storage-schemes-for-lapack-routines.html`.
6. BLACKFORD, Susan (ed.). *Netlib - Matrix Storage Schemes* [online]. 2019 [visited on 2023-03-08]. Available from: `https://netlib.org/lapack/lug/node121.html`.
7. DONGARRA, Jack J. *Netlib - Sparse Matrix Storage Formats* [online]. Ed. by BLACKFORD, Susan. 2000 [visited on 2023-03-08]. Available from: `https://netlib.org/utk/people/JackDongarra/etemplates/node372.html`.
8. THE NETLIB FOUNDATION. *Netlib - BLAS (Basic Linear Algebra Subprograms)* [online]. 2022 [visited on 2023-02-16]. Available from: `https://netlib.org/blas/`.
9. LAWSON, Charles L.; HANSON, Richard J.; KINCAID, David R.; KROGH, Fred T. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Trans. Math. Softw.* [online]. 1979, vol. 5, no. 3, pp. 308–323 [visited on 2023-05-10]. ISSN 0098-3500. Available from DOI: `10.1145/355841.355847`.
10. DONGARRA, Jack J.; DU CROZ, Jeremy; HAMMARLING, Sven; HANSON, Richard J. An Extended Set of FORTRAN Basic Linear Algebra Subprograms. *ACM Trans. Math. Softw.* [online]. 1988, vol. 14, no. 1, pp. 1–17 [visited on 2023-05-10]. ISSN 0098-3500. Available from DOI: `10.1145/42288.42291`.
11. DONGARRA, Jack J.; DU CROZ, Jeremy; HAMMARLING, Sven; DUFF, Iain S. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Softw.* [online]. 1990, vol. 16, no. 1, pp. 1–17 [visited on 2023-05-10]. ISSN 0098-3500. Available from DOI: `10.1145/77626.79170`.

12. VAN DE GEIJN, Robert A.; QUINTANA-ORTÍ, Enrique S. *The Science of Programming Matrix Computations.* 5th Edition. Morrisville, NC, US: Lulu.com, 2012. ISBN 978-1257166138.
13. UNIVERSITY OF TENNESSEE. *Basic Linear Algebra Subprograms Technical (BLAST) Forum Standard* [online]. © 1996-2000 [visited on 2023-03-08]. Available from: `https://netlib.org/blas/blast-forum/blas-report.pdf`.
14. DOMBEK, Daniel; KALVODA, Tomáš; KLEPRLÍK, Luděk; KLOUDA, Karel; ŠÍSTEK, Jakub. *Lineární algebra 2* [online, accesible in the internal network of CTU FIT] [visited on 2023-03-29]. Available from: `https://courses.fit.cvut.cz/BI-LA2/@master/textbook/`.
15. ABDELFATTAH, Ahmad; ANZT, Hartwig; DONGARRA, Jack J.; GATES, Mark; HAIDAR, Azzam; KURZAK, Jakub; LUSZCZEK, Piotr; TOMOV, Stanimire; YAMAZAKI, Ichitaro; YARKHAN, Asim. Linear algebra software for large-scale accelerated multicore computing. *Acta Numerica* [online]. 2016, vol. 25, pp. 1–160 [visited on 2023-05-10]. Available from DOI: `10.1017/S0962492916000015`.
16. DEMMEL, James W.; DONGARRA, Jack J.; CROZ, Jeremy Du; GREENBAUM, Anne; HAMMARLING, Sven; SORENSEN, Danny C. *Prospectus for the Development of a Linear Algebra Library for High-Performance Computers* [online]. 1987 [visited on 2023-05-10]. Available from: `http://www.netlib.org/lapack/lawnspdf/lawn01.pdf`. Technical report. LAPACK Working Note.
17. INTEL CORPORATION. *Get Started with Intel® oneAPI Math Kernel Library* [online]. © 2023 [visited on 2023-04-19]. Available from: `https://www.intel.com/content/www/us/en/docs/onemkl/get-started-guide/2023-0/overview.html`.
18. GITHUB, INC. *OpenBLAS - GitHub* [online]. © 2023 [visited on 2023-04-19]. Available from: `https://github.com/xianyi/OpenBLAS`.
19. WANG, Qian; ZHANG, Xianyi; ZHANG, Yunquan; YI, Qing. AUGEM: Automatically Generate High Performance Dense Linear Algebra Kernels on X86 CPUs. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* [online]. Denver, Colorado: Association for Computing Machinery, 2013 [visited on 2023-05-10]. SC '13. ISBN 9781450323789. Available from DOI: `10.1145/2503210.2503219`.
20. ADVANCED MICRO DEVICES, INC. *AOCL User Guide* [online]. 2022 [visited on 2023-04-19]. Available from: `https://www.amd.com/content/dam/amd/en/documents/pdfs/developer/aocl/aocl-v4.0-ga-user-guide.pdf`.
21. ARM LIMITED. *Arm Performance Libraries* [online]. © 1995-2023 [visited on 2023-04-19]. Available from: `https://developer.arm.com/downloads/-/arm-performance-libraries`.
22. MYLLYKOSKI, Mirko. *Task-based parallelism in scientific computing: Introduction to task-based parallelism* [online]. © 2021 [visited on 2023-04-23]. Available from: `https://hpc2n.github.io/Task-based-parallelism/branch/spring2021/motivation/`.
23. OPENMP ARB. *OpenMP Fortran Application Program Interface* [online]. 1997 [visited on 2023-04-23]. Available from: `https://www.openmp.org/wp-content/uploads/fspec10.pdf`.
24. OPENMP ARB. *OpenMP 5.2 API Syntax Reference Guide* [online]. © 2021 [visited on 2023-04-23]. Available from: `https://www.openmp.org/wp-content/uploads/OpenMPRefCard-5-2-web.pdf`.
25. OPENMP ARB. *OpenMP Compilers & Tools* [online]. © 2012-2023 [visited on 2023-04-23]. Available from: `https://www.openmp.org/resources/openmp-compilers-tools/`.

26. AUGONNET, Cédric; THIBAULT, Samuel; NAMYST, Raymond; WACRENIER, Pierre-André. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In: SIPS, Henk; EPEMA, Dick; LIN, Hai-Xiang (eds.). *Euro-Par 2009 Parallel Processing* [online]. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 863–874 [visited on 2023-05-10]. ISBN 978-3-642-03869-3. Available from DOI: `10.1007/978-3-642-03869-3_80`.

27. INRIA CENTRE AT THE UNIVERSITY OF BORDEAUX. *StarPU Handbook* [online]. © 2009-2023 [visited on 2023-04-23]. Available from: `https://files.inria.fr/starpu/doc/html/`.

28. INRIA CENTRE AT THE UNIVERSITY OF BORDEAUX. *StarPU* [online]. 2023 [visited on 2023-04-23]. Available from: `https://starpu.gitlabpages.inria.fr/`.

29. THE NETLIB FOUNDATION. *LAPACK - Linear Algebra PACKage* [online]. 2022 [visited on 2023-03-15]. Available from: `https://netlib.org/lapack/`.

30. IT4INNOVATIONS. *IT4Innovations Documentation* [online]. © 2013-2023 [visited on 2023-04-25]. Available from: `https://docs.it4i.cz/`.

31. FUJITSU. *FUJITSU Processor A64FX - Datasheet* [online]. © 2023 [visited on 2023-04-25]. Available from: `https://www.fujitsu.com/downloads/SUPER/a64fx/a64fx_datasheet_en.pdf`.

Attachment structure

```
README.md ........................ description of the structure of the attachment
├─ impl ......................... directory containing the source code of the implementation
└─ thesis ....................... directory containing the text of this thesis
   ├─ src ....................... directory containing the source code of this thesis in the LaTeX format
   └─ thesis.pdf ................ text of this thesis in the PDF format
```