University of BRISTOL

SCHOOL OF COMPUTER SCIENCE

Advanced Linear Algebra Operations in C++17: Building a High-Performance Library with Modern C++ Features

Xinran Wang

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Bachelor of Science in the Faculty of Engineering.

Monday 6th May, 2024

Abstract

This project developed a high-performance linear algebra library using only the C++17 standard library, filling a gap as the C++ standard library lacks built-in support for linear algebra operations. The library supports critical operations such as matrix multiplication and LU factorization, each optimized using a block algorithm approach. Built on object-oriented principles, this library offers scalability and user-friendliness. Moreover, the use of C++17 features such as auto type deduction, lambda expressions, and parallel execution policy enabled the implementation of parallel versions of the functions, thereby improving performance on multi-core computers. The library supports both dense and sparse matrix classes, implementing COO and CSR data structures for sparse matrices. Performance evaluations indicate that my library achieves faster LU factorization runtimes compared to the widely-used Eigen library. Conclusively, this project has successfully developed an efficient, user-friendly, and highly scalable linear algebra library, providing a powerful tool for scientific computing and engineering applications.

Dedication and Acknowledgements

I would like to express my sincere appreciation to my supervisor, Dr. Tom Deakin, for the priceless guidance and support provided during the entirety of this project.

Declaration

Xinran Wang, Monday 6th May, 2024

Contents

List of Figures

Ethics Statement

This project did not require ethical review, as determined by my supervisor, Dr. Tom Deakin.

Supporting Technologies

- I used CLion as the C++ code editor.
- I used GitHub for version control.
- I used ChatGPT to debug code and to replace words in this dissertation with academic vocabulary.
- I used Python's matplotlib.pyplot and pandas libraries to plot figures.
- I compiled code using the NVIDIA compiler, nvc++.
- I built the project using the C++17 standard library.

Notation and Acronyms

HPC	:	High Performance Computing
COO	:	Coordinate List (format for representing sparse matrices)
CSR	:	Compressed Sparse Row (format for representing sparse matrices)
CPU	:	Central Processing Unit
GPU	:	Graphics Processing Unit
PFLOPS	:	PetaFLOPS (Floating Point Operations Per Second, 10^{15} FLOPS)
GFLOPS	:	GigaFLOPS (Floating Point Operations Per Second, 10^{9} FLOPS)
LAPACK	:	Linear Algebra PACKage
BLAS	:	Basic Linear Algebra Subprograms
BLIS	:	BLAS-like Library Instantiation Software
GEMM	:	General Matrix Multiply (a BLAS routine)
AXPY	:	A*X plus Y (a BLAS routine for linear combination of vectors)
MAGMA	:	Matrix Algebra on GPU and Multicore Architectures
DGETRF	:	Double precision General Triangular Factorization
DLASWP	:	Double precision Linear Algebra Swap

Chapter 1

Introduction

The Importance of High-Performance Computing (HPC)

With the evolution of technologies such as Artificial Intelligence (AI), machine learning, and 3-D imaging, the scale of data has grown rapidly. Real-time data processing is essential to the core of various applications such as live sports broadcasting and stock trend analysis. This demand has spurred a reliance on high-performance computers, consisting of hundreds or even thousands of computing nodes, each equipped with CPUs and GPUs. The essence of High-Performance Computing lies in the development and utilization of advanced technologies and algorithms to maximize the use of these computing resources. In the field of meteorology, HPC enables scientists to perform higher resolution weather forecasting, more detailed simulations of atmospheric phenomena, and complex data assimilation of vast observational data. HPC has significantly improved the accuracy of weather forecasts. Today's five-day weather forecasts are as accurate as three-day forecasts were twenty years ago. This reflects a trend where forecasting skills have improved by one day per decade over the last forty years. Specifically, 20 of the world's fastest 500 supercomputers, which collectively held 7% of the total computational power, or 60 PFLOPs, as of November 2017, are dedicated to weather forecasting [18].

The Importance of Linear Algebra

As a fundamental branch of mathematics, linear algebra is an indispensable part of applications in HPC. It provides a framework for describing and solving problems involving systems of linear equations, vector spaces, and linear transformations, which are central to many scientific and engineering computations. The algorithms and theories of linear algebra underpin various computational methods, such as those used in physical simulations, engineering analysis, machine learning, and data mining, where processing vector and matrix operations is a common requirement [15].

The Importance of Matrix Multiplication and LU Factorization

Matrix multiplication is the foundation of many complex operations in linear algebra. It is fundamental for solving systems of linear equations and performs essential transformations in image processing, such as translations, reflections, and combinations of these operations [22]. It allows these transformations to be represented as matrix forms, and through the product of matrices and image vectors, complex graphical transformations are achieved. Additionally, matrix multiplication plays a vital role in LU factorization, a method that decomposes a matrix into a lower triangular matrix (L) and an upper triangular matrix (U). LU factorization is essential for computing solutions to linear equations, matrix inverses, and for estimating determinants or condition numbers [6].

The Importance of Sparse Matrix

Compared to the more familiar dense matrices, sparse matrices are also extensively used in various industries. For example, in circuit simulation, which often involves solving linear systems composed of node voltage and current equations. These systems can generally be represented as large sparse matrices. Sparse matrix LU Factorization is a crucial computational step in circuit simulation problems [14].

Motivation for the Study

The motivation for this study stems from three primary areas:

1. **Lack of Specialized Components for Linear Algebra in the Existing C++ Standard Library:** Although powerful numerical computing libraries like LAPACK exist, they lack direct support or interfaces for modern C++. This limits their usability and integration within C++ environments.

2. **LU Factorization:** Particularly traditional algorithms such as the Doolittle algorithm, perform efficiently with small-scale matrices. However, these algorithms face performance bottlenecks when dealing with large-scale matrices. This is primarily due to extensive memory access and data movements. Traditional LU factorization also fails to leverage the parallel computing capabilities of modern multi-core processors.

3. **Sparse Matrices:** Sparse matrices, characterized by a majority of zero elements, cannot be efficiently stored using conventional data structures such as two-dimensional arrays. This traditional approach not only wastes significant memory space but also leads to inefficient memory access. It is therefore imperative to develop specialized storage formats and algorithms for sparse matrices.

Project Objectives

The primary objective of this project is to develop a high-performance linear algebra library. This library will leverage the new features of the C++17 standard library and include basic matrix operations such as addition, multiplication, and transposition, with special emphasis on LU factorization. The specific objectives for this project are as follows:

1. Utilize C++ features to design and implement basic class structures and their inheritance relationships, enhancing portability, scalability, and user experience through encapsulation, template classes, and friend functions.

2. Implement efficient linear algebra functions by utilizing C++17's new features, including auto type deduction and lambda expressions to develop a series of foundational linear algebra functions, improving code quality and performance.

3. Optimize matrix multiplication and LU factorization by implementing the block algorithm.

4. Develop parallel versions of algorithms by using C++17's parallel execution policy.

5. Implement data structures and algorithms for sparse matrices using Compressed Sparse Row (CSR) and Coordinate List (COO) formats to support various application scenarios.

6. Assess efficiency improvements by comparing the performance of serial and parallel algorithms. Analyze the impact of different data structures, parallelization strategies, and block sizes on algorithm performance to optimize code structure and enhance scalability. Additionally, my library will be compared with the Eigen library to benchmark its performance.

Chapter 2

Background

This section introduces the related work and methodologies of the project. It encompasses various aspects such as linear algebra libraries, sparse matrix data structure, matrix operation algorithms (such as matrix multiplication and LU factorization) and new features in C++17.

2.1 Linear Algebra Library

2.1.1 BLAS

The Basic Linear Algebra Subprograms (BLAS) package originally covered 38 low-level subprograms essential for the basic operations of numerical linear algebra [16]. The BLAS library is divided into three levels:

- **Level 1 BLAS** focuses on vector operations such as vector addition, scalar multiplication, and dot products.
- **Level 2 BLAS** deals with matrix-vector operations, such as matrix-vector multiplication and solving specific types of linear equations.
- **Level 3 BLAS** involves matrix-to-matrix operations, such as matrix multiplication.

Since complex linear algebra algorithms like LU factorization spend a significant portion of their execution time on these low-level operations, enhancing the performance of these operations can improve the efficiency of complex linear algebra programs.

The primary goal of BLAS is to provide efficient and reliable low-level routines for linear algebra operations. The BLAS standard offers interfaces for both C and Fortran languages; however, its standard reference implementation is written only in Fortran and is not particularly efficient. For instance, matrix multiplication in BLAS is implemented using three nested loops, a straightforward but inefficient approach, especially when dealing with large-scale data, which I will explain in detail in the Algorithm section.

There are optimized versions of BLAS usually provided by hardware vendors or developed by specific projects, such as ATLAS and GotoBL [15]. These BLAS libraries are highly optimized by hardware vendors to fully utilize the specific features of the hardware. For example, BLAS implementations can take advantage of vectorization, where a single instruction can handle multiple data. Additionally, BLAS libraries typically support multithreading, allowing the algorithms to run in parallel on multicore processors, significantly boosting the execution speed of large-scale computational tasks by distributing different computational tasks to different cores.

2.1.2 P1674R1

Owing to the absence of C++ interface support in BLAS and the increasing need for modern C++ features, several proposals have been made to develop a C++ interface for BLAS. Among them, **”P1674R1”** is a proposal for a C++ standard that aims to develop an interface and functionalities based on the BLAS library, better suited for modern C++ usage. My implementation also adopts ideas from this proposal.

Data Type Flexibility

The C interface of BLAS supports only four data types: `float`, `double`, `std::complex<float>`, and `std::complex<double>`. This limits its application in handling different precision floating-point numbers, integers, or custom data types. In C++, template classes can be used to design C++ interfaces. This minimizes the need for redundant functionalities. They enable functions to accept various types of data, enhancing flexibility and efficiency in programming.

Error Handling Improvements

The BLAS standard specifies ways for error checking and reporting; however, these error-handling mechanisms have some flaws, including:

- **Insufficient error recovery:** BLAS's error handling typically causes the program to exit rather than allowing error recovery.
- **Stack unwinding:** When an error occurs, the stack is not unwound.
- **Limited customization of error handling:** Although users can replace the default error handler at link time, this handler must print error information and stop execution, without returning control to the caller.

In C++, exception handling is a common method for managing errors and recovery. It allows programs to catch exceptions upon encountering errors. This allows for subsequent recovery and further error handling, rather than terminating abruptly. This can be implemented by using the `throw` keyword to throw exceptions when specific conditions are met, and by employing `try-catch` blocks. This exception-handling mechanism is more flexible and easier for programmers to maintain.

2.1.3 BLIS

In addition to the C++ implementation of BLAS proposed earlier, the BLIS (BLAS-like Library Instantiation Software) framework represents a new infrastructure for rapidly instantiating BLAS [27]. BLIS provides a C interface that resembles BLAS but is more user-friendly and addresses certain shortcomings of the BLAS interface. It thereby extends its functionalities. Here, I outline some of the inherent issues found in BLAS and demonstrate how BLIS resolves them:

Storage Methods and Flexibility

Native BLAS supports column-major storage, aligning with Fortran conventions, and its C language interface, CBLAS, extends support to both row-major and column-major storage. CBLAS, however, mandates a consistent storage order for all matrices involved in a single operation, prohibiting a combination of row-major and column-major storage formats. Additionally, non-standard storage methods must be implemented through temporary copying and transposing, which can increase performance overhead in large-scale data operations.

BLIS employs the following solutions: The BLIS framework supports mixed operations with column-major, row-major, or general stride storage. For instance, BLIS's GEMM (General Matrix Multiply, $C = \beta C + \alpha AB$) operation can handle matrix multiplication with different storage types, where the three matrices C, A, and B may originate from different data sources or processing stages, thus existing in varying storage formats. For example, matrix A might come from a Fortran-based mathematical library using column-major order, while matrix C might be processed by a C/C++ program stored in row-major order. Matrix B could be acquired from another system that uses general stride storage for specific optimization considerations. The benefits of this approach in BLIS are as follows:

- Different storage methods may optimize performance for specific hardware architectures.
- It eliminates the overhead of data conversion.
- It increases flexibility to accommodate a broad range of user needs.

In my implementation, I have adopted a similar concept by implementing multiple constructors. These constructors enable users to initialize matrix objects using various types of matrix storage.

Handling Complex Numbers

BLAS faces notable limitations when handling certain operations involving complex numbers. Specifically, it cannot simultaneously process complex and real operands within a single operation. For example, BLAS is unable to use a real triangular matrix to update a complex vector, which limits its application in complex numerical computation scenarios. BLIS has implemented comprehensive support for the complex domain; it supports multiple data arrangement methods, such as separate storage (where the real and imaginary parts are stored separately) and interleaved storage (where the real and imaginary parts are stored alternately). It also implements efficient complex arithmetic operations. For instance, in complex matrix multiplication, BLIS processes the real and imaginary parts separately. This allows these operations to take advantage of modern hardware features such as vectorization and parallelization.

Enhancing Level 1 Capabilities

BLIS has enhanced BLAS's level 1 capabilities by directly supporting matrix operations: Traditional BLAS's level 1 supports only vector-level operations, and matrix operations must be indirectly performed through multiple calls to vector operations. BLIS uses instruction fusion technology to optimize consecutive vector operations. For instance, in traditional BLAS implementations, when multiple consecutive vector addition operations are needed, such as:

```
y = y + α0 * x0
y = y + α1 * x1
y = y + α2 * x2
y = y + α3 * x3
```

Each operation would require a separate call to the AXPY (A*X plus Y) function. This means that each execution of AXPY would necessitate loading the vector y into memory, then computing and storing it back. When performing multiple consecutive vector additions, it will load and store the y vector every time and this will cause a significant memory bandwidth consumption. BLIS solves this issue by introducing AXPYF (a fused operation for multiple vector additions). AXPYF allows multiple additions like these to be completed in a single call. Using AXPYF, the vector additions can be combined into one operation, requiring only one loading and storing process to update the y vector. For example, AXPYF can combine the above four operations, compute them all at once, and update y. This significantly reduces memory access and enhances overall efficiency.

2.1.4 LINPACK and LAPACK

LINPACK is an advanced linear algebra library that uses BLAS to perform certain low-level linear algebra operations. Its primary features include support for complex linear algebra problems, such as QR decomposition and LU factorization. It provides optimization for various matrix types, including banded matrices, indefinite symmetric matrices, and triangular matrices [19]. LINPACK relies heavily on BLAS for performing its underlying mathematical operations. This allows it to enhance its computational performance, as BLAS implementations are specifically optimized by hardware vendors for certain hardware architectures. This means that when LINPACK performs complex matrix operations, such as LU factorization, the BLAS subroutines it calls are highly optimized [23]. However, LINPACK was primarily designed for single-processor architectures. With advancements in computing technology, especially with the advent of shared memory and vector supercomputers, the limitations of LINPACK have become apparent. It performs poorly with large-scale data because its memory access patterns do not take advantage of the multi-tiered memory hierarchy of machines, spending too much time moving data rather than performing useful floating-point operations [5].

LAPACK (Linear Algebra PACKage) was developed to overcome the limitations of LINPACK. LAPACK is designed for high-performance computers. LAPACK optimizes memory access patterns and utilizes the multithreading capabilities of modern processors for parallel operations, making it more efficient in handling large-scale problems. A key innovation in LAPACK is its use of block algorithms for operations such as matrix multiplication, LU factorization, and QR decomposition. Small blocks of the matrix are loaded into the cache, and operations are confined to these small blocks, which reduces the need for repeated reads from the main memory and minimizes access to it. Block algorithms are also utilized in my implementation. A detailed discussion of these will be provided in the Algorithm section. The LAPACK paper presents benchmarks of LAPACK running on different architectures. It compares them with LINPACK, showing that LAPACK significantly outperforms LINPACK on shared-memory vector and parallel processors supercomputers, and performs at least as well as LINPACK on serial computers [7].

2.1.5 MAGMA

MAGMA (Matrix Algebra on GPU and Multicore Architectures) is a mathematical library specifically designed for multicore processors and Graphics Processing Units (GPUs) [13]. MAGMA++ is a high-level C++ API for MAGMA.

Data Abstraction

MAGMA++ leverages C++ class concepts to abstract specific data storage types (column-major, row-major, dense/sparse, etc.) and storage memory locations (CPU, GPU, etc.). A key design component is the *SharedStorage* class, an abstract storage class providing a unified interface to manage matrix data of different types. Whether it's column-major, row-major, or dense/sparse matrix formats, *SharedStorage* can handle these through its abstraction layer. This means that developers using this class can apply the same methods and interfaces to operate on these different types of matrices without concerning themselves with the underlying data layout and storage specifics.

Parameter Encapsulation

In traditional BLAS and LAPACK libraries, calling a function to perform an operation involves passing numerous parameters to describe the matrix. This design approach can result in a high number of parameters within the code, which is inconvenient for programmers to remember and use. To address this, MAGMA++ has implemented information encapsulation. MAGMA defines a matrix class (*BaseMatrix*) where basic matrix information such as the number of rows and columns is stored as attributes within the class when creating a matrix object. Furthermore, specialized classes derived from the *BaseMatrix* class are used for different types of matrices (such as general matrices, trapezoidal matrices, triangular matrices, symmetric matrices, etc.). Each specialized class inherits the characteristics of the *BaseMatrix* class and adds features specific to that type of matrix. In my implementation, I also adopted this concept from MAGMA++; I developed an abstract class called *IMatrix*. From this, other specialized classes for dense matrices, sparse matrices, etc. were derived. Additionally, I encapsulated basic matrix information within these classes.

Sparse Matrix

In recent years, over 70 different data storage formats for sparse matrices have been developed [24]. Commonly used sparse matrix data structures include COO (coordinate list) and CSR (compressed sparse row).

The COO format is simple. It uses three arrays to store the values of non-zero elements, their corresponding column indices, and their respective row indices. This format is characterized by its intuitiveness and minimal memory overhead, but it suffers from low computational efficiency. This will be analyzed in the Performance analysis section.

The CSR format, however, is more complex. It also uses three arrays: the `data` array stores the values of non-zero elements, `columnIndex` stores the column indices, but instead of row indices, the third array `RowPointer` stores row pointers. The `RowPointer` array has a length of the number of rows plus one (i.e., if the matrix has M rows, `RowPointer` has a length of $M+1$). `RowPointer[i]` indicates the position in the `data` array where the first non-zero element of the i-th row starts. `RowPointer[i+1]` indicates the position of the first non-zero element of the $i+1$-th row, which is also the end position of the non-zero elements of the i-th row. Here is an example:

$$\begin{matrix} 3 & 0 & 5 & 0 \\ 0 & 4 & 0 & 0 \\ 0 & 0 & 2 & 0 \end{matrix}$$

$$\text{data} = [3, 5, 4, 2], \quad \text{columnIndex} = [0, 2, 1, 2], \quad \text{RowPointer} = [0, 2, 3, 4]$$

Therefore, using this storage method allows for quick access to all the non-zero elements of the i-th row through `RowPointer[i]` and `RowPointer[i+1]`. The CSR storage format not only has low memory overhead but also exhibits high computational performance, which I will analyze in the performance analysis section.

MAGMA++ inherits MAGMA Sparse's support for sparse data formats. To better utilize hardware features, MAGMA++ uses different formats depending on whether the data is stored in CPU or GPU. For CPU, it uses COO and CSR. For GPU, in addition to COO and CSR, MAGMA++ also supports other formats such as ELL, SELL-P, and CSR5 [13].

In my implementation, I primarily implemented two types of sparse matrix storage formats for CPU: COO and CSR.

2.1.6 Eigen library

Eigen is an open-source, powerful, and highly flexible C++ template library specifically designed for linear algebra, matrix, and vector operations. It has become an extremely popular library due to its efficiency and ease of use. It allows programmers to quickly implement complex numerical computations without concerning themselves with the underlying numerical operation details.

Using the Eigen library, one can flexibly define matrices, for example, `Eigen::Matrix3f mat;` defines a 3x3 matrix of type float, or `Eigen::Matrix<float, 3, 4> mat;` defines a 3x4 matrix of type float. If the size of the matrix is not known at compile time, Eigen allows for the definition of dynamically sized matrices with `Eigen::MatrixXf`.

The Eigen library operates independently of external libraries for its fundamental operations. No additional complex mathematics libraries need to be installed; simply including the relevant Eigen header files is adequate for its use.

Furthermore, some of Eigen's functions support parallel execution, such as the `PartialPivLU` (Partial Pivoting LU factorization) function, which Eigen implements using OpenMP. Programmers only need to:

1. Add the appropriate flag when compiling it, such as the `-fopenmp` flag for GCC.
2. Set the number of threads using an environment variable or `Eigen::setNbThreads(n);`

to easily parallelize functions [3].

In the performance analysis section, I will compare the performance of my linear algebra library with the Eigen library.

2.2 Algorithm

2.2.1 Matrix Multiplication

Loop Structures

Matrix multiplication is a critical step in LU factorization. The naive implementation of matrix multiplication is straightforward, using three nested loops. However, since the computations in the innermost loop are independent, the order of these loops can be altered without affecting the final result. Thus, for the loop order, there are six possible combinations. However, the performance of these six implementations can vary depending on how the data is stored and accessed. If a row-major two-dimensional array, which is typical in languages like Java, is used to store matrix data, the performance might resemble the outcomes illustrated in Figure 2.1.

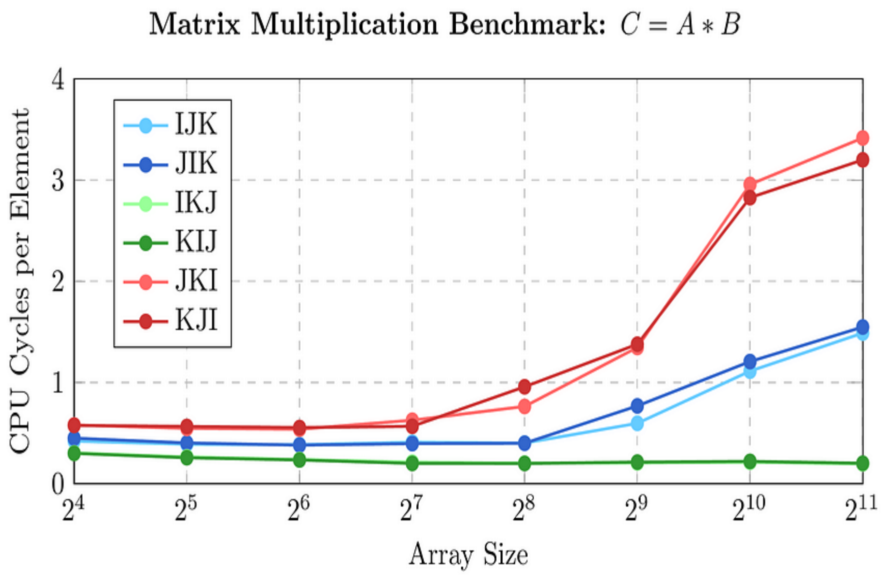


Figure 2.1: **Comparative Performance of Six Loop Order Combinations for Matrix Multiplication in Java.**:This figure shows the results from running JAVA matrix multiplication on a modern Intel platform without thermal throttling or Turbo Boost. Taken from [25].

From the figure, it can be observed that with increasing array sizes, the number of CPU cycles for the IKJ and KIJ loop orders does not increase significantly and always remains the least. Fewer CPU cycles mean that the same computational tasks are completed in less time, thus indicating higher efficiency. This suggests that the loop structures of IKJ and KIJ are the most efficient. Here is an example using the IKJ loop order:

```
for (int i = 0; i < m; ++i) {
    for (int k = 0; k < p; ++k) {
        for (int j = 0; j < n; ++j) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

Analyzing the most efficient loop structure, as in the IKJ example: During the execution of the inner loop, $A[i][k]$, remains constant with each iteration over j, meaning that the elements of matrix A do not need to be repeatedly read from memory during the traversal of j. This reduces the number of memory accesses. For matrix B, the elements $B[k][j]$ are accessed sequentially in the inner loop over j, which matches the access pattern with the memory layout, improving cache hit rate. This is because modern processors prefetch consecutive memory blocks to reduce access latency. Similarly, matrix C is also accessed sequentially.

By contrast, considering the least efficient loop structure, as in the JKI example: In this loop structure, elements of matrices A and C are accessed by columns. Since JAVA uses row-major storage, elements between columns are not contiguous in physical memory. Such an access pattern leads to potentially reloading data from memory in each iteration, causing frequent cache misses and higher memory access latency. Therefore, due to the processor's inability to effectively prefetch and cache the required data, this significantly reduces the CPU cache utilization and results in more CPU cycles being consumed in memory access.

Therefore, for languages that use row-major storage, using the IKJ or KIJ loop structure in matrix multiplication can yield better efficiency. In my implementation, as my matrix objects also use row-major storage, I have used the IKJ loop structure for the algorithms.

Block algorithm

While the mentioned method is suitable for handling small matrices, it exhibits poor performance when dealing with large matrices. This is because the row of the matrix cannot be fully stored in the cache.

It can potentially lead to frequent reloading of data from memory during computations. To address this issue, a block algorithm is employed. The block algorithm divides the matrix into smaller submatrices (blocks) so that the size of blocks fit within the CPU's cache. The algorithm is implemented with three outer loops and three inner loops. I will explain why the block algorithm can optimize performance through the following example, as illustrated in Figure 2.2. The matrix size is 4×4, and the block size is 2×2. The algorithm calculates each block of the C matrix in turn. When calculating the C_{11} block, firstly when calculating γ_{11}, if the cache can accommodate three blocks, the blocks A_{11}, B_{11}, and C_{11} are simultaneously loaded into the cache. As these blocks are already in the cache, there is no need for further memory reads when calculating γ_{12}, γ_{21}, γ_{22}. In the second round, the blocks A_{12}, B_{21}, and C_{11} are loaded into the cache, and the product of A_{12} and B_{21} is computed and added to the C_{11} block. This finishes the calculation of the C_{11} block.

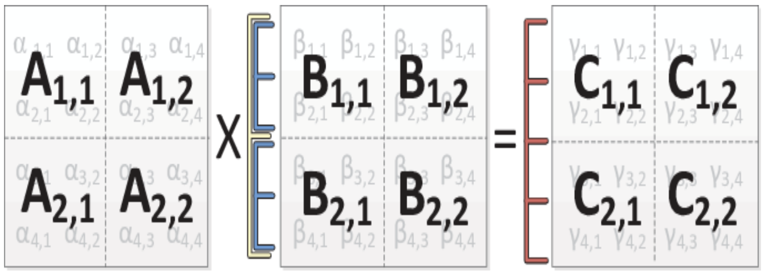


Figure 2.2: Block Matrix Multiplication. Taken from [9].

2.2.2 LU Factorization

Overview of LU Factorization

LU factorization is a mathematical method used to decompose a matrix into the product of two matrices, specifically a lower triangular matrix (L) and an upper triangular matrix (U). LU factorization is also employed as a benchmarking tool due to its high computational complexity, effectively utilizing and testing the performance of computing systems, especially in parallel computing environments. For instance, both the LINPACK Benchmark [12] and the High-Performance Linpack (HPL) Benchmark [8] utilize LU factorization as a part of their benchmarks. In LU factorization, matrix A is the multiplication of a lower triangular matrix L and an upper triangular matrix U, such that $A = LU$. In matrix L, the diagonal elements are typically set to 1, with all non-zero elements located below the diagonal. In matrix U, all non-zero elements are located on or above the diagonal. This decomposition is particularly useful in numerical analysis, especially for solving systems of linear equations, calculating matrix inverses, or determining determinants.

Solving linear equations using LU Factorization

When solving the system of linear equations $Ax = b$, methods like Gaussian elimination or Cramer's rule could be used, but these can be time-consuming and numerically unstable for large matrices. However, once the L and U matrices of A are determined, the solution can be quickly obtained by breaking down the original system $Ax = b$ into two simpler systems: $Ly = b$ and $Ux = y$. The answer to why solving two systems is faster than solving one lies in the special structure of the matrices. In the equation $Ly = b$, since L is a lower triangular matrix, all its non-zero elements are on or below the diagonal. This structure allows the equation to be solved through forward substitution. Starting from the first row, each row contains one new unknown, which can be directly solved and then substituted into the next row to solve the next unknown, and so forth, until all unknowns are resolved. Similarly, in the equation $Ux = y$, since U is an upper triangular matrix, all its non-zero elements are on or above the diagonal. This arrangement permits using backward substitution starting from the last row. Starting with the bottom equation, each equation can immediately solve for one unknown (starting with the last unknown), and then this value is substituted into the previous equation to solve for another unknown, and so forth, until all unknowns are resolved. Solving such two systems is computationally simpler than solving general systems of linear equations. Each step involves solving for a single unknown directly without the need for complex elimination processes. This approach reduces computational effort and avoids potential numerical instabilities that might arise during the elimination process.

Doolittle Algorithm

I will now explain how to perform LU factorization on a matrix: first, introduce a naive algorithm for computing the LU factorization, known as the Doolittle algorithm [26]. The calculation order of the algorithm is illustrated in Figure 2.3, following the order of colors shown. Based on the principles of matrix multiplication, the decomposition calculates each element according to the formulas:

For u_{ij}, the upper triangular matrix:

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik}u_{kj} \quad \text{for } j \geq i$$

For l_{ij}, the lower triangular matrix:

$$l_{ij} = \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik}u_{kj}\right)/u_{jj} \quad \text{for } i > j$$

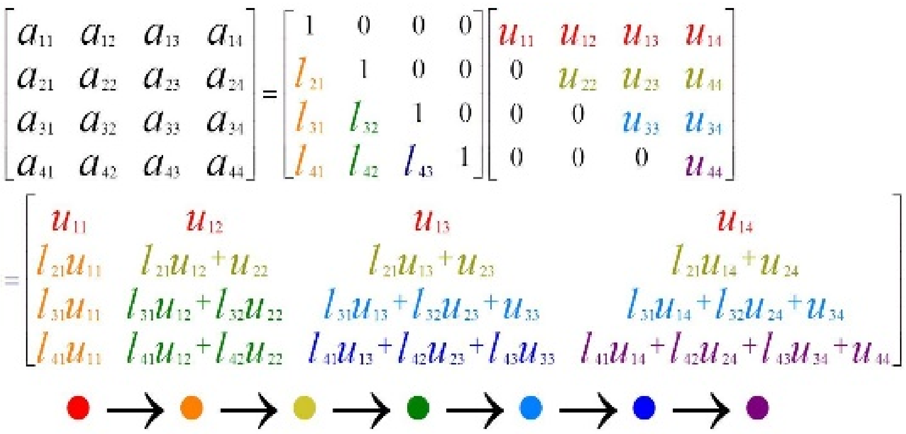


Figure 2.3: Doolittle algorithm.

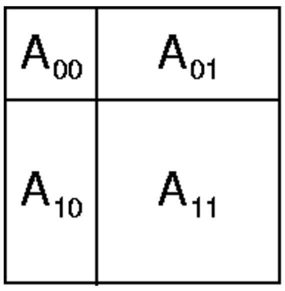
The code implementation is as follows:

```c
for (int k = 0; k < n; ++k) {
    for (int i = k + 1; i < n; ++i) {
        mat[i*n + k] /= mat[k*n + k];
        for (int j = k + 1; j < n; ++j) {
            mat[i*n + j] -= mat[i*n + k] * mat[k*n + j];
        }
    }
}
```

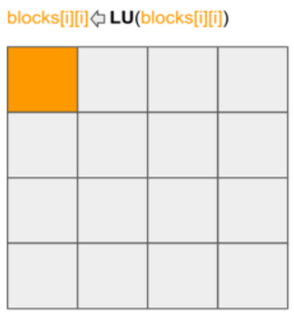
The naive Doolittle algorithm is relatively simple to implement but has several disadvantages. First, if there are zero or near-zero elements on the diagonal of the matrix, the Doolittle algorithm may fail, as this results in division by zero. Secondly, similar to naive matrix multiplication, processing large-scale matrices with this algorithm results in significant cache inefficiencies. This increases the frequency with which the processor must access main memory, thus reducing the efficiency of the program.

Block LU Factorization

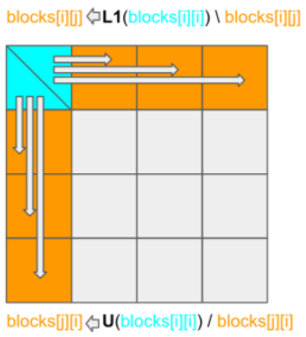
I will explain how LU factorization is implemented in LAPACK. LAPACK employs a block algorithm approach. First, LAPACK implements serial block LU factorization, which is as follows:



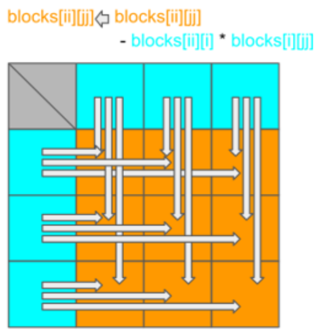
(a) Step 1



(b) Step 2


blocks[i][j] ⇦ L1(blocks[i][i]) \ blocks[i][j]
blocks[j][i] ⇦ U(blocks[i][i]) / blocks[j][i]

(c) Step 4


blocks[ii][jj] ⇦ blocks[ii][jj]
- blocks[ii][i] * blocks[i][jj]

(d) Step 5

Figure 2.4: Block LU Factorization. Taken from [4]

1. **Partition Matrix into four blocks 2.4a**
2. **Apply the Naïve LU factorization on block A_{00} 2.4b**
   (a) Find the pivot value in the k-th column and set that row index to pivotRow.
   (b) Interchange the pivot row with the k-th row.
   (c) Scale the column, making k-th column but in a block all divided by the diagonal element.
   (d) Perform a rank-k update on a matrix but within a given range of rows and columns.
3. **For the blocks to the left and right of A_{00}, perform row exchange.**
4. **Calculate the new values for A_{01} and A_{10}, the new values correspond to U_{01} and L_{10} respectively 2.4c**
   - Formula for U_{01}:
   $$U_{01} = L_{00}^{-1} \times A_{01}$$
   - Formula for L_{10}:
   $$L_{10} = A_{10} \times U_{00}^{-1}$$
5. **Update A_{11} 2.4d**
   - Formula:
   $$A_{11} = A_{11} - L_{10} \times U_{01}$$
6. **Partition A_{11} into four blocks again and iterate the above steps**
   [20]

This algorithm utilizes a block algorithm, which offers the advantage of decomposing large matrices into smaller blocks. The size of each block is designed to match the processor's cache size, ensuring that data remains within the cache during processing and reducing the frequency of memory accesses. Moreover, during the initial step of the algorithm, when executing Naïve LU factorization, partial pivoting is introduced. Partial pivoting involves selecting the largest absolute remaining element as the pivot at each step, which mitigates numerical instability caused by very small divisors. This addresses the issue mentioned earlier where diagonal elements are close to zero. By choosing larger pivots, numerical errors are minimized, thereby enhancing the overall numerical stability of the algorithm. Since partial pivoting is used, a permutation matrix P is required to record the row swaps, because the decomposition now is $A = PLU$ rather than $A = LU$.

Since native LAPACK lacks support for distributed memory, additional measures are necessary to enable parallelism on distributed memory systems. LAPACK was extended to create ScaLAPACK, a distributed memory system comprised of multiple computing nodes, each with its local memory. These nodes are interconnected through high-speed networks or point-to-point links, and message-passing interfaces (MPI) is a common method for these nodes to share data and communicate. This contrasts with the shared memory model, which allows multiple cores on the same node to access the same memory, with common implementations including the use of OpenMP. In my implementation, I adopted the shared memory model and specifically utilized the C++ execution policy for implementing parallel LU Factorization.

2.2.3 Execution Policy

C++17 introduced execution policies to its Standard Algorithms Library, enabling the parallelization of these algorithms. Originally, the C++ Standard Algorithms Library offered a variety of general-purpose algorithms, such as `for_each`, `swap`, and `transform` [2]. Historically, utilizing these algorithms were cumbersome, largely because they required a callable object, necessitating the creation of a new object. However, the introduction of Lambda expressions in C++11 simplified their usage. For instance, below is an example that uses a lambda expression with the `for_each` algorithm to iterate over a container:

```cpp
std::vector<int> nums {1, 2, 3, 4, 5};
std::for_each(nums.begin(), nums.end(), [](int& x) {
    x = x * x;
});
```

In C++17, these algorithm interfaces were further refined to support optional execution policies. These execution policies inform the library about the assumptions it can make, enabling certain optimizations. C++17 provides three execution policies: the default, `std::execution::seq` for sequential execution; `std::execution::par`, used for parallel execution, suggesting that the algorithm may execute across multiple threads to enhance performance; and, `std::execution::par_unseq` for parallel and vectorized execution, the most aggressive form of parallel execution that allows for multi-threading and vectorization within individual threads using SIMD (Single Instruction Multiple Data) instructions. C++20 introduced an additional execution policy, `std::execution::unseq`, specifically optimized for vectorization [1].

Utilizing execution policies is straightforward:

```cpp
std::for_each(std::execution::par, nums.begin(), nums.end(), [](int& x)
    {
    x = x * x;
});
```

When using the `std::execution::par` policy, the parallel implementation of the algorithm is typically managed by the C++ runtime environment and the operating system, which includes the creation, execution, and destruction of threads. This reduces the complexity of parallel programming, but it limits finer-grained control over threads. For instance, programmers cannot specify the number of parallel threads; the C++ runtime library automatically adjusts the number of threads based on available hardware resources and system load.

Chapter 3

Implementation

In this section, I will explain how I developed a high-performance linear algebra library using the C++17 standard library.

3.1 Library Architecture

3.1.1 Inheritance

Given that C++ is an object-oriented programming language, I opted to utilize its three main object-oriented features — inheritance, polymorphism, and encapsulation – to build a user-friendly, scalable, and highly modular class structure. The most used data type in linear algebra is the matrix, hence, I will design a matrix class.

There are many types of matrices, such as dense matrices, sparse matrices, and triangular matrices. Since matrices such as sparse matrices, can be implemented in more than one way, I decided to first construct a base matrix class called IMatrix. This class is an abstract class that provides several general methods, such as getRows(), getCols(), and printNonZeroElements(). Following this, I built the general matrix class: Matrix, also known as the dense matrix class. Additionally, I also constructed two sparse matrix classes: SparseMatrixCOO and SparseMatrixCSR. These three classes all inherit from the IMatrix abstract class which are demonstrated in Figure 3.1.

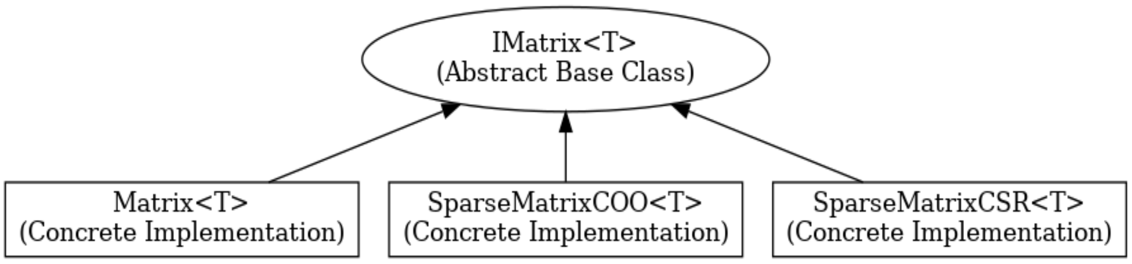


Figure 3.1: Inheritance

3.1.2 Template Class

In the Java programming language, there is a mechanism known as generics that allows methods to accept parameters of various types at call time. Based on the type of parameter passed to the generic method, the compiler appropriately handles each method call [11]. In C++, a similar technology exists, known as template classes and template functions. By utilizing template classes and functions, I enabled users to create matrices of any data type, such as int, float, double, etc. Additionally, I have incorporated the declaration:

```
typename = std::enable_if_t<std::is_arithmetic<T>::value>
```

in the template class to ensure that the data types provided by users are arithmetic and not non-computational, such as, char or string. It is important to note that both the declaration and implementation of template classes must be placed in the header file, not split between a .h and a .cpp file, as with non-template classes and functions. This is primarily because templates in C++ are instantiated with specific types at compile time. The compiler needs to see both the declaration and the definition of the template at the same time. If the declaration and definition of a template are placed in separate files, the compiler might not see the template definition when compiling other .cpp files, which can lead to linkage errors.

3.1.3 Constructors

I also wrote various initialization constructors to facilitate user access, for example, when a user wishes to initialize a dense matrix object, they can pass in a 2D vector, a 1D vector, or a 2D list. All of these can successfully initialize a dense matrix object. The constructors are defined as follows:

```
Matrix();
Matrix(size_t rows, size_t cols);
Matrix(size_t rows, size_t cols, T fillValue);
Matrix(size_t rows, size_t cols, const std::vector<T>& data);
Matrix(const std::vector<std::vector<T>>& data);
Matrix(std::initializer_list<std::initializer_list<T>> list);
```

3.1.4 Data Structure

Furthermore, irrespective of the type of data provided by the user during the initialization process, I constructed a row-major one-dimensional vector to store the values of the dense matrix object. The reason for using a one-dimensional vector as the storage for matrix element values is that all elements in a one-dimensional vector are structured in contiguous memory. This enhances the number of cache hits, leading to better cache utilization. In contrast, when using a 2D vector (`std::vector<std::vector<T>>`) for storage, the outer vector allocates a contiguous block of memory only to store references (pointers) to the inner vectors. This ensures only the continuity of these references' addresses, not the continuity of the addresses to which they point. Thus, while the elements in an inner vector are continuous, the addresses between each inner vector are discontinuous. Additionally, I did not use a one-dimensional array because `std::vector` supports dynamic resizing, allowing it to expand automatically according to data needs, accommodate uncertain data volumes, and manage memory automatically. It also provides a wealth of methods such as `push_back()`, `size()`, etc., simplifying programming work and reducing the risk of errors. In contrast, a one-dimensional array has a fixed size and is not easily expandable.

3.1.5 Encapsulation

Next, in each class, I used the `private` keyword to encapsulate member variables. I wrote public member functions, allowing users to access or modify member variables through these functions, such as `get()` and `set()`. Beyond encapsulating variables, I also encapsulated functions, for example, in the dense matrix class, these functions are public functions:

```
Matrix<T> LU_Factorization(int blockLength, int smallBlockSize) const;
Matrix<T> Parallel_LU_Factorization(int blockLength, int smallBlockSize)
    const;
```

but they make extensive use of private functions within these two public functions. Encapsulation leads to the code becoming easier to understand and maintain. It also increases the security of the code.

I then override basic operators to facilitate user interaction, for example, I override the `+`, `-`, `==`, `=`, `!=`, and `<<` operators using friend functions. In C++, *friend* functions are a mechanism that allows certain non-member functions to access a class's private and protected members. This functionality allows two or more classes to share direct access to each other's data without going through a public interface. The following example is the definition of a friend function that overrides the addition operator:

```
friend Matrix<T> operator+(const Matrix<T>& mat1, const Matrix<T>& mat2)
    ;
```

Using friend functions makes it more convenient for users, providing a more intuitive calling experience. I will provide usage examples at the end of this section.

3.1.6 Exception Handling

I have implemented comprehensive exception handling in my functions. This enhances program robustness and maintainability, while also improving the user experience. For example, before executing the core function `DGETRF_intern()`, I validated the parameters to ensure the inputs are correct. After the core function, if the core function fails, the external function `LU_Factorization()` can catch the exception and handle it accordingly. Below is an example code snippet:

```cpp
template<typename T>
Matrix<T> Matrix<T>::LU_Factorization(int blockLength, int
    smallBlockSize) const {
    if (this->m_rows != this->m_cols) {
        throw std::invalid_argument("Matrix is not square.");
    }
    if (blockLength <= 0 || smallBlockSize <= 0) {
        throw std::invalid_argument("Block size must be positive.");
    }
    if (!std::is_arithmetic<T>::value) {
        throw std::invalid_argument("The type of the matrix must be
            arithmetic.");
    }

    // Initialization
    ...

    // Exception handling during LU factorization
    try {
        // Linear algebra function
        DGETRF_intern(result.m_data, result.getRows(), blockLength, p,
            smallBlockSize);
    } catch (const std::exception& e) {
        std::cerr << "An error occurred during LU factorization: " << e.
            what() << std::endl;
        throw;
    }

    // return
    ...
}
```

3.2 Algorithm Implementation

3.2.1 Sparse Matrix

This section discusses the implementation of sparse matrices, focusing on two common data structures used in the MAGMA library: the coordinate list (COO) and the compressed sparse row (CSR).

COO Implementation

For the COO implementation, I use three one-dimensional vector arrays to store the non-zero elements, row indices, and column indices, respectively. I implement the addition operation for sparse matrices in COO format in two methods.

Addition Operation Method 1 First, I iterate the elements of the first matrix and place these elements in the corresponding positions of the result matrix. Then, I iterate the non-zero elements of the second matrix. If there is already an element at the same position in the result matrix, I add the values of the two matrices; if there is no element at that position in the result matrix, I directly set the element. This method proved to be inefficient and was ultimately not used, which I will discuss in the performance analysis section.

Addition Operation Method 2 I employed an unordered_map data structure to merge the non-zero elements of the two matrices, which performs well with low hash collisions.

CSR Implementation

For the CSR implementation, I use three one-dimensional vectors to store the non-zero elements, column indices, and row pointers. The addition operation for CSR data structure is relatively more complex.

Addition Operation The method is as follows: for each row, use two pointers to iterate the non-zero elements of the two input matrices in that row:

- If both matrices have elements in the same column and the column indices match, add these two elements and place the sum in the result matrix.
- If one matrix has an element in a column while the other does not, directly add the element and column index of the matrix with the element to the result matrix.

For handling the remaining elements: after processing the shared elements of the two matrices in the current row, if one matrix has finished processing its elements in that row while the other still has remaining elements, continue to add the remaining elements to the result matrix. To update row pointers: after processing each row, update the row pointers of the result matrix to point to the position after the last non-zero element.

3.2.2 Matrix Multiplication

In the general matrix class (dense matrix) Matrix, I initially implemented a serial block matrix multiplication algorithm. The function is declared as follows:

```
Matrix<T> blockMultiplication(const Matrix<T>& other, int blockSize =
    100) const;
```

This function allows users to manually set the block size, with a default size of 100. This design lets users adjust the block size based on different hardware platform requirements to optimize performance. Following this, I implemented a parallel block matrix multiplication algorithm using C++17 parallel execution policy. First, I constructed pairs of block indices for the result matrix. These index pairs allow access to the blocks currently being calculated. Then, I used the parallel execution policy to parallelize loops over these block indices. Each thread handles one block and computes the values of its respective block independently. As each block's calculations are independent of others, there is no data race.

3.2.3 Serial LU Factorization

I will now describe how I implemented serial LU factorization, adopting the algorithmic approach used in LAPACK for serial LU factorization, with modifications in the actual implementation code.

Permutation Array

Initially, I added a permutation matrix attribute, P, to the matrix class to represent the permutation matrix P. Typically, a permutation matrix P is a two-dimensional matrix used to record the operations of swapping matrix rows. However, as each row of the permutation matrix has only one element set to 1 (indicating the new position to which that row has been swapped) and the rest are 0, using a two-dimensional matrix for storage would significantly waste storage space. Therefore, I opted to use a one-dimensional array to replace the traditional two-dimensional permutation matrix. In this array, the value at each position represents the new position of the corresponding row after row swapping. For example, in the case of a 5x5 matrix, after LU factorization, the values in the result matrix object's attribute P might finally be $[1, 0, 3, 2, 4]$, indexing from 0 to 4. This indicates that the original matrix's row 0 and row 1 are swapped, and row 2 and row 3 are swapped.

DGETRF_internal

Next, here is the overall function process: initially, users call the external interface `LU_Factorization`, and the function outputs a new matrix object containing the L and U matrices. The size of this result matrix matches those of the input matrix. Specifically, the elements of the L matrix are stored in the lower triangular part (excluding the diagonal), and the elements of the U matrix are stored in the upper triangular part. The `LU_Factorization` function calls the private function `DGETRF_internal`. This function is defined as follows:

```cpp
template<typename T>
void Matrix<T>::DGETRF_internal(std::vector<T> &mat, int n, int
    blockLength, std::vector<int> &pMat, int smallBlockSize) const {
     int blockStart = 0;
     for (; blockStart+blockLength <= n; blockStart+=blockLength) {

         DGETF2(mat, pMat, blockLength, blockStart, n);

         DLASWP(mat, pMat, n, blockStart, blockLength);

         DTRSM(mat, n, blockStart, blockLength);

         DTRSM_L21(mat, n, blockStart, blockLength);

         BLOCK_DGEMM(mat, n, blockStart, blockLength, smallBlockSize);
     }
     if (blockStart < n) {
         DGETF2(mat, pMat, n-blockStart, blockStart, n);
         DLASWP(mat, pMat, n, blockStart, n-blockStart);
     }
}
```

In implementing function `DGETRF_internal`, I employed an iterative approach to iterate blocks, rather than using recursion. I made this decision to mitigate the risk of stack overflow, which can be induced by deep recursion due to increased stack depths. Additionally, I incorporated boundary handling at the end of the function to manage scenarios where the block size does not evenly divide the matrix, resulting in residual blocks.

During each iteration, the function initially invokes the `DGETF2` function to compute the LU values of the A_{00} block while updating the P array to record the necessary row-swapping information. Subsequently, the `DLASWP` function applies these row swaps to both the left and right sides of the A_{00} block referred to as the green parts in Figure 3.2.

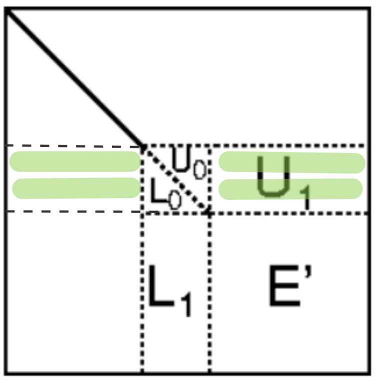


Figure 3.2: Illustration of the green parts where row swaps are applied.

DLASWP

The implementation of the `DLASWP` function utilizes two strategies:

Simple Copy Strategy This method involves creating a copy of a portion of the matrix and updating the appropriate rows of the original matrix based on the P array. While this approach is straightforward and easy to understand, it is inefficient due to the additional memory operations and data copying involved.

Optimized Swap Strategy (the final implementation) This method operates directly on the original matrix, avoiding unnecessary copying. It leverages the row swap information recorded in the P array, by directly updating the matrix by performing multiple row swaps. This approach is more efficient as it reduces data copying and operates directly on the original matrix.

The specific implementation is as follows: initially, a local array A is initialized to track the initial position of each row. During the iteration process, each element $A[i]$ is checked to see if it is in the correct position; if not, reiterate to find the index of the element it should be swapped with. Once found, the `DSWAP` function is called to exchange the left and right parts of the matrix. Finally, the local array A is updated to reflect the swapped rows.

BLOCK_DGEMM

Following this, the functions to compute new values for U_1 and L_1 as depicted in Figure 3.2, are executed. Then, the function `BLOCK_DGEMM` updates E'. The essence of this function is to compute the multiplication of two matrices, so I implemented it using the block algorithm previously developed for matrix multiplication. The default block size of this function is consistent with the block size used in LU factorization, but it can also be set by the user.

Serial LU Factorization function Call graph

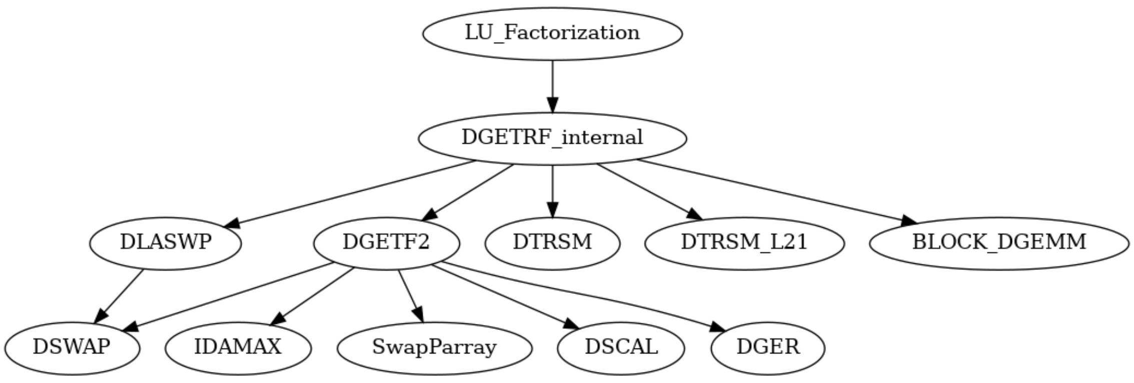


Figure 3.3: Call graph for the serial `LU_factorization` function.

3.2.4 Parallel LU Factorization

Execution Policy

To implement parallel LU factorization, I utilized the parallel execution policy in C++17. Introduced in the background section, standard library algorithms such as `std::for_each` support the acceptance of execution policy parameters, enabling parallel execution. By passing `std::execution::par` as a parameter to `std::for_each`, iterations can be executed in parallel across multiple threads. Each iteration performs operations defined by a lambda expression. Additionally, C++ automatically determines the default number of threads based on the system's physical or logical core count, a decision handled internally by C++. My parallel LU factorization algorithm is almost identical to the serial version, except that, for functions called within `PDGETRF_intern` and the subsequent subroutines they invoke, I strive to utilize execution policy to achieve parallel optimization wherever possible. Below is an example implementation using execution policy:

```
template<typename T>
void Matrix<T>::PDSCAL(std::vector<float>& mat, int n, int blockLength,
    int k, int blockStart) const {
    int numElements = blockStart + blockLength - (k + 1);
    std::vector<int> indices(numElements);
    std::iota(indices.begin(), indices.end(), k + 1);
    float diagonalElement = mat[k * n + k];
    std::for_each(std::execution::par, indices.begin(), indices.end(), [
       n, k, diagonalElement, &mat](int i) {
        mat[i * n + k] /= diagonalElement;
    });
}
```

The PDSCAL function, used within PDGETF2 for matrix scaling operations, generates an increasing sequence starting from $k + 1$, which represents row indices. During the subsequent parallel operations, each thread processes different elements (i.e., different rows) from the indices, ensuring that the matrix elements handled by each thread do not overlap. This design prevents data races. Additionally, in the lambda's capture clause, I explicitly capture variables as $[n, k, diagonalElement, \&mat]$ instead of using [&] to capture all by reference. Specifying the capture list allows the compiler to optimize memory access and allocation, reducing performance overhead due to unnecessary variable captures. When a lambda expression captures variables by value, it only creates copies of the variables being used, avoiding the capture of the entire outer scope. This approach not only reduces memory usage but also prevents unintended access to unrelated variables in a multithreaded environment, enhancing the efficiency of data access. Moreover, explicit value capture ensures that each thread accesses a local copy, reducing the need for mutex lock during multi-threaded execution.

Data Race and Solution

However, not all for-loops can directly utilize execution policy for parallel execution due to data dependencies. It is necessary to consider the order of data access and correctly parallelize the appropriate for-loops. Take the following two functions as examples:

- Compute U_{12} (A_{12}):

$$A_{12} = L_{11}^{-1} \times A_{12}$$

```
template<typename T>
void Matrix<T>::PDTRSM(std::vector<float>& mat, int n, int
    blockStart, int blockLength) const {
    std::vector<int> indicesJ(n - blockStart - blockLength);
    std::iota(indicesJ.begin(), indicesJ.end(), blockStart +
        blockLength);
    for (int i = blockStart; i < blockStart + blockLength; ++i) {
        std::for_each(std::execution::par, indicesJ.begin(),
            indicesJ.end(), [blockStart, i, n, &mat](int j) {
            float sum = 0.0;
            for (int k = blockStart; k < i; ++k) {
                sum += mat[i * n + k] * mat[k * n + j];
            }
            mat[i * n + j] -= sum;
        });
    }
}
```

- Compute L_{21} (A_{21}):

$$A_{21} = A_{21} \times U_{11}^{-1}$$

```
template<typename T>
void Matrix<T>::PDTRSM_L21(std::vector<float>& mat, int n, int
    blockStart, int blockLength) const {
    std::vector<int> indices(n - blockStart - blockLength);
    std::iota(indices.begin(), indices.end(), blockStart +
        blockLength);
    std::for_each(std::execution::par, indices.begin(), indices.end(), [blockStart, blockLength, &mat, n](int i) {
        for (int j = blockStart; j < blockStart + blockLength; ++j) {
            float L21_value = mat[i * n + j];
            for (int k = blockStart; k < j; ++k) {
                L21_value -= mat[i * n + k] * mat[k * n + j];
            }
            if (mat[j * n + j] != 0) {
                mat[i * n + j] = L21_value / mat[j * n + j];
            }
        }
    });
}
```

Firstly, the function PDTRSM updates the U_{12} matrix block shown in Figure 3.4. The update formula relies on the known elements of L_{11} and the current elements of A_{12}. The parallelization in this function is conducted column-wise (i.e., based on the j index), because the update of elements within each column is independent. However, for each row i, the computation of its elements depends on the results of all preceding rows in that row. This means that the computation for row i must wait until row $i-1$ is completed, hence parallelizing by rows would lead to data dependency issues and incorrect results.

In contrast, for the function PDTRSM_L21, it is safe to parallelize iteratively over i. The computation of each column j depends on the results of earlier calculations in the same column, which means that parallelizing over j would lead to read-write conflicts and erroneous outcomes.

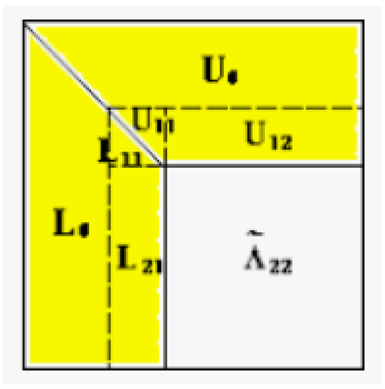


Figure 3.4: Data dependencies illustration. Taken from [20].

3.3 Library Usage Examples

This subsection provides a detailed example demonstrating how to utilize my library.

```
#include "Matrix.hpp"
#include "SparseMatrixCOO.hpp"
#include "SparseMatrixCSR.hpp"

int main() {
    // Initialize two matrices by using different constructors
    std::vector<std::vector<float>> vec{{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
    Matrix<float> matrixA(vec);
    Matrix<float> matrixB({{1, 4, 7}, {2, 5, 8}, {3, 6, 9}});

    // example of how to use functions
    Matrix<float> matrixC = matrixA + matrixB;

    Matrix<float> matrixD = matrixA * matrixB;

    Matrix<float> matrixE = matrixA.blockMultiplication(matrixB, 3);

    Matrix<float> matrixF = matrixA.parallelBlockMultiply(matrixB, 3);

    Matrix<float> matrixG = matrixA.LU_Factorization(3);

    Matrix<float> matrixH = matrixA.Parallel_LU_Factorization(3);

    // test sparse matrix COO
    SparseMatrixCOO<int> matrix1 = {
        {0, 0, 3, 0, 0},
        {4, 0, 0, 0, 0},
        {0, 0, 0, 0, 0},
    };
    SparseMatrixCOO<int> matrix2 = {
        {1, 0, 3, 0, 0},
        {0, 0, 0, 0, 0},
        {0, 0, 0, 8, 0},
    };

    SparseMatrixCOO<int> result1 = matrix1 + matrix2;

    // test sparse matrix CSR
    SparseMatrixCSR<int> matrix3({
        {1, 0, 0, 0},
        {0, 0, 3, 0}
    });

    SparseMatrixCSR<int> matrix4({
        {1, 0, 0, 0},
        {0, 2, 0, 0}
    });
    SparseMatrixCSR<int> result2 = matrix3 + matrix4;

    result2.printNonZeroElements();
    return 0;
}
```

Chapter 4

Performance Analysis

In this section, I will test, analyze, and compare the performance of the linear algebra library I implemented. All tests were conducted on my personal computer, which has the following specifications:

CPU Information

- Model: Intel(R) Core(TM) i5-9300H CPU
- Clock Speed: 2.40 GHz
- Number of Physical Cores: 4
- Number of Threads (Logical Cores): 8
- Cache Size: L1 - 256 KB, L2 - 1.0 MB, L3 - 8.0 MB

Memory Information

- Total Size: 8 GB

4.1 Loop Structure Analysis

As mentioned in the Background section, there are six distinct loop structures for matrix multiplication, along with test results using Java. I have written C++ code to test whether the performance and conclusions are consistent with those findings. I utilized a row-major one-dimensional vector to store matrix data, which is consistent with the storage format used in the Java implementation from the background. I calculated the GFLOPS (Giga Floating-point Operations Per Second) for the six loop structures of matrix multiplication for matrices of varied sizes. The calculation method is as follows: assuming matrix A is of size $m \times k$ and matrix B is $k \times n$, then matrix C will be $m \times n$. Calculating each element of matrix C requires k multiplications and $k - 1$ additions. For simplicity, I assumed about $2k$ operations per element. Therefore, the total number of floating-point operations required for the entire matrix multiplication is $2k \times m \times n$. This leads to the following formula:

$$\text{GFLOPS} = \frac{2 \times k \times m \times n}{\text{execution time in seconds}} \times 10^{-9}$$

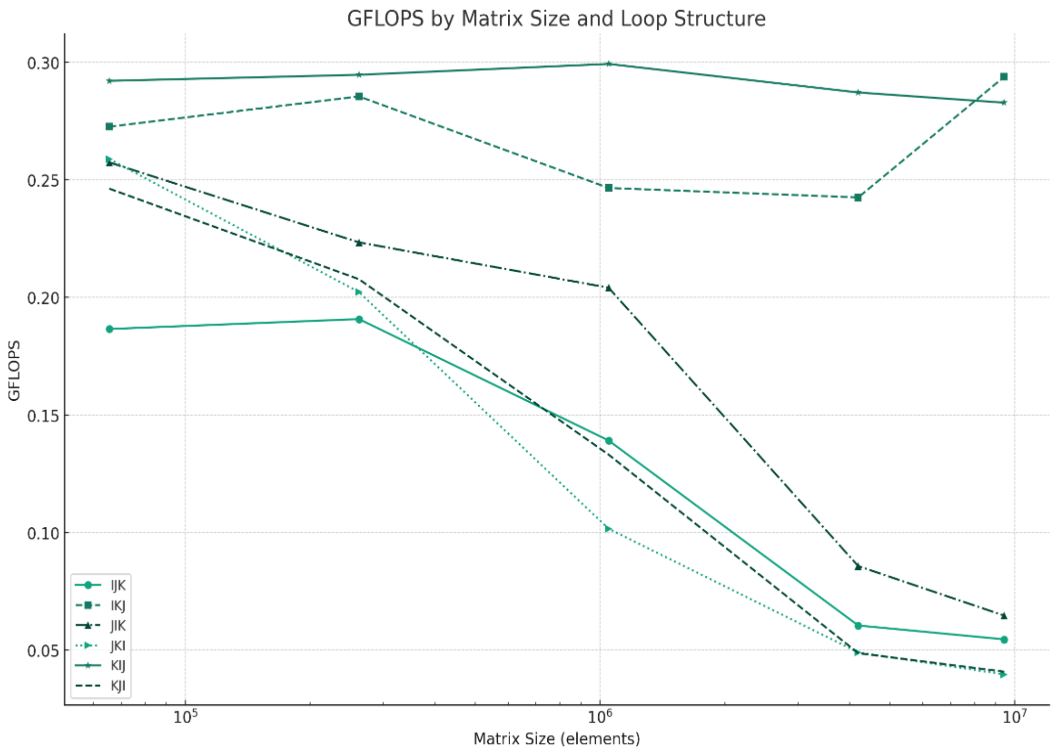


Figure 4.1: **Loop Structures**: Comparison of GFLOPS across six loop structures for matrix multiplication, using C++. The graph illustrates the performance of each loop structure for varying matrix sizes.

According to the results shown in Figure 4.1, the GFLOPS differences among the six loop structures are not significant when the matrix size is small. However, as the matrix size increases, the GFLOPS for loop structures IKJ and KIJ remain stable, while the GFLOPS for the other loop structures gradually decrease, indicating a decline in performance. These results are consistent with the conclusions presented in the Background section. Therefore, when writing code, programmers should consider the data's storage format in memory and choose the appropriate loop structure, especially when dealing with large-scale data, to significantly enhance computational efficiency. In my implementations of matrix multiplication and LU Factorization, I have adopted the IKJ or KIJ loop structures.

4.2 Block Matrix Multiplication Analysis

I have implemented a block algorithm for matrix multiplication and conducted experiments to determine the optimal block size for maximizing the algorithm's performance. The reading speed of the L1 cache is 1-4 clock cycles, the L2 cache is 8-12 clock cycles, the L3 cache is 20-30 clock cycles, and the main memory (DRAM) takes 200-400 clock cycles to read [17]. To facilitate faster data retrieval, my strategy involves fitting data blocks within the L1 cache. For my computer, the L1 cache size is 256KB. Assuming each element is a float occupying 4 bytes, the L1 cache can theoretically hold $(256 \times 1024)/4 = 65,536$ elements. With three blocks needing to fit into the cache, each block would contain about $\frac{65,536}{3} \approx 21,845$ elements. Taking the square root gives a block edge length of approximately 148.

Following this, I conducted experiments using matrices of sizes 2048x2048 and 1024x1024 as test data. The horizontal axis represented the block size, starting from 1 and increasing incrementally, while the vertical axis measured the speedup, defined as the computation time of naive matrix multiplication divided by the block matrix multiplication computation time.

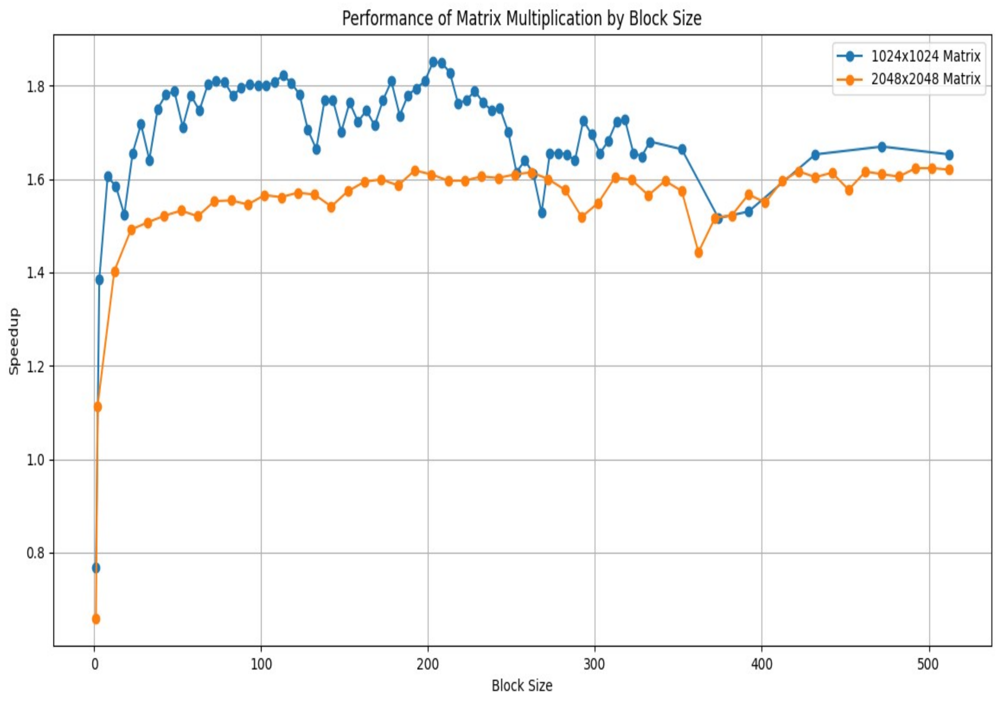


Figure 4.2: Performance Speedup of Block Matrix Multiplication by Block Size

As seen in Figure 4.2, when the block size is reduced to 1, the algorithm degenerates to naive matrix multiplication, resulting in a performance decrease. This is due to the innermost loop of my block algorithm requiring additional calls to the `std::min` function for boundary checks, which incurs some overhead. For matrices of size 1024x1024, the average speedup is approximately 1.7. For 2048x2048 matrices, the average speedup is slightly less than that for 1024x1024 matrices. The graph indicates that for both sizes of matrices, the speedup enters a stable range around a block size of 60, with slight variation as block size increases. This result shows a discrepancy from my calculated optimal block size of 148. However, as depicted in the graph, even when the block size exceeds the theoretical capacity of the L1 cache, the speedup does not decline and consistently remains around 1.6. I believe this may be because the experiments were conducted on my personal computer; although I attempted to close as many software and processes as possible during testing, some non-closable processes might still occupy a portion of the L1 or even L2 cache, hence block data might actually be cached in L2 or even L3 cache. Overall, the block matrix multiplication algorithm improves about 1.6-1.8 times compared to naive matrix multiplication.

4.3 Serial and Parallel Block Matrix Multiplication Performance Analysis

Not all compilers support C++17's parallel execution policy. For my experiments, I used the NVIDIA HPC C++ compiler:

- Compiler version: `nvc++ 23.11-0 64-bit target on x86-64 Linux -tp haswell`
- Compiler flags: `nvc++ -std=c++17 -Ofast -stdpar=multicore`

The `-Ofast` flag enables all optimizations, and `-stdpar=multicore` specifies compilation for multicore CPU execution. By default, `nvc++` uses `-stdpar=gpu` to compile for parallel execution on a GPU [21].

Due to the limited GPU capabilities of my computer, I compiled for CPU execution. As mentioned in the background, programmers cannot explicitly specify or know how many threads the execution policy will use in parallel execution. To investigate how `std::execution::par` manages threads, I conducted the following experiment:

Before entering the parallel algorithm, I used the following code to get the number of available threads:

```
unsigned int n = std::thread::hardware_concurrency();
```

I then created a hashmap to record thread IDs. During the parallel section, I retrieved the current thread ID with

```
std::this_thread::get_id();
```

and added it to the hashmap. Finally, I compared whether the number of thread IDs in the hashmap matched n. The results showed that they were the same. Additionally, I printed thread IDs within the parallel loop to compare. The experimental results indicated that the number of threads available before executing the parallel algorithm was indeed the number used by the parallel algorithm. Moreover, with my CPU having 8 threads, multiple tests confirmed that the execution policy actively utilized all 8 threads during the parallel loop.

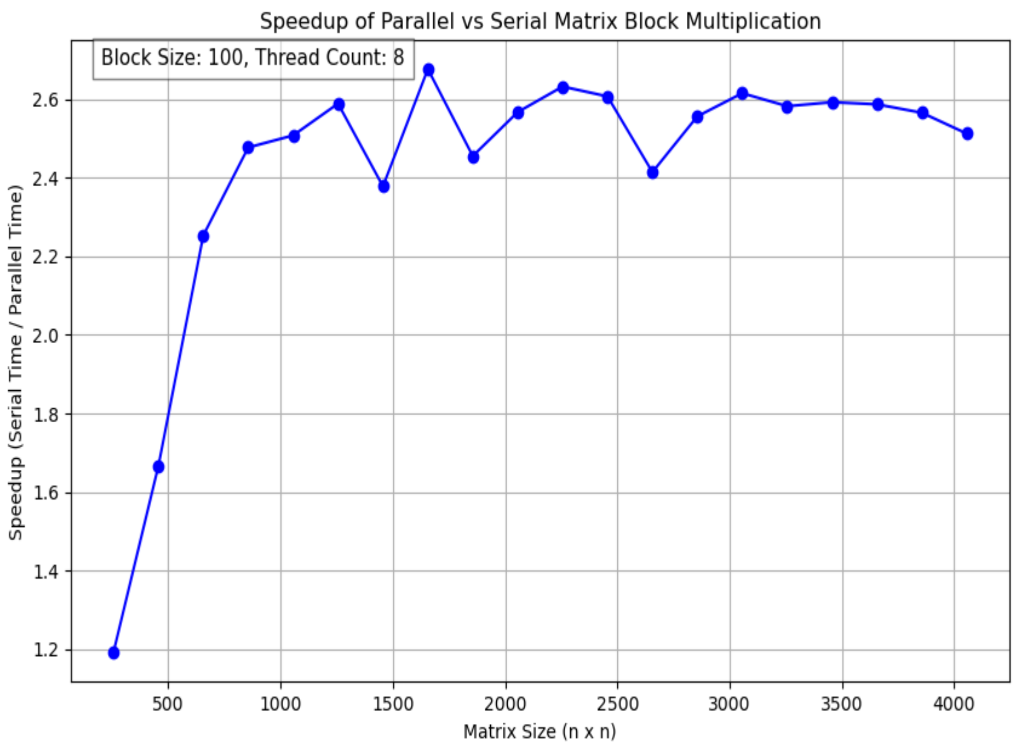


Figure 4.3: Parallel Matrix Multiplication performance

The graph depicted in Figure 4.3 demonstrates that using the parallel execution policy for matrix multiplication significantly enhances the algorithm's performance. As the matrix size increases, the speedup also increases for smaller matrices. However, when the matrix size reaches 1500x1500, the speedup stabilizes at around 2.5 times, with a slight downward trend thereafter.

4.4 LU Factorization Performance Analysis

Figure 4.4 presents a performance comparison between serial and parallel LU factorization. For LU factorization, the speedup increases as the matrix size increases for smaller matrices. The speedup reaches its peak when the matrix size is approximately 6000x6000. After reaching this peak, the speedup begins to decline gradually. This trend highlights the scalability limits of the parallel execution in handling large matrices.

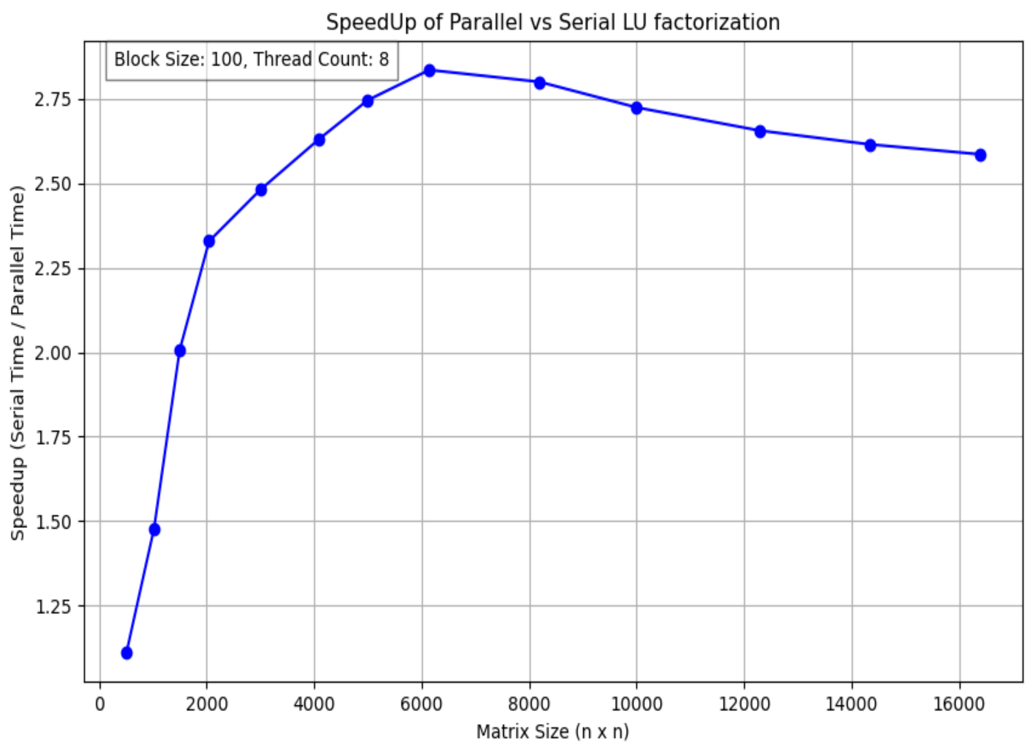


Figure 4.4: Parallel LU Factorization Performance Analysis

4.5 LU Factorization Performance Comparison with the Eigen Library

Figure 4.5 shows the serial comparison chart, and Figure 4.6 displays the parallel comparison chart. From both charts, it is evident that for both serial and parallel LU factorization, the runtime of my LU factorization implementation consistently remains lower than that of the Eigen library. Moreover, for serial LU factorization, my implementation significantly outperforms the Eigen library's performance when dealing with larger matrices. This demonstrates the effectiveness of the optimizations I applied in my implementation, especially in handling substantial computational loads efficiently in both serial and parallel contexts.

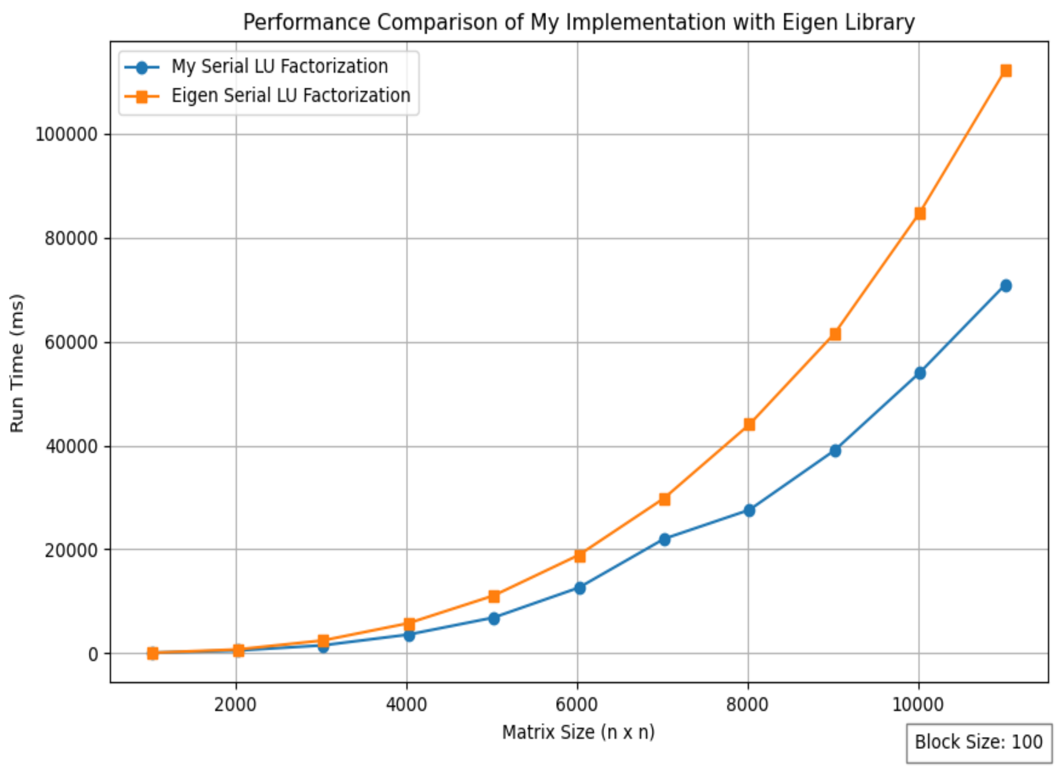


Figure 4.5: Performance Comparison of my serial LU factorization with Eigen Library

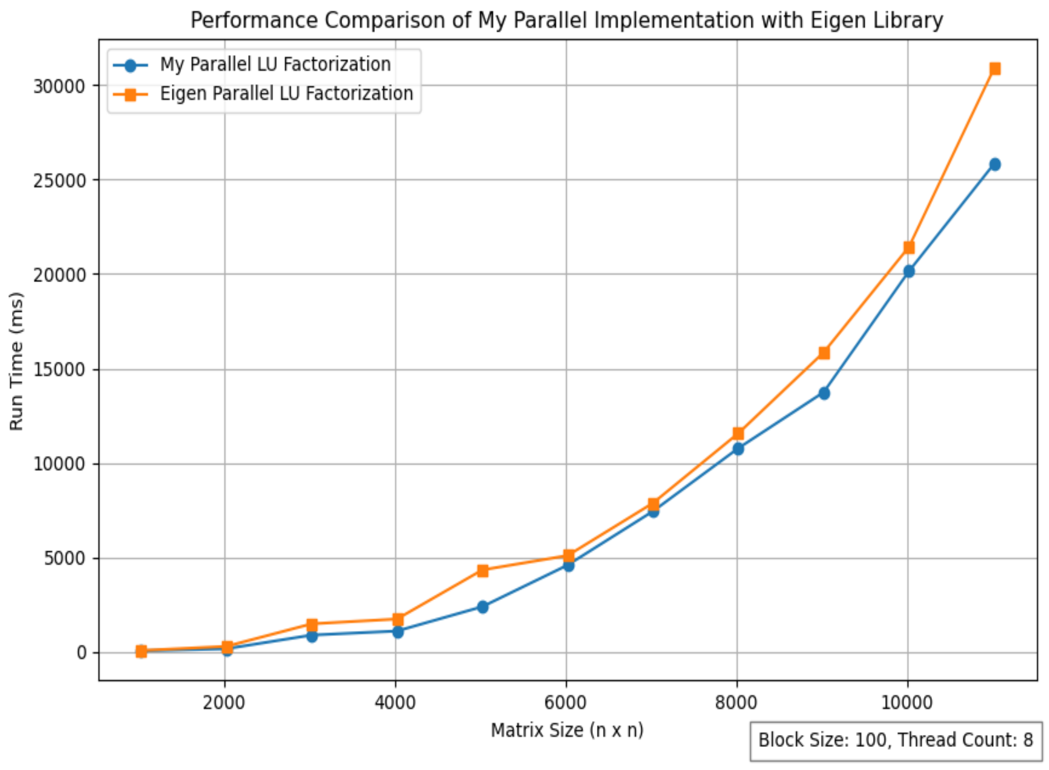


Figure 4.6: Performance Comparison of my parallel LU factorization with Eigen Library

4.6 Sparse Matrix Performance Analysis

I will analyze the performance of my two sparse matrix classes with a structured approach. Firstly, I'll generate a sparse matrix with a custom sparsity level. Next, I'll initialize objects using two different sparse matrix classes. Finally, I'll conduct a comparison of the performance of addition operations across these implementations.

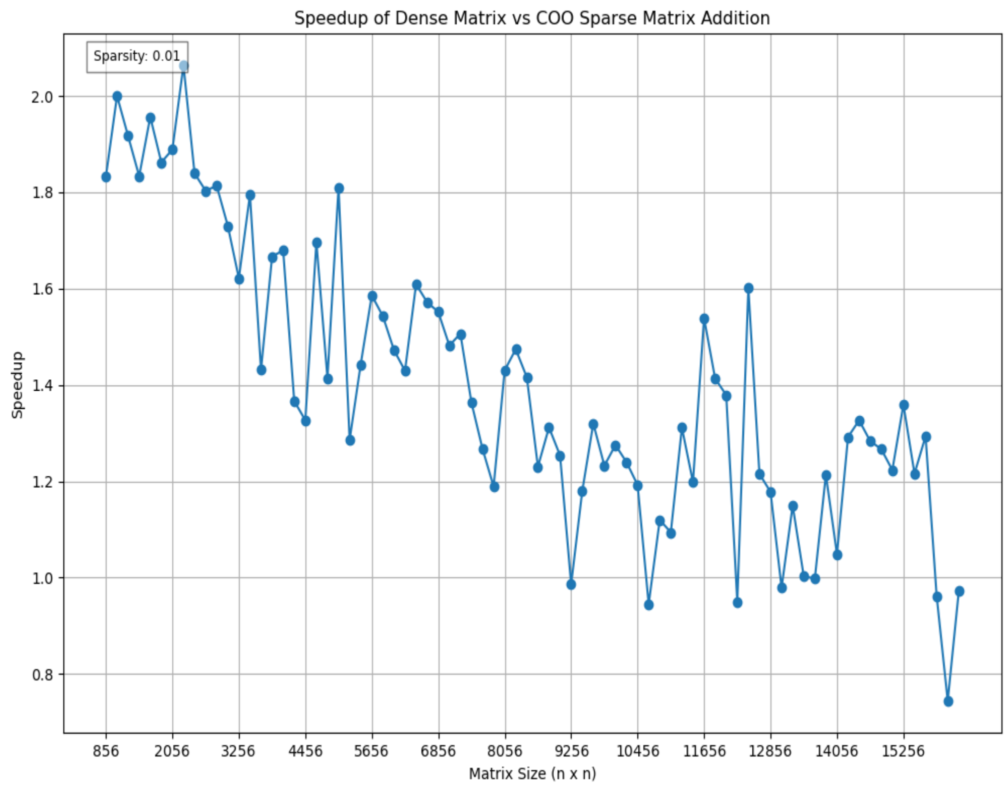


Figure 4.7: Performance Comparison of Dense matrix and COO Sparse matrix addition operation

For Figure 4.7, it is observable that using the COO (Coordinate List) storage format can enhance performance to a certain extent. However, as the matrix size increases, the performance gradually declines, and when the matrix size exceeds 15500x15500, the performance becomes slower than using dense matrix storage. This decrease is due to my implementation of the addition operation in the COO class using `std::unordered_map` to merge data. To avoid introducing external libraries, I implemented a simple hash function myself, which may have a higher collision rate than those found in external libraries. As the matrix size increases, the number of elements in the hash map also increases, leading to more frequent hash collisions. `std::unordered_map` uses chaining to resolve hash collisions, so as the number of collisions increases, the length of the chains grows, leading to longer search times in the hash table. The worst-case time complexity for the lookup operation lookup(x) in a hash table is O(length of chain containing x) [10].

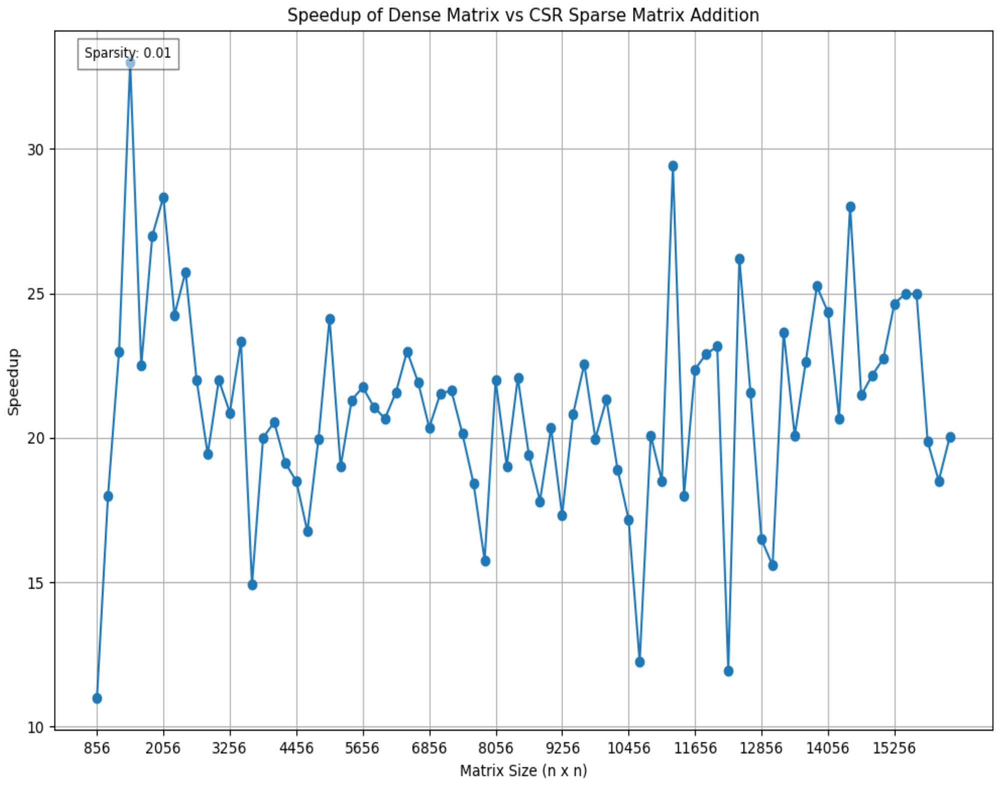


Figure 4.8: Performance Comparison of Dense matrix and CSR Sparse matrix addition operation

For Figure 4.8, the CSR (Compressed Sparse Row) storage format significantly outperforms the COO format, with a speedup ratio reaching up to 33, and an average speedup of 21, far surpassing that of COO. This is because the row pointers in CSR allow quick access to any row's elements without traversing all elements.

In summary, although the COO storage format uses significantly less memory compared to dense matrices, its performance in addition operations is not satisfactory. The CSR storage format, however, shows clear advantages with its low memory usage while significantly outperforming dense matrices in addition operations.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

In this project, I successfully developed a powerful linear algebra library. My accomplishments include:

1. I achieved scalability and user-friendliness in the library by utilizing object-oriented design principles.

2. I implemented efficient linear algebra functions, such as matrix multiplication and LU factorization, enhanced by block algorithm.

3. I leveraged new features of C++17, such as parallel execution policy, to create parallel versions of these algorithms.

4. I developed sparse matrix classes, implementing both COO and CSR data structures, and implemented their addition operations.

Extensive testing was conducted on my library. All library functions passed correctness tests, and performance analysis of the linear algebra package showed excellent results. My implementation of LU factorization was compared with the open-source library Eigen, demonstrating superior performance. Thus, I consider this project successful.

5.2 Future Work

1. Performance tests were confined to my personal computer with limited cores and CPU-only testing; GPU parallelism was not supported. Future tests could be run on various CPU and GPU models, such as on the Isambard supercomputer, using professional performance analysis tools like Intel Advisor to generate roofline models and further analyze memory bandwidth utilization and performance bottlenecks.

2. While running tests on my computer, certain uncontrollable background processes might have consumed CPU resources and cache space, potentially distorting performance test results and complicating the validation of the theoretical optimal block size for the block algorithm. Future work could involve using high-performance computers to allocate dedicated CPU resources for processes and employing tools like `perf` to analyze memory access patterns and cache hit rates.

3. Current implementations of the sparse matrix classes, COO and CSR, only support addition operations. Implementing their LU factorization is a necessary next step.

4. The addition operation for the COO implementation of sparse matrices underperforms, possibly requiring more advanced hashing functions to reduce hash collisions, or more efficient addition algorithms.

5. Additional data structures could be developed for sparse matrices to accommodate different architectures, such as ELL and SELL-P for GPUs [13].

6. The library currently does not support operations with complex numbers. Complex number operations could be developed to enhance the library's applicability in fields requiring complex numerical computations.
7. More advanced linear algebra functions, such as QR factorization and Cholesky factorization, could be implemented to enhance the library's offerings.

Bibliography

[1] Std::Execution::Parallel_policy - cppreference.com. https://en.cppreference.com/w/cpp/algorithm/execution_policy_tag_t. Accessed: May 04, 2024.

[2] Algorithms library - cppreference.com. https://en.cppreference.com/w/cpp/algorithm. Accessed: May 04, 2024.

[3] Eigen: Eigen and multi-threading. https://eigen.tuxfamily.org/dox/TopicMultiThreading.html. Accessed May 04, 2024.

[4] Example: Lu factorization — task-based parallelism in scientific computing documentation. https://hpc2n.github.io/Task-based-parallelism/branch/spring2021/task-basics-lu/. Accessed: May 04, 2024.

[5] LAPACK – linear algebra package. https://www.netlib.org/lapack/. Accessed: 2024-05-05.

[6] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, J. Langou, H. Ltaief, and S. Tomov. LU factorization for accelerator-based systems. In *Proceedings of the 9th IEEE/ACS International Conference on Computer Systems and Applications (AICCSA)*, pages 217–224, Sharm El Sheikh, Egypt, 2011. doi:10.1109/AICCSA.2011.6126599.

[7] E. Anderson et al. LAPACK: A portable linear algebra library for high-performance computers. *Conference on High-Performance Computing*, pages 2–11, Oct 1990. doi:10.1109/SUPERC.1990.129995.

[8] R. F. Barrett, T. H. F. Chan, E. F. D'Azevedo, E. F. Jaeger, K. Wong, and R. Y. Wong. Complex version of high performance computing LINPACK benchmark (HPL). *Concurrency and Computation: Practice and Experience*, pages n/a–n/a, 2009. doi:10.1002/cpe.1476.

[9] I. Choudhury, B. Wang, P. Rosen, and V. Pascucci. Topological analysis and visualization of cyclical behavior in memory reference traces. In *Proceedings of the IEEE Pacific Visualization Symposium*, Feb 2012. doi:10.1109/pacificvis.2012.6183557.

[10] R. Clifford. Lecture 2. Advanced Algorithm Course, University of Bristol, 2024. Course lecture notes.

[11] Oracle Corporation. Generic types (the java(tm) tutorials > learning the java language > generics (updated)). https://docs.oracle.com/javase/tutorial/java/generics/types.html. Accessed: 2024-05-05.

[12] J. Dongarra, Piotr Luszczek, and A. Petitet. The LINPACK benchmark: Past, present and future. *Concurrency and Computation: Practice and Experience*, 15(9):803–820, Aug 2003. doi:10.1002/cpe.728.

[13] M. Al Farhan, A. Abdelfattah, S. Tomov, et al. MAGMA templates for scalable linear algebra on emerging architectures. *The International Journal of High Performance Computing Applications*, 34(6):645–658, 2020. doi:10.1177/1094342020938421.

[14] K. He, Sheldon, H. Wang, and G. Shi. GPU-Accelerated Parallel Sparse LU Factorization Method for Fast Circuit Analysis. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(3):1140–1150, Mar 2016. doi:10.1109/TVLSI.2015.2421287.

[15] M. Hoemmen, D.S Hollman, and C. Trott. P1674r1: Evolving a standard c++ linear algebra library from the blas. 2022. URL: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p1674r1.pdf.

[16] Chuck L Lawson, Richard J. Hanson, David R Kincaid, and Fred T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Transactions on Mathematical Software (TOMS)*, 5(3):308–323, 1979. doi:/10.1145/355841.355847.

[17] S. McIntosh-Smith. Lecture 3. High Performance Computing Course, 2024. Course lecture notes.

[18] J. Michalakes. *HPC for Weather Forecasting*, pages 297–323. Springer, 2020. doi:10.1007/978-3-030-43736-7_10.

[19] Netlib. LINPACK. Online source, 2024. Accessed May 04, 2024. URL: https://www.netlib.org/linpack/.

[20] Netlib. Lu factorization. Online image, 2024. URL: https://netlib.org/utk/papers/factor/node7.html.

[21] NVIDIA. Nvidia hpc compilers user's guide. https://docs.nvidia.com/hpc-sdk/compilers/hpc-compilers-user-guide/index.html#stdpar-use. Accessed: May 04, 2024.

[22] S. Ristov, M. Gusev, and G. Velkoski. Optimal block size for matrix multiplication using blocking. *Proceedings of the 2014 IEEE International Conference on MIPRO*, May 2014. doi:10.1109/MIPRO.2014.6859580.

[23] G.W. Stewart. Research, development, and linpack. In John R. Rice, editor, *Mathematical Software*, pages 1–14. Academic Press, 1977. doi:10.1016/B978-0-12-587260-7.50005-4.

[24] C. Stylianou and M. Weiland. Exploiting dynamic sparse matrices for performance portable linear algebra operations. *arXiv*, Nov 2022. doi:10.1109/p3hpc56579.2022.00010.

[25] M. Stypinski. Why loops do matter – a story of matrix multiplication, cache access patterns and java. Medium, Sep 2023. Accessed: May 04, 2024.

[26] Wikipedia. Lu decomposition. https://en.wikipedia.org/wiki/LU_decomposition#Doolittle_algorithm, Mar 2024. Accessed: May 04, 2024.

[27] V. Zee and Robert. BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Transactions on Mathematical Software*, 41(3):1–33, Jun 2015. doi:10.1145/2764454.

Appendix A

Code Repository

All code for the project can be found on the GitHub repository at https://github.com/Xinran1205/HPC-Project

Appendix B

AI Prompts

I used ChatGPT to debug code and replace some words in this dissertation with academic vocabulary.