IA-Chol: Input-Aware Cholesky Decomposition on CPU and GPU

Jixiao Deng*
National University of Defense Technology
Laboratory of Digitizing Software for Frontier Equipment
Changsha, Hunan, China
National University of Defense Technology
National Key Laboratory of Parallel and Distributed Computing
Changsha, Hunan, China
National University of Defense Technology
College of Computer Science and Technology
Changsha, Hunan, China
1169068630@qq.com

Qinglin Wang*†
National University of Defense Technology
Laboratory of Digitizing Software for Frontier Equipment
Changsha, Hunan, China
National University of Defense Technology
National Key Laboratory of Parallel and Distributed Computing
Changsha, Hunan, China
National University of Defense Technology
College of Computer Science and Technology
Changsha, Hunan, China
wangqinglin@nudt.edu.cn

Lin Chen
National University of Defense Technology
Laboratory of Digitizing Software for Frontier Equipment
Changsha, Hunan, China
National University of Defense Technology
National Key Laboratory of Parallel and Distributed Computing
Changsha, Hunan, China
National University of Defense Technology
College of Computer Science and Technology
Changsha, Hunan, China
chenlin1080@outlook.com

Tun Li
National University of Defense Technology
College of Computer Science and Technology
Changsha, Hunan, China
tunli@nudt.edu.cn

Bo Yang
National University of Defense Technology
Laboratory of Digitizing Software for Frontier Equipment
Changsha, Hunan, China
National University of Defense Technology
National Key Laboratory of Parallel and Distributed Computing
Changsha, Hunan, China
National University of Defense Technology
College of Computer Science and Technology
Changsha, Hunan, China
yb@nudt.edu.cn

Xinhai Chen
National University of Defense Technology
Laboratory of Digitizing Software for Frontier Equipment
Changsha, Hunan, China
National University of Defense Technology
National Key Laboratory of Parallel and Distributed Computing
Changsha, Hunan, China
National University of Defense Technology
College of Computer Science and Technology
Changsha, Hunan, China
chenxinhai16@nudt.edu.cn

Jie Liu
National University Of Defense Technology
Laboratory of Digitizing Software for Frontier Equipment
Changsha, Hunan, China
National University Of Defense Technology
National Key Laboratory of Parallel and Distributed Computing
Changsha, Hunan, China
National University Of Defense Technology
College of Computer Science and Technology
Changsha, Hunan, China
liujie@nudt.edu.cn

Abstract

Cholesky decomposition plays a pivotal role in numerical linear algebra, providing an efficient approach for solving linear systems and computing the determinant of positive definite matrices. However, despite advancements in computational power and cache capacity, optimizing performance for small and medium-sized matrices remains challenging, with the potential of modern computing resources underutilized. To address this, we propose, for the first time, an innovative operator fusion scheme to optimize the Cholesky decomposition

*Both authors contributed equally to this research.
†Corresponding author

ICS '25, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-1537-2/25/06
https://doi.org/10.1145/3721145.3725756

algorithm, achieving remarkable results. Nevertheless, this has led to a new problem: the optimal tile size has undergone significant changes, which has resulted in poor performance of traditional tile size prediction methods. To tackle this issue, we developed a novel tile size prediction method tailored for the fused-operator Cholesky algorithm. This method automatically determines the optimal tile size based on the input matrix size and is applicable to both CPU and GPU. We name this method IA-Chol. Experimental results demonstrate that IA-Chol delivers outstanding performance on both CPU and GPU platforms, effectively mitigating sensitivity issues while significantly outperforming many state-of-the-art libraries. For instance, on the NVIDIA A100 GPU, IA-Chol achieves an efficiency of 85.1%, substantially surpassing cuSOLVER's 75.8%. Moreover, IA-Chol not only exhibits significant performance improvements for medium-sized matrices but also maintains exceptional performance for matrices larger than 20,000 on GPUs. Additionally, IA-Chol outperforms other tile size prediction methods, supported by in-depth theoretical analysis and architectural insights explaining the root causes of these performance gains.

CCS Concepts

• **Computing methodologies** → *Shared memory algorithms*; • **Mathematics of computing** → *Solvers*.

Keywords

Cholesky decomposition, Input-aware adaptive-size tile, Operator fusion, GPU, CPU

ACM Reference Format:

Jixiao Deng, Qinglin Wang, Lin Chen, Tun Li, Bo Yang, Xinhai Chen, and Jie Liu. 2025. IA-Chol: Input-Aware Cholesky Decomposition on CPU and GPU. In *2025 International Conference on Supercomputing (ICS '25), June 08–11, 2025, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3721145.3725756

1 Introduction

Cholesky factorization is a well-known method for solving linear systems involving symmetric positive-definite matrices[10]. It is widely used in fields such as numerical linear algebra, scientific computing[12, 24], machine learning[17, 22], engineering and physical simulations[31], computational finance[27], image processing, computer vision[15], signal processing[23], and weather forecasting[4]. These applications often require performing Cholesky factorization on matrices ranging from hundreds to thousands, or even larger in size. As such, accelerating the Cholesky factorization is of significant practical importance.

Challenge. With the rapid advancement of multi-core systems and the significant increase in the number of cores and cache capacity, the aforementioned issue has hindered the full utilization of computational performance for medium-sized matrices (ranging from hundreds to thousands or more). As shown in Figure 1, even for medium-sized matrices with dimensions in the thousands, many state-of-the-art operator libraries still exhibit a considerable gap between the performance of their Cholesky decomposition and the actual peak performance of Cholesky, failing to fully leverage the computational power of the hardware.

Solution. To address this issue, we propose an operator fusion scheme tailored for Cholesky decomposition. This scheme theoretically reduces $\frac{1}{12}N^3 + \frac{3}{2}N^2$ cache accesses, with detailed explanations provided in Chapter 3. As illustrated in Figure 1, our approach(IA-Chol) significantly enhances the performance of medium-sized Cholesky decomposition, enabling it to better exploit the computational capabilities of the hardware.

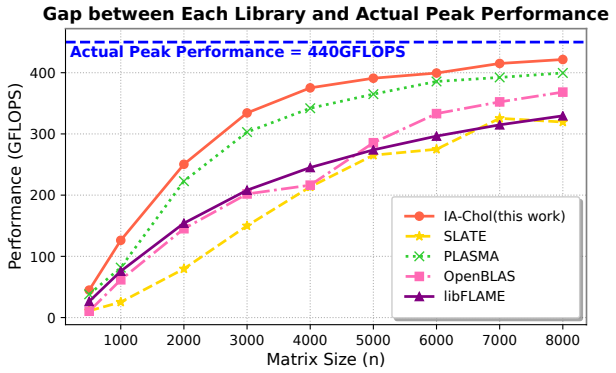

Figure 1: This figure demonstrates the gap between the Cholesky decomposition of our work and other advanced libraries on medium-sized matrices, compared to the actual peak performance. This is an example using performance on the Intel(R) Xeon(R) Silver 4210R. The tile size for SLATE is set to 256, for PLASMA to 192, and for libflame to 128, as these values achieve the best average performance within the range of 1000 to 10000.

New Challenge. Although significant performance improvements have been achieved, a new challenge has emerged: the optimal tile size for Cholesky decomposition after operator fusion is smaller than before. This indicates that smaller tile sizes can deliver better performance, and the parallelism of the algorithm has been enhanced. However, this also means that the selection of tile size has become more critical, and the previous approach is no longer effective in identifying the optimal tile size for peak performance. Figure 2 shows this change.

New Solution. To address the aforementioned problem, we propose a method for predicting tile sizes specifically

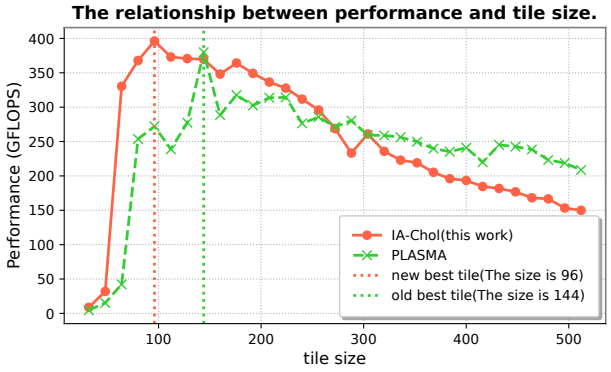

Figure 2: This figure shows the relationship between performance and tile size on the Intel(R) Xeon(R) Silver 4210R for n=5000. It can be observed that the optimal tile size has shifted noticeably, with the optimal tile size becoming significantly smaller. This trend is also observed on other machines. The reason for comparing with PLASMA is that the code of IA-Chol on CPU is modified based on PLASMA (this will be discussed in detail in Section 1 of Chapter 3 in the Overview).

for the fused Cholesky decomposition. Compared to several state-of-the-art operator libraries, our method offers two key advantages: first, it predicts the optimal tile size based on both the input matrix size and the performance parameters of the computing platform, whereas existing methods primarily rely on performance parameters alone; second, our method is applicable to both CPU and GPU platforms. We collectively refer to the operator fusion scheme and the tile size prediction method as IA-Chol.

Evaluation. We implemented IA-Chol on both CPU and GPU and conducted experiments on various devices, comparing its performance with several state-of-the-art libraries. Surprisingly, not only was the issue of insufficient performance for medium-sized matrices effectively addressed, but significant performance improvements were also achieved for large matrices (dimensions exceeding 20,000) on GPUs. For instance, on the A100 GPU, IA-Chol achieved an efficiency of 85.1% for Cholesky decomposition, compared to only 75.8% achieved by cuSOLVER. Our work will be open-sourced on GitHub.

In summary, we offer the following contributions:

- IA-Chol proposes an operator fusion scheme for Cholesky decomposition, which theoretically reduces memory access overhead by $\frac{1}{12}N^3 + \frac{3}{2}N^2$.
- To address the issue of optimal tile size offset after operator fusion, IA-Chol introduces an automatic tile size adjustment method based on matrix size and computing platform parameters. This method is applicable to both CPU and GPU platforms, and outperforms other tile size prediction approaches in terms of performance.
- IA-Chol not only significantly enhances the performance of medium-sized Cholesky decomposition but also improves the performance of large matrices on GPUs. The performance of IA-Chol surpasses that of many current state-of-the-art operator libraries.

2 Background

Cholesky decomposition is a numerical method used for factorizing a symmetric positive definite matrix A into the product of a lower triangular matrix L and its transpose, such that $A = LL^T$. As the size of the matrix increases, the computational cost and memory requirements also rise significantly, which can lead to performance bottlenecks on modern architectures.Algorithm 1 presents the pseudocode for the Cholesky decomposition, detailing its execution process. Figure 3 shows the Cholesky decomposition process of a 5×5 matrix.

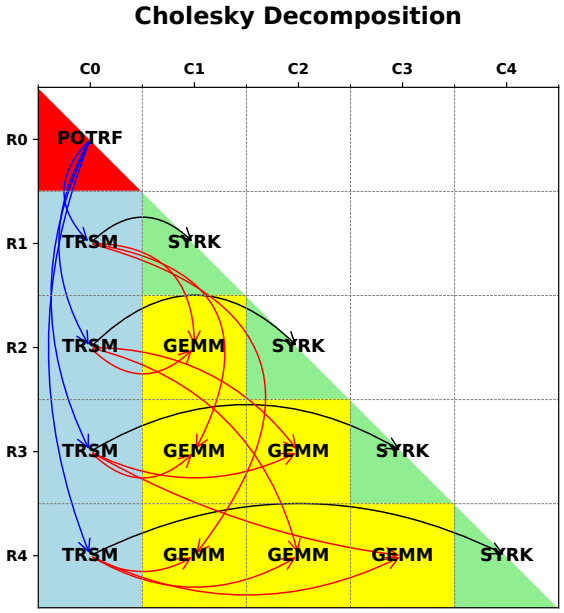

Figure 3: This figure shows the process of Cholesky decomposition of a 5×5 positive definite matrix, with arrows indicating the direction of data flow.

The Cholesky decomposition can be summarized as the following steps:

- **POTRF**: The Cholesky factorization of the diagonal tile.
- **TRSM**: Solving triangular systems to update the sub-diagonal elements of the tiles.

- **SYRK**: Updating the diagonal tiles to ensure symmetry and maintain positive definiteness.
- **GEMM**: Performing general matrix-matrix multiplications to compute the updates of the remaining tiles.

Algorithm 1 Cholesky Decomposition

1: **Input:** Positive definite matrix A of size $N \times N$, tile size $n_b \times n_b$
2: SplitTile(A, n_b)
3: $n \leftarrow \lceil N/n_b \rceil$
4: **for** $k = 0$ to $n-1$ **in parallel do**
5: $A[k,k] \leftarrow \text{POTRF}(A[k,k])$
6: **for** $i = k+1$ to $n-1$ **in parallel do**
7: $A[i,k] \leftarrow \text{TRSM}(A[k,k], A[i,k])$
8: **end for**
9: **for** $j = k+1$ to $n-1$ **in parallel do**
10: $A[j,j] \leftarrow \text{SYRK}(A[j,k], A[j,j])$
11: **for** $i = j+1$ to $n-1$ **in parallel do**
12: $A[i,j] \leftarrow \text{GEMM}(A[i,k], A[j,k])$
13: **end for**
14: **end for**
15: **end for**
16: **Return** A

3 Method

3.1 Overview

Our implementation is based on adaptations of PLASMA and SLATE's Cholesky decomposition. Specifically, PLASMA's code is used for CPUs, while SLATE's code is employed for GPUs. Why not use a unified codebase? The reason lies in their respective strengths: on CPUs, PLASMA demonstrates better performance, features mature task scheduling, and has a simpler codebase that is easier to modify. However, its major limitation is the lack of GPU support. For GPUs, SLATE was chosen because Cholesky decomposition on GPUs typically involves large matrices, and SLATE is specifically designed for such cases. Although SLATE can also run on CPUs, its task scheduling overhead is relatively high when dealing with small- to medium-sized matrices, leaving little room for performance improvement. Since PLASMA and SLATE have similar implementations of Cholesky decomposition, our IA-Chol can be seamlessly ported to both.

Algorithm 2 represents the pseudocode for IA-Chol and consists of three main parts.

The first step is **adaptive sizing**, which determines the optimal tile size based on the input matrix size and the machine's performance parameters (corresponding to Line 2 of Algorithm 2, with details elaborated in Section 3.3).

The second step involves partitioning the input matrix into multiple tiles, a process automatically handled by PLASMA or SLATE (corresponding to Line 3 of Algorithm 2).

The third step introduces our **operator fusion scheme**, specifically designed for Cholesky decomposition (corresponding to Lines 5 to 26 of Algorithm 2).

Algorithm 2 IA-Chol

1: **Input:** Positive definite matrix A of size $N \times N$
2: $n_b \leftarrow$ Input-AwareAdaptiveSize(N, CacheSize, CoreNum, Peak)
3: SplitTile(A, n_b)
4: $n \leftarrow \lceil N/n_b \rceil$
5: **for** $k = 0$ to $n-1$ step 2 **in parallel do**
6: $A[k,k] \leftarrow \text{POTRF}(A[k,k])$
7: $A[k+1,k] \leftarrow \text{TRSM}(A[k,k], A[k+1,k])$
8: $A[k+1,k+1] \leftarrow \text{SYRK}(A[k+1,k], A[k+1,k+1])$
9: $A[k+1,k+1] \leftarrow \text{POTRF}(A[k+1,k+1])$
10: **for** $i = k+2$ to $n-1$ **in parallel do**
11: $A[i,k] \leftarrow \text{TRSM}(A[k,k], A[i,k])$
12: $A[i,i] \leftarrow \text{SYRK}(A[i,k], A[i,i])$
13: **end for**
14: **for** $j = k+2$ to $n-1$ **in parallel do**
15: $A[j,k+1] \leftarrow \text{GEMM}(A[j,k], A[k+1,k])$
16: $A[j,k+1] \leftarrow \text{TRSM}(A[k+1,k+1], A[j,k+1])$
17: $A[j,j] \leftarrow \text{SYRK}(A[j,k+1], A[j,j])$
18: **end for**
19: **for** $j = k+2$ to $n-1$ **in parallel do**
20: **for** $i = j+1$ to $n-1$ **in parallel do**
21: $A[i,j] \leftarrow \text{GEMM}(A[i,k], A[j,k])$
22: $A[i,j] \leftarrow \text{GEMM}(A[i,k+1], A[j,k+1])$
23: **end for**
24: **end for**
25: **end for**
26: **Return** A

3.2 Operator Fusion

In the following sections of the paper, we will use N to represent the size of the input matrix, n_b to denote the tile size, and n to represent the size of the matrix after being tiled.

Our operator fusion scheme is inspired by the works of Erin Carson in 2018 and 2022[5, 6]. First, we set the lookahead to 1 in the Tiled Cholesky algorithm, meaning that in each iteration of the for loop in Algorithm 1, we execute one step ahead, effectively doubling the for loop's step size. Then, we applied fusion to the process. Lines 5 to 26 of Algorithm 2 represent the pseudocode for this process. Specifically, there are four main fusion strategies as follows:

(1) **PTSP (Lines 6 to 9 of Algorithm 2):** In Algorithm 1, we will expand the loop and merge POTRF($A[k,k]$),

TRSM(A[k,k], A[k+1,k]), SYRK(A[k+1,k], A[k+1,k+1]), and POTRF(A[k+1,k+1]) to form 6-9 lines in Algorithm 2. It reduces memory operations compared to the Cholesky(Algorithm 1) by saving $2n_b^2$ memory accesses. With k looping $\frac{n}{2}$ times, this reduces $nn_b^2 = Nn_b$ memory accesses.

(2) **TRSM-SYRK (Lines 10 to 13 of Algorithm 2):** In Algorithm 1, we merge the TRSM operation in the seventh row with the SYRK operation in the tenth row to form a new TRSM-SYRK operation. As shown in lines 10 to 12 of Algorithm 2. The for loops of both operations access the same indices and involve $A[i,k]$. By fusing them, after TRSM completes, $A[i,k]$ remains in the cache, reducing additional read and write operations. During the k-th iteration, fusing TRSM($A[k,k], A[i,k]$) and SYRK($A[i,k], A[i,i]$) reduces the write of $A[i,k]$ in TRSM and the read of $A[i,k]$ in SYRK, saving $2n_b^2$ memory accesses. With $(n-k-1)$ tiles, the total reduction is $2n_b^2 \times (n-k-1)$. Summing over k, the reduction is $\sum_{k=0}^{n-1} 2n_b^2 \times (n-k-1) = n_b^2 n(n-1) \approx N^2 - Nn_b$. Thus, operator fusion in this step reduces $N^2 - Nn_b$ memory accesses, which is significant for small matrices.

(3) **pannelGEMM-TRSM-SYRK (Lines 14 to 18 of Algorithm 2):** Then we first update the data in the second column on the left, GEMM($A[i,k], A[k+1,k]$), TRSM(A[k+1,k+1], A[i,k+1]), SYRK(A[i,k], A[i,i]). We call this step pannelGEMM-TRSM-SYRK, as shown in rows 14 to 18 of Algorithm 3. In line 14-18 of algorithm 3. Fusion of panelGEMM(A[i,k], A[k+1,k]), TRSM(A[k+1,k+1], A[i,k+1]), and SYRK(A[i,k+1], A[i,i]) reduces memory accesses by $2n_b^2$. With i looping $n-k-2$ times, this results in $2n_b^2(n-k-2)$ memory accesses saved. Summing over k with $k = 0, 2, 4, \ldots, n-1$, the total reduction is $\sum_{l=0}^{(n-1)/2} 2n_b^2(n-2l-2) \approx n_b^2 n^2/2 = N^2/2$.

(4) **trailingGEMM (Lines 19 to 24 of Algorithm 2):** Next,we update the remaining data, merging GEMM(A[i, k], A[j, k]) and GEMM($A[i,k+1], A[j,k+1]$),namely trailingGEMM, in lines 19 to 24 of Algorithm 2. From line 19 to line 24 of Algorithm 2, fusion of trailingGEMM reduces memory accesses by fusing two GEMM operations, saving one write and one read per $A[i,j]$. With k looping $\frac{n}{2}$ times, the total reduction is $\sum_{l=0}^{(n-1)/2} 2n_b^2(n-l)n/2 \approx n_b^3 n^3/12 = N^3/12$.

Therefore, $lookahead = 1$ and operator fusion reduce approximately $N^3/12 + N^2 - Nn_b + N^2/2 + Nn_b$ memory operations. The total reduction is approximately $\frac{1}{12}N^3 + \frac{3}{2}N^2$ memory operations.

3.3 Input-Aware Adaptive Tile Size

To achieve a universal approach for tile size prediction on both CPU and GPU, we simplified the computer model, as shown in Figure 4. Both CPUs and GPUs consist of memory, cache, and multiple computing cores. Specifically, the cache for CPUs refers to the largest L3 cache, while for GPUs, it refers to the L2 cache. Since memory bandwidth is typically much lower than cache bandwidth, we assume that the data exchange speed between each core (CPU or GPU SM) and the cache is infinitely fast. Therefore, only the data exchange speed between the cache and memory, i.e., the memory bandwidth, is considered. Although actual computer are significantly more complex, this simplification was made to enable a unified tile size prediction across CPUs and GPUs.

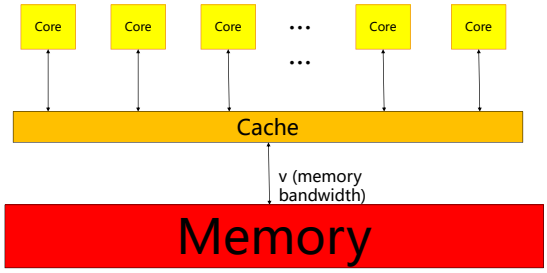

Figure 4: Simple computer model.

Next, we analyze the number of tasks and data transfer processes during the execution of IA-Chol. Figure 5 illustrates the directed acyclic graph (DAG) for a 6×6 matrix in the IA-Chol algorithm.

Analysis reveals that for an $N \times N$ matrix during the k-th iteration of IA-Chol, after PTSP(k, k) is completed, its data will be used to update TRSM-SYRK(k+2, k), TRSM-SYRK(k+3, k), ..., TRSM-SYRK(n, k), totaling $N-k-2$ TRSM-SYRK operations, as well as PTSP(k+2, k+2).

Subsequently, each TRSM-SYRK interacts with the pannelGEMM-TRSM-SYRK and trailingGEMM of its corresponding row and column for updates. The pannelGEMM-TRSM-SYRK, in turn, updates the remaining tiles requiring trailingGEMM operations.The total number of PTSP, TRSM-SYRK,pannelGEMM-TRSM-SYRK, and trailingGEMM in this process is summarized in Table 1.

Based on the aforementioned model, the impact of tile size on IA-Chol performance is primarily influenced by three factors: data exchange between memory and cache, the computation speed of the cores, and data exchange and task scheduling among multiple cores. Naturally, the actual scenario is far more complex, but we focus here on the most

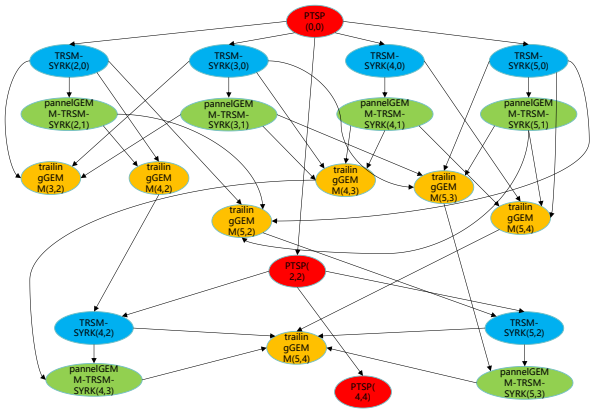

Figure 5: This is a directed acyclic graph (DAG) representation of the IA-Chol algorithm for a 6×6 matrix.

Table 1: Total Number of Tasks for PTSP, TRSM-SYRK, pannelGEMM-TRSM-SYRK, and trailingGEMM

Type of task	Number of tasks
PTSP	$\lfloor \frac{N}{2} \rfloor$
TRSM-SYRK	$\lfloor N/2 \rfloor (N-1-\lfloor N/2 \rfloor)$
pannelGEMM-TRSM-SYRK	$\lfloor N/2 \rfloor (N-1-\lfloor N/2 \rfloor)$
trailingGEMM	$\frac{\lfloor \frac{N-2}{2} \rfloor (\lfloor \frac{N-2}{2} \rfloor +1)(4\lfloor \frac{N-2}{2} \rfloor -1)}{6}$

critical factors. Next, we will discuss these three aspects in detail.

First, we discuss the data exchange between memory and cache. Let the size of the input matrix be N, the tile block size be $n_b \times n_b$, and the cache bandwidth be v. The input matrix is partitioned into $n \times n$ tiles. The time required to transfer a tile block from memory is then given by:

$$n = \lceil N/n_b \rceil, \tag{1}$$

$$sizeofn_b = \frac{n_b \times n_b \times 8}{1024^3}, \tag{2}$$

$$timeofn_b = \frac{sizeofn_b}{v}. \tag{3}$$

Since data is not always read from memory during the computation process, and sometimes data can be directly accessed from the cache, we define Num_{sum} as the total number of data operations required throughout the IA-Chol process, Num_{hit} as the number of times data is directly accessed from the cache, and Num_{memory} as $Num_{sum} - Num_{hit}$, which represents the number of times data must be read from memory.

From Algorithm 2, it can be observed that PTSP requires access to three data elements: $A[k,k]$, $A[k+1,k]$, and $A[k+1,k+1]$ in each iteration. TRSM-SYRK requires access to three data elements: $A[k,k]$, $A[i,k]$, and $A[i,i]$. PanelGEMM-TRSM-SYRK requires access to five data elements: $A[j,k]$, $A[k+1,k]$, $A[k+1,k+1]$, $A[j,k+1]$, and $A[j,j]$. trailingGEMM requires access to five data elements: $A[i,k]$, $A[j,k]$, $A[i,k+1]$, $A[j,k+1]$, and $A[i,j]$. Based on the number of occurrences of each task in Table 1, the value of Num_sum can be determined as follows:

$$NumofPTSP = 3 \times \left\lfloor \frac{n}{2} \right\rfloor, \tag{4}$$

$$NumofTRSM-SYRK = 3 \times \lfloor n/2 \rfloor (n-1-\lfloor n/2 \rfloor), \tag{5}$$

$$NumofpannelGEMM-TRSM-SYRK = 5 \times \lfloor n/2 \rfloor (n-1-\lfloor n/2 \rfloor), \tag{6}$$

$$NumoftrailingGEMM = 5 \times \frac{\lfloor \frac{n-2}{2} \rfloor (\lfloor \frac{n-2}{2} \rfloor +1)(4\lfloor \frac{n-2}{2} \rfloor -1)}{6}, \tag{7}$$

$$Num_{sum} = (4) + (5) + (6) + (7). \tag{8}$$

Next, we discuss the number of cache hits. First, we assume that the cache can hold a maximum of m tile blocks, and the cache replacement policy is Least Recently Used (LRU). We assume that all tile blocks follow a Zipf distribution. The process involves repeated accesses to tile blocks; if the requested tile block is not in the cache, it must first be loaded from memory. If the tile block is already in the cache, it can be accessed directly. The objective of the problem is to calculate the expected number of direct accesses when requesting Num_sum tile blocks. The total number of tiles is NumofTile.

$$m = \left\lfloor \frac{CacheSize}{sizeofn_b} \right\rfloor, \tag{9}$$

$$NumofTile = n \times (n+1)/2. \tag{10}$$

Each tile has a distinct probability of being accessed, which follows Zipf's distribution. The probability $P(i)$ for the i-th tile block is given by:

$$P(i) = \frac{i^{-\alpha}}{\sum_{j=1}^{NumofTile} j^{-\alpha}}, \tag{11}$$

where α is a parameter controlling the skewness of the distribution. A higher α leads to a greater concentration of access probability on the lower-index tile blocks (i.e., "hotspots").

Given that the cache can hold up to m tile blocks, we are interested in determining the probability of direct access,

which occurs when the desired tile block is already present in the cache. In a Least Recently Used (LRU) replacement strategy, the tile blocks in the cache are replaced based on their access history, with the most frequently accessed tile blocks having a higher chance of being present in the cache. The probability of direct access for a tile block i is approximated as:

$$P_{\text{hit}}(i) = \frac{\sum_{j=1}^{m} P(j)}{\sum_{j=1}^{NumofTile} P(j)}. \tag{12}$$

For each of the k accesses, the expected number of direct accesses is:

$$\mathbb{E}[X] = \sum_{i=1}^{NumofTile} P(i) \cdot P_{\text{hit}}(i), \tag{13}$$

where X is a random variable indicating whether tile block i is directly accessed. The total expected number of direct accesses over k accesses is then:

$$\mathbb{E}[\text{direct accesses}] = k \cdot \mathbb{E}[X]. \tag{14}$$

There are some special cases:

(1) **When** $m \geq NumofTile$**:** All tile blocks are in the cache, and every access is a direct access. Therefore, the expected number of direct accesses is simply k.
(2) **When** $m = 1$**:** Only one tile block is in the cache at a time, and the probability of direct access is determined solely by the most frequently accessed tile block.
(3) **When** $\alpha = 0$ **(Uniform Distribution):** All tile blocks have the same access probability, and the expected number of direct accesses becomes proportional to the fraction of tile blocks in the cache, i.e., $\mathbb{E}[\text{direct accesses}] = k \cdot \frac{m}{NumofTile}$.

Through multiple rounds of experimental testing, we found that $\alpha = 0.9$ is more suitable, which also indicates that IA-Chol belongs to a type of hot-spot distribution.

So, we need to access a total of Num_sum data, and the number of hits that are directly read from the cache is given by :

$$Num_{hit} = Num_{sum} \times \mathbb{E}[X], \tag{15}$$

$$Num_{hit} = Num_{sum} \times \sum_{i=1}^{NumofTile} P(i) \cdot P_{\text{hit}}(i). \tag{16}$$

Therefore, the data transfer time between memory and cache is approximately :

$$Num_{memory} = Num_{sum} - Num_{hit}, \tag{17}$$

$$DataExchange_{time} = timeofn_b \times Num_{memory}. \tag{18}$$

In addition to the significant impact of data exchange between memory and cache on performance, the computation of data also plays a crucial role in determining the final performance. Since IA-Chol only performs operator fusion, the overall computational load remains the same as the original Cholesky decomposition, as shown below:

$$\text{TotalComputation} = \frac{1}{3}N^3 + \frac{1}{2}N^2 + \frac{1}{6}N. \tag{19}$$

Thus, the time required for computation can be expressed as:

$$Computation_{time} = \frac{TotalComputation}{TheoreticalPeakPerformance}. \tag{20}$$

Next, we discuss the impact of task scheduling and data exchange among multiple cores on overall performance. As observed in Figure 4, for the k-th iteration, PTSP requires task scheduling with $n - k - 1$ tile blocks, where the step size of k is 2. In total, there are $\lceil \frac{n}{2} \rceil$ iterations, and each iteration involves one PTSP. For TRSM-SYRK, task scheduling is performed with $n - k$ tile blocks, resulting in $n - k - 2$ TRSM-SYRK operations. For pannelGEMM-TRSM-SYRK, each task scheduling involves $n - k - 1$ tile blocks, leading to $n - k - 2$ pannelGEMM-TRSM-SYRK operations. Finally, trailingGEMM schedules tasks with one tile block per operation, with a total of $(n-k-2)(n-k)/2$ trailingGEMM operations. Therefore, the total number of associated tasks for each of them is as follows:

$$\text{ASS}_{\text{PTSP}} = \sum_{k=0,\, k \text{ step } 2}^{\lceil \frac{n}{2} \rceil} (n - k - 1), \tag{21}$$

$$\text{ASS}_{\text{TRSM-SYRK}} = \sum_{k=0,\, k \text{ step } 2}^{\lceil \frac{n}{2} \rceil} (n - k)(n - k - 2), \tag{22}$$

$$\text{ASS}_{\text{pannelGEMM-TRSM-SYRK}} = \sum_{k=0,\, k \text{ step } 2}^{\lceil \frac{n}{2} \rceil} (n - k - 1)(n - k - 2), \tag{23}$$

$$\text{ASS}_{\text{trailingGEMM}} = \sum_{k=0,\, k \text{ step } 2}^{\lceil \frac{n}{2} \rceil} \frac{(n - k - 2)(n - k)}{2}. \tag{24}$$

The total number of associated tasks is:

$$ASS_{sum} = (21) + (22) + (23) + (24). \tag{25}$$

In a multi-core system, the task scheduling and data synchronization time is related both to the number of task schedules and the number of cores(Here, we define the influencing

factor on the GPU as the number of SMs (Streaming Multiprocessors).). Therefore, we can express this relationship using the following formula:

$$\text{TaskScheduling}_{\text{time}} = \beta \cdot num_{core} \cdot \log\left(\text{ASS}_{\text{sum}}\right). \quad (26)$$

It is important to note that this formula does not represent the actual task scheduling time but is used to indicate a relative time for comparing which tile size results in the least time. The reason for using $\log(\text{ASS}_{\text{sum}})$ is that, during the experiments, we observed that the task scheduling time indeed increases with the increase of ASS_{sum}. However, the rate of increase starts off fast and then gradually slows down, which closely resembles the shape of the $\log(x)$ function curve. Through experiments, it was found that a value of β around 1×10^{-6} is reasonable. In practice, task scheduling in a computer typically takes between a few microseconds and milliseconds. The value of β confirms this observation, indicating that our approach of predicting relative time is reasonable.

Finally, in addition to $\text{TaskScheduling}_{time}$, $\text{Computation}_{time}$, $\text{DataExchange}_{time}$, we also need to introduce a correction term:

$$b = \gamma \cdot \left(n_b - \frac{\text{Peak}}{\text{CacheSize}} \cdot 1.9\right)^2. \quad (27)$$

This adjustment is made to fine-tune the calculations. A value of $\gamma = 0.05$ is suitable. The term $n_b - \frac{\text{Peak}}{\text{CacheSize}}$ is an empirical factor, and the selection of n_b is related to the ratio of Peak to CacheSize. The ratio of Peak to CacheSize can, to some extent, represent the computational cache ratio. Based on empirical data and multiple experiments, this ratio significantly influences the choice of n_b. Similarly, the value of 1.9 is also an empirical constant.

Therefore, we estimate that when the tile block size is chosen as n_b, the possible execution time for IA-Chol is given by:

$$f(n_b) = \max((18), (20)) + (26) + (27), \quad (28)$$

where n_b is either a power of 2, i.e., 2^a, or $2^a + 2^{a-1}$, and $n_b \leq \text{CacheSize}$.

The choice of powers of 2 (2^a) or a combination of 2^a and 2^{a-1} for tile sizes is primarily to better align with the cache and memory access patterns of modern processors.

- 2^a: This is the most common choice because computer systems, particularly caches and memory hardware, are often optimized for memory blocks of size 2^n. Using 2^a ensures memory and cache accesses are aligned, reducing access latency and cache misses, while also improving parallel computation efficiency.
- $2^a + 2^{a-1}$: This combination is used in some hardware or algorithms to leverage the alignment benefits of 2^a while adding an extra memory block size (2^{a-1}) to enhance cache locality or utilization. This approach can help balance performance, especially when memory access patterns are not entirely regular.

Overall, these size choices aim to take full advantage of hardware optimizations, reduce cache misses, enhance computational efficiency, and make memory accesses more efficient.

Therefore, as discussed above, we only need to select the value of n_b that minimizes $f(n_b)$, which will be the most suitable choice for n_b. Since the possible values of n_b are either powers of 2, i.e., 2^a, or 2^a+2^{a-1}, this can be done within $\log_2(\text{CacheSize})$ time, making it a very efficient process.

4 Result

4.1 Experimental Setup

Table 2 and Table 3 are the performance parameters of different devices used in our experiment.To reduce the impact of noise, system load variations, and thermal management on performance, we iterate five or more times and take the average. The software used in our experiments includes: PLASMA-23.8.2, OpenBLAS-0.3.26, SLATE-2024.05.31, libFLAME-5.2.0, oneMKL 2020.0.4, MAGMA-2.8.0, cuSOLVER 11.5.0.

Table 2: Evaluation environment of CPU

CPU	*Intel Xeon Silver 4210R*	*Intel Xeon Gold 6240R*	*Phytium 2000+*
Arch.	x86_64	x86_64	aarch64
Cores	20	48	64
Sockets	2	2	1
Clock	2.4 GHz	2.4 GHz	2.2 GHz
L1 cache	32 K/core	32 K/core	32 K/core
L2 cache	1024 K/core	1024 K/core	2048 K/core
L3 cache	14080 K	36608 K	None
Compiler	GCC 9.3.0	GCC 9.3.0	GCC 9.3.0
Memory bandwidth	140.78 GB/s	115.2 GB/s	18.5 GB/s

4.2 Performance Improvement

Figure 6 shows the performance comparison results on the Intel Xeon Silver 4210R. The selected tile sizes are as follows: for matrices smaller than 1000, SLATE uses 128, while PLASMA and libFLAME use 64; for matrix sizes between 1000 and 5000, SLATE uses 256, while PLASMA and libFLAME use 128; for matrices larger than 5000, SLATE and PLASMA use 256 and 192, respectively, while libFLAME uses 128. Figure 7 illustrates the performance comparison results on the Intel Xeon Gold 6240R. The selected tile sizes are as follows: for matrix sizes smaller than 3000, SLATE uses 128, while

Table 3: Evaluation environment of GPU

GPU	*NVIDIA A100 80GB PCIe*	*NVIDIA H100 PCIe*
Number of SM	108	132
FP64 peak computational performance	19.5TFLOPS	51.2TFLOPS
L1 cache	192 K/core	256 K/core
L2 cache	40960 K	51200 K
Memory bandwidth	1935 GB/s	2000 GB/s
Compiler	NVCC 12.2	NVCC 12.2

PLASMA and libFLAME use 96; for matrix sizes larger than 3000, SLATE uses 256, PLASMA uses 192, and libFLAME uses 128. Figure 8 shows the performance comparison results on the Phytium 2000+ platform. The selected tile sizes are as follows: for matrix sizes smaller than 2000, SLATE uses 128, while PLASMA and libFLAME use 64; for matrix sizes between 2000 and 5000, SLATE uses 256, and both PLASMA and libFLAME use 128; for matrix sizes larger than 5000, SLATE uses 256, and both PLASMA and libFLAME use 192. Figure 9 presents the performance comparison results on the NVIDIA A100. The selected tile sizes are as follows: for matrices smaller than 10,000, SLATE uses a tile size of 896; for matrices larger than 10,000, SLATE uses a tile size of 1024. MAGMA, on the other hand, employs its internal adaptive tile size function. Figure 10 illustrates the performance comparison results on the NVIDIA H100. The selected tile sizes are as follows: for matrices smaller than 10,000, SLATE uses a tile size of 960; for matrices larger than 10,000, SLATE uses a tile size of 1024. MAGMA, on the other hand, utilizes its internal adaptive tile size function. These tile sizes correspond to the average performance-optimal values for each library in their respective ranges.

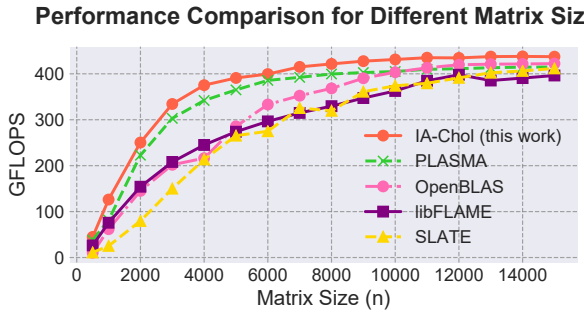

Figure 6: Performance comparison results on the Intel Xeon Silver 4210R.

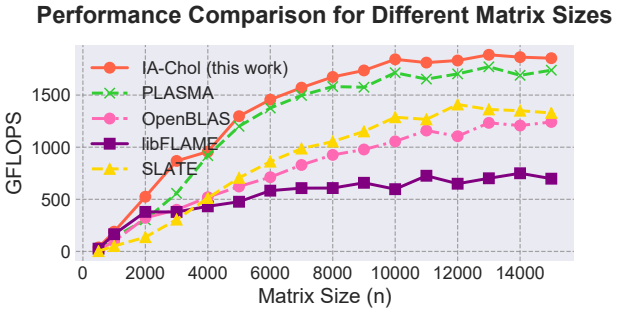

Figure 7: Performance comparison results on the Intel Xeon Gold 6240R.

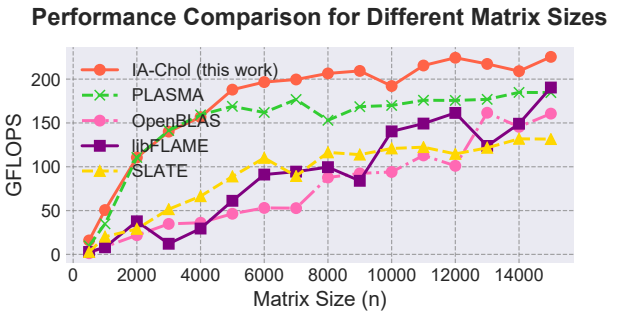

Figure 8: Performance comparison results on the Phytium 2000+ platform.

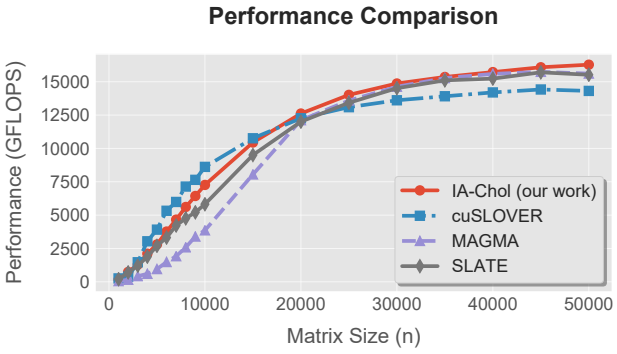

Figure 9: Performance comparison results on the NVIDIA A100.

Tables 4 to 6 present the average performance comparisons of IA-Chol with other state-of-the-art libraries across different matrix sizes on three distinct CPU platforms. The

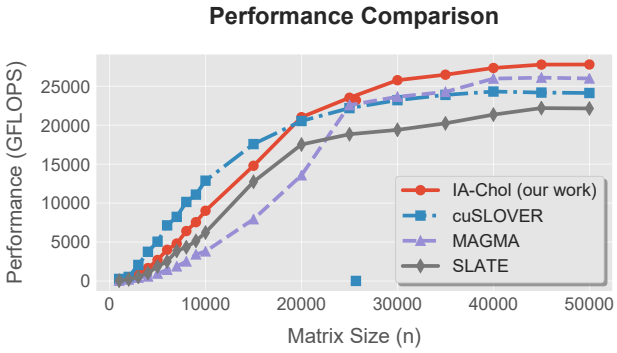

Figure 10: Performance comparison results on the NVIDIA H100.

results demonstrate that IA-Chol achieves significant performance improvements for small to medium-sized matrices on CPU platforms. For matrices larger than 5000, while IA-Chol still outperforms the competing libraries, the performance gains are relatively modest. This is because, as the matrix size increases, the proportion of trailing GEMM (i.e., matrix multiplication) in the overall computation becomes more dominant, gradually leading to compute-bound limitations. Notably, our optimization efforts focus on reducing cache accesses through the trailing GEMM strategy, without specifically optimizing matrix multiplication on CPUs.

Tables 7 and 8 present the average performance comparisons of IA-Chol with other state-of-the-art libraries across different matrix sizes on two GPU platforms. The results indicate that for matrices smaller than 10,000, IA-Chol performs slightly worse than cuSOLVER. This is primarily because IA-Chol is implemented based on SLATE, which is specifically designed for large-scale matrix computations. Consequently, its task scheduling introduces additional overhead for smaller matrices, leading to higher execution times compared to cuSOLVER. While this represents a limitation of IA-Chol on GPUs, it is acceptable in practical scenarios, as GPUs are typically utilized for large-scale matrix computations. For matrices larger than 10,000, IA-Chol demonstrates significant performance improvements. For instance, on the A100 platform, cuSOLVER achieves an efficiency of only 75.8%, whereas IA-Chol reaches 85.1%.

4.3 Comparison and Analysis of Tile Prediction Performance

Figure 11 illustrates the performance comparison between **IA-Chol** and **ILAENV** for adaptive tile size optimization on the Intel(R) Xeon(R) Silver 4210R platform. **ILAENV**, an auxiliary function in LAPACK, dynamically selects optimization parameters (e.g., block size NB) based on the target hardware and algorithm requirements. By providing performance-related default values, **ILAENV** enhances the execution efficiency of block algorithms while reducing the complexity of

Implementation	Intel(R) Xeon(R) Silver 4210R		
matrix size	500-1000	1000-5000	5000-15000
IA-Chol (this work)	**83.46**	**300.11**	**424.39**
OpenBLAS	37.79	181.80	388.41
LibFLAME	52.04	194.55	360.60
SLATE	17.13	143.12	355.54
PLASMA	49.90	235.19	373.61
Improvement over best baseline	**60.38%**	**27.61%**	**9.26%**

Table 4: This is a comparison of the average performance of IA-Chol and other libraries at different matrix sizes on Intel(R) Xeon(R) Silver 4210R, measured in GFLOPS.

Implementation	Intel(R) Xeon(R) Gold 6240R		
matrix size	500-1000	1000-5000	5000-15000
IA-Chol (this work)	**104.32**	**774.75**	**1752.86**
OpenBLAS	53.18	403.47	1006.58
LibFLAME	62.91	370.23	641.54
SLATE	28.12	334.38	1160.36
PLASMA	66.85	609.39	1590.91
oneMKL	62.04	319.05	726.65
Improvement over best baseline	**56.06%**	**27.13%**	**10.18%**

Table 5: This is a comparison of the average performance of IA-Chol and other libraries at different matrix sizes on Intel(R) Xeon(R) Gold 6240R, measured in GFLOPS.

Implementation	Phytium 2000+		
matrix size	500-1000	1000-5000	5000-15000
IA-Chol (this work)	**33.02**	**123.61**	**202.18**
OpenBLAS	4.50	30.03	100.74
LibFLAME	5.56	29.17	122.24
SLATE	12.15	48.28	114.68
PLASMA	21.49	120.26	172.47
Improvement over best baseline	**53.63%**	**2.78%**	**17.22%**

Table 6: This is a comparison of the average performance of IA-Chol and other libraries at different matrix sizes on Phytium 2000+, measured in GFLPOPS.

Implementation	NVIDIA A100 80GB PCIe	
matrix size	1000-10000	10000-50000
IA-Chol (this work)	3484.78	**13628.68**
cuSLOVER	**4377.01**	12798.31
MAGMA	1541.81	12717.77
SLATE	3001.71	12981.61
Improvement over best baseline	None	**4.98%**

Table 7: This is a comparison of the average performance of IA-Chol and other libraries at different matrix sizes on NVIDIA A100, measured in GFLPOPS.

Implementation	NVIDIA H100 PCIe	
matrix size	1000-10000	10000-50000
IA-Chol (this work)	3744.72	**22614.62**
cuSLOVER	**6114.75**	21435.51
MAGMA	1522.45	19318.89
SLATE	2573.86	17580.84
Improvement over best baseline	None	**5.50%**

Table 8: This is a comparison of the average performance of IA-Chol and other libraries at different matrix sizes on NVIDIA H100, measured in GFLPOPS.

manual parameter tuning. However, as shown in the figure, **IA-Chol** significantly outperforms **ILAENV**.

This improvement is primarily due to the change in the optimal tile size after operator fusion (as depicted in Figure 2 of Chapter 1), a factor not accounted for by **ILAENV**. Additionally, **ILAENV** determines tile size primarily based on hardware performance metrics, with minimal sensitivity to the input matrix size N. For instance, even when the input matrix size N exceeds 10,000, **ILAENV** still recommends a tile size of 64, which is suboptimal under the operator fusion scenario, leading to its inferior performance compared to **IA-Chol**.

Figure 12 presents a comparison of adaptive tile size approaches on the NVIDIA A100. **Magma_get_dpotrf_nb** is a function in MAGMA that dynamically adjusts the tile size based on the input matrix N. **Statistical Modeling**, proposed by Ray-Bing Chen, Yaohung M. Tsai, and Weichung Wang in 2014[7], uses statistical surrogate models to predict performance for different block sizes and employs online monitors to detect and avoid unexpected performance fluctuations.

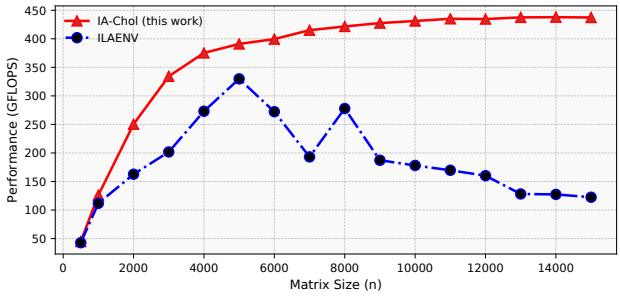

Figure 11: Comparison of Adaptive Tile Size Strategies on Intel(R) Xeon(R) Silver 4210R.

IA-Chol outperforms **Statistical Modeling** slightly and significantly surpasses **magma_get_dpotrf_nb**. The limitations of **Statistical Modeling** stem from its model being primarily designed for QR decomposition, with limited efficacy when extended to Cholesky decomposition. Moreover, it requires numerous input parameters (e.g., matrix size, block size), necessitating extensive experimentation and parameter tuning, which increases complexity and hinders practical usability.

As for **magma_get_dpotrf_nb**, its performance is subpar because it is specifically designed for MAGMA, where the optimal tile size is generally smaller, making it less effective in broader scenarios.

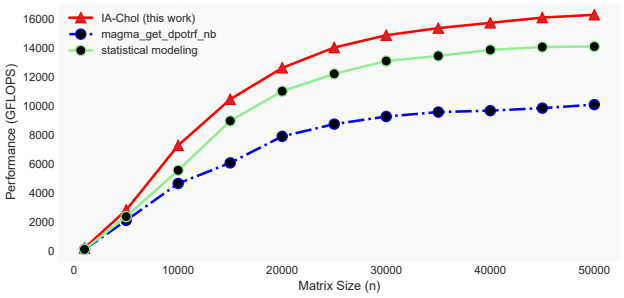

Figure 12: Comparison of Adaptive Tile Size Strategies on NVIDIA A100 80GB PCIe.

4.4 Ablation Study Analysis

Figure 13 presents the ablation study on Intel(R) Xeon(R) Silver 4210R and NVIDIA A100 80GB PCIe platforms. The figure compares the average performance under two conditions: operator fusion only and adaptive tile size only, for different matrix sizes. In the baseline experiments, PLASMA is used on the Intel(R) Xeon(R) Silver 4210R, while SLATE is used on the NVIDIA A100 80GB PCIe. The respective tile sizes are detailed in Section 2 of Chapter 4.

It can be seen that using either of the two methods independently results in some performance improvement. However, on the CPU, for matrices of size 5000 to 15000, the average performance of the adaptive tiling is slightly lower than that of the baseline. This is primarily because our adaptive tiling method is specifically designed for Cholesky decomposition after operator fusion, leading to a smaller predicted optimal tile size. As the matrix size increases, the required tile size also increases, which explains this phenomenon. So why doesn't this issue arise on the GPU? Moreover, for matrices between 20000 and 50000, the performance of operator fusion on the GPU is slightly lower than the baseline. This is because the computational-to-memory access ratio on the GPU is much higher than that of the CPU. In the case of matrices from 20000 to 50000, the smaller tile prediction actually improves the GPU's cache hit rate. If the tile size were larger, the cache might not be able to accommodate it, or frequent tile exchanges could decrease the hit rate, thus negatively affecting performance.

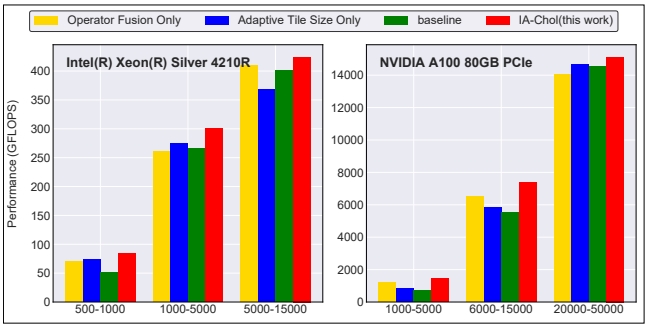

Figure 13: Ablation Study on Intel(R) Xeon(R) Silver 4210R and NVIDIA A100 80GB PCIe.

4.5 In-depth Performance Analysis

Figure 14 shows a comparison of cache hit rates. It can be observed that as the matrix size N increases, the cache miss rate of IA-Chol decreases, while the cache miss rates of other libraries increase. This can be attributed to two main factors: firstly, the operator fusion scheme itself reduces the cache miss rate, and secondly, IA-Chol adaptively adjusts the tile size as N increases. These factors contribute to the superior performance of IA-Chol compared to other libraries.

5 Related Work

Cholesky factorization finds use in areas beyond conventional matrix decomposition. Ribizel and Anzt [21] presented a parallel symbolic Cholesky approach to optimize graph-based algorithms for sparse systems. Kurzak et al. [16] showed its adaptability across computing platforms, extending the work on batched factorization. Lu et al. [18] explored

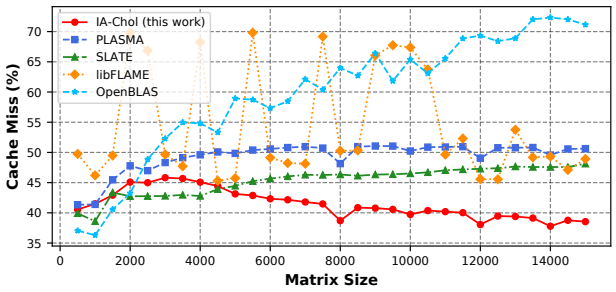

Figure 14: Comparison of Cache Miss Rates between IA-Chol and Other Linear Algebra Libraries (Lower is Better).

Cholesky's application on AI accelerators, demonstrating its potential for data-intensive tasks in artificial intelligence and machine learning workloads.

IA-Chol has demonstrated impressive performance on CPU architectures, achieving significant efficiency gains in Cholesky decomposition. Although IA-Chol is task level parallelism inherited from PLASMA, our method has good versatility, and its wide application will promote the development of other platforms, such as LAPACK.

However, with the rapid advancements in GPU technology, it is becoming increasingly evident that GPUs may take the lead in parallel computing in the near future[3, 8, 9, 11, 14, 19, 20, 25, 26, 28–30]. Our IA-Chol has also achieved significant performance improvements on GPUs, with efficiency increasing from 75.8% to 85.1% compared to cuSOLVER. Compared to other state-of-the-art GPU operator libraries such as MAGMA and SLATE, we observed a 5% to 12% performance improvement. However, for GPUs like the H100, which feature higher bandwidth and greater computational power, larger matrices are required to fully unleash their potential (this is also true for cuSOLVER, MAGMA, and SLATE). Optimizing algorithms to better leverage such high-performance hardware is an interesting direction for future research.

Furthermore, in the area of adaptive blocking, there is an alternative approach that avoids complex modeling by directly bypassing the need to select tile sizes, instead optimizing the Cholesky decomposition using a recursive method. This approach is discussed in detail in the work of Bjarne Stig Andersen, Jerzy Waśniewski, Fred G. Gustavson[2], Nawaaz Ahmed, Keshav Pingali[1], and I. Jonsson[13]. We have also implemented it and observed some improvement; however, it still does not perform as well as the blocking method. This is the reason we opted for blocking optimization in our Cholesky decomposition. Nevertheless, this method remains highly promising, and if the computation-to-memory access

ratio improves in future hardware, I believe it will prove to be valuable.

6 Conclusion

Our initial goal was to address the performance deficiencies of Cholesky decomposition for small- and medium-sized matrices. To this end, we proposed an operator fusion scheme tailored to Cholesky decomposition, achieving outstanding results that outperformed existing operator libraries. However, this introduced a new challenge: the optimal tile size shifted, rendering traditional prediction methods ineffective. To overcome this, we developed a novel tile prediction model, which delivered highly satisfactory results. Experimental results demonstrate that IA-Chol not only significantly outperforms various linear algebra libraries in computational efficiency across a range of matrix sizes on both CPUs and GPUs, but also reduces memory access overhead while substantially improving cache utilization. In multicore environments, IA-Chol effectively leverages hardware resources, showcasing exceptional parallel scalability. The design principles and optimization strategies of IA-Chol provide valuable insights for further enhancing matrix decomposition algorithms.

Acknowledgments

We sincerely appreciate the invaluable comments provided by all the reviewers, which have greatly helped us improve our work. This research was supported by the National Key Research and Development Program of China: 2023YFB3002300, 2023YFA1011704, 2021YFBO300101. We extend our heartfelt thanks for their support.

References

[1] Nawaaz Ahmed and Keshav Pingali. 2000. Automatic Generation of Block-Recursive Codes. In *Proceedings of the 6th International Euro-Par Conference on Parallel Processing (Euro-Par '00)*. Springer-Verlag, Berlin, Heidelberg, 368–378.

[2] Bjarne Stig Andersen, Jerzy Waśniewski, and Fred G. Gustavson. 2001. A recursive formulation of Cholesky factorization of a matrix in packed storage. *ACM Trans. Math. Software* 27, 2 (2001), 214–244. doi:10.1145/383738.383741

[3] Peter Benner, Pablo Ezzatti, Daniel Kressner, Enrique S. Quintana-Ortí, and Alfredo Remón. 2011. A mixed-precision algorithm for the solution of Lyapunov equations on hybrid CPU-GPU platforms. *Parallel Comput.* 37, 8 (2011), 439–450.

[4] Qinglei Cao, Yu Pei, Kadir Akbudak, Aleksandr Mikhalev, George Bosilca, Hatem Ltaief, David E. Keyes, and Jack J. Dongarra. 2020. Extreme-Scale Task-Based Cholesky Factorization Toward Climate and Weather Prediction Applications. In *Proceedings of the Platform for Advanced Scientific Computing (PASC 2020)*. 2:1–2:11.

[5] Erin C. Carson. 2018. The Adaptive s-Step Conjugate Gradient Method. *SIAM J. Matrix Anal. Appl.* 39, 3 (2018), 1318–1338. doi:10.1137/16M1077892

[6] Erin C. Carson, Tomás Gergelits, and Ichitaro Yamazaki. 2022. Mixed precision s-step Lanczos and conjugate gradient algorithms. *Numerical Linear Algebra with Applications* 29, 3 (2022). doi:10.1002/nla.2487

[7] Ray-Bing Chen, Yaohung M. Tsai, and Weichung Wang. 2014. Adaptive block size for dense QR factorization in hybrid CPU-GPU systems via statistical modeling. *Parallel Comput.* 40, 5-6 (2014), 70–85.

[8] Terry Cojean, Abdou Guermouche, Andra Hugo, Raymond Namyst, and Pierre-André Wacrenier. 2019. Resource aggregation for task-based Cholesky Factorization on top of modern architectures. *Parallel Comput.* 83 (2019), 73–92.

[9] Terry Cojean, Abdou Guermouche, Andra Hugo, Raymond Namyst, and Pierre-André Wacrenier. 2019. Resource aggregation for task-based Cholesky Factorization on top of modern architectures. *Parallel Comput.* 83 (2019), 73–92.

[10] T. A. Davis, S. Rajamanickam, and W. M. Sid-Lakhdar. 2016. A survey of direct methods for sparse linear systems. *Acta Numer.* 25 (2016), 383–566.

[11] Jack J. Dongarra, Mathieu Faverge, Hatem Ltaief, and Piotr Luszczek. 2011. High performance matrix inversion based on LU factorization for multicore architectures. In *Proceedings of the MTAGS Workshop at SC*. 33–42.

[12] G. H. Golub and C. F. Van Loan. 2013. *Matrix Computations*. Johns Hopkins University Press, Baltimore, MD.

[13] F. G. Gustavson and I. Jonsson. 2000. Minimal-storage high-performance Cholesky factorization via blocking and recursion. *IBM Journal of Research and Development* 44, 6 (2000), 823–850. doi:10.1147/rd.446.0823

[14] Azzam Haidar, Ahmad Abdelfattah, Mawussi Zounon, Stanimire Tomov, and Jack J. Dongarra. 2018. A Guide for Achieving High Performance with Very Small Matrices on GPU: A Case Study of Batched LU and Cholesky Factorizations. *IEEE Transactions on Parallel and Distributed Systems* 29, 5 (2018), 973–984.

[15] Richard Hartley and Andrew Zisserman. 2004. *Multiple View Geometry in Computer Vision*. Cambridge University Press.

[16] Jakub Kurzak, Hartwig Anzt, Mark Gates, and Jack J. Dongarra. 2016. Implementation and Tuning of Batched Cholesky Factorization and Solve for NVIDIA GPUs. *IEEE Transactions on Parallel and Distributed Systems* 27, 7 (2016), 2036–2048.

[17] Yuechen Lu, Yuchen Luo, Haocheng Lian, Zhou Jin, and Weifeng Liu. 2021. Implementing LU and Cholesky factorizations on artificial intelligence accelerators. *CCF Transactions on High Performance Computing* 3, 3 (2021), 286–297.

[18] Yuechen Lu, Yuchen Luo, Haocheng Lian, Zhou Jin, and Weifeng Liu. 2021. Implementing LU and Cholesky factorizations on artificial intelligence accelerators. *CCF Transactions on High Performance Computing* 3, 3 (2021), 286–297.

[19] Sparsh Mittal and Jeffrey S. Vetter. 2015. A Survey of CPU-GPU Heterogeneous Computing Techniques. *Comput. Surveys* 47, 4 (2015), 69:1–69:35.

[20] Ali Mohammadjafari and Poorya Khajouie. 2024. Optimizing Task Scheduling in Heterogeneous Computing Environments: A Comparative Analysis of CPU, GPU, and ASIC Platforms Using E2C Simulator. *CoRR* abs/2405.08187 (2024).

[21] Tobias Ribizel and Hartwig Anzt. 2023. Parallel Symbolic Cholesky Factorization. In *SC Workshops 2023*. 1721–1727.

[22] Matthias W. Seeger. 2004. Gaussian Processes For Machine Learning. *International Journal of Neural Systems* 14, 2 (2004), 69–106. https://api.semanticscholar.org/CorpusID:63955376

[23] Sailes K. Sengijpta. 1993. Fundamentals of Statistical Signal Processing: Estimation Theory. *Technometrics* 37, 4 (1993), 465–466.

[24] Lino M. Silva and Aurelio R. L. Oliveira. 2021. Modified controlled Cholesky factorization for preconditioning linear systems from the interior-point method. *Comput. Appl. Math.* 40, 4 (2021).

[25] Yuki Tsujita and Toshio Endo. 2015. Data Driven Scheduling Approach for the Multi-node Multi-GPU Cholesky Decomposition. *Journal of*

Supercomputing: Special Issue on High Performance Computing for Scientific Applications (2015), 69–82.

[26] Sundaresan Venkatasubramanian and Richard W. Vuduc. 2009. Tuned and wildly asynchronous stencil kernels for hybrid CPU/GPU systems. In *Proceedings of the ACM International Conference on Supercomputing (ICS)*. 244–255.

[27] Xiaoqun Wang and Ian H. Sloan. 2011. Quasi-Monte Carlo Methods in Financial Engineering: An Equivalence Principle and Dimension Reduction. *Operations Research* 59, 1 (2011), 80–95.

[28] Canqun Yang, Feng Wang, Yunfei Du, Juan Chen, Jie Liu, Huizhan Yi, and Kai Lu. 2010. Adaptive Optimization for Petascale Heterogeneous CPU/GPU Computing. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER)*. 19–28.

[29] Depeng Yang, Gregory D. Peterson, and Husheng Li. 2012. Compressed sensing and Cholesky decomposition on FPGAs and GPUs. *Parallel Comput.* 38, 8 (2012), 421–437.

[30] Yuanhang Yu, Dong Wen, Ying Zhang, Xiaoyang Wang, Wenjie Zhang, and Xuemin Lin. 2021. Efficient Matrix Factorization on Heterogeneous CPU-GPU Systems. In *Proceedings of the 37th IEEE International Conference on Data Engineering (ICDE)*. 1871–1876.

[31] O. C. Zienkiewicz and R. L. Taylor. 2005. *The Finite Element Method for Solid and Structural Mechanics* (6th ed.).