

Non-Delayed Cholesky Factorization

Yuchen Luo

SSSLab, Dept. of CST
China University of Petroleum-Beijing
Beijing, China
yuchen.luo@student.cup.edu.cn

Shaoshuai Zhang

University of Electronic Science and Technology of China
Chengdu, China
szhang94@uestc.edu.cn

Hongyu Chen

SSSLab, Dept. of CST
China University of Petroleum-Beijing
Beijing, China
hongyu.chen@student.cup.edu.cn

Weifeng Liu

SSSLab, Dept. of CST
China University of Petroleum-Beijing
Beijing, China
weifeng.liu@cup.edu.cn

Abstract

While General Matrix-Matrix Multiplication (GEMM) often nearly approaches the peak performance of modern GPUs, other fundamental dense linear algebra algorithms, such as Cholesky factorization, exhibit a significant performance gap. We identify the root cause of this inefficiency as the conventional ‘delayed update’ paradigm, a multi-kernel approach that relies on coarse-grained synchronization and incurs substantial data movement overhead, particularly for small-to-medium matrices.

This paper challenges that paradigm by introducing a **Non-Delayed Cholesky Factorization**. Our core insight is that the tile-level dataflow of the Cholesky trailing update is structurally isomorphic to that of a high-performance GEMM. We exploit this by redesigning the entire factorization to execute like a GEMM within a single, persistent CUDA kernel. In our tile-centric model, each thread block assumes lifetime ownership of a single matrix tile, performing all updates and the final factorization in-place using on-chip memory. A lightweight, asynchronous dependency-tracking mechanism coordinates thousands of blocks without global barriers, largely increasing concurrency.

We evaluate our design on diverse hardware platforms, including three generations of NVIDIA GPUs (A100, RTX 5090, and H100) and an AMD GPU (R9700). Our non-delayed Cholesky factorization consistently outperforms state-of-the-art libraries, achieving speedups of up to **2.28×** over cuSOLVER and **2.16×** over rocSOLVER. More profoundly, we demonstrate that our algorithm’s structural efficiency is its key advantage: it consistently converts over 90% of its underlying GEMM kernel’s performance into final factorization throughput across all architectures. This stands in stark contrast to cuSOLVER, whose efficiency relative to cuBLAS fluctuates significantly (from >90% down to 60%) across different GPUs. By importing the execution philosophy of GEMM, rather than merely calling it as a subroutine, our work presents a new and highly effective model for designing dense Cholesky factorizations on modern accelerators.

CCS Concepts

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

Keywords

Cholesky factorization, Linear algebra, GEMM, GPU

ACM Reference Format:

Yuchen Luo, Hongyu Chen, Shaoshuai Zhang, and Weifeng Liu. 2026. Non-Delayed Cholesky Factorization. In *2026 International Conference on Supercomputing (ICS '26)*, July 06–09, 2026, Belfast, United Kingdom. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3797905.3800554>

1 Introduction

General Matrix–Matrix Multiplication (GEMM) [15] remains the major dense linear algebra primitive on modern GPUs that consistently runs near architectural peak [41]. Decades of optimization, combined with highly engineered vendor libraries such as cuBLAS [36] and frameworks such as CUTLASS [38], have produced GEMM kernels that saturate compute pipelines, maximize data reuse, and fully exploit on chip execution resources. This makes GEMM a near-perfect computational primitive and the gold standard against which other dense linear algebra algorithms are measured.

By comparison, Cholesky factorization, another cornerstone of scientific and engineering applications [7], exhibits a significant performance gap within a single GPU. On the same hardware, the state-of-the-art implementations in widely-used libraries such as cuSOLVER [37] and MAGMA [5] lag considerably behind GEMM, leaving a substantial portion of the GPU’s floating-point capability untapped. This performance gap is particularly pronounced for the small- and medium-sized matrices common in many real-world applications, where overheads are harder to amortize.

A central limitation is the delayed update paradigm inherent in classical blocked factorizations. By decomposing the computation into panel factorizations and trailing updates, these methods rely on coarse-grained kernel boundaries and frequent global synchronization. Each stage forces tiles to be written back to global memory and reloaded later, preventing the factorization from retaining the fine-grained data locality and continuous compute-bound behavior that make GEMM efficient.

This motivates a fundamental question: must Cholesky factorization be organized around a coarse panel-update structure, or can it

be redesigned to behave like a GEMM at runtime? At the tile level, the trailing update has the same rank- b accumulation structure that underlies blocked GEMM. The similarity in dataflow suggests that the performance gap is not inherent to Cholesky itself, but to the execution model used by existing implementations.

Motivated by this observation, we propose a **Non-Delayed Cholesky Factorization** designed for single-GPU execution. The method abandons the traditional panel-update paradigm and adopts a tile-centric, GEMM-like execution model built upon three key ideas:

- (1) **Monolithic Kernel Execution:** Unlike DAG-based runtimes like MAGMA that schedule sequences of distinct CUDA kernels, our entire factorization is encapsulated within a *single, persistent CUDA kernel*. This eliminates all kernel launch overhead and the need for inter-kernel synchronization mechanisms like CUDA events.
- (2) **Intra-Kernel, Tile-Centric Scheduling:** The scheduling logic is moved inside the kernel. Each thread block is permanently assigned to a matrix tile and manages its own progress. Dependencies are resolved at the tile-granularity through a lightweight, asynchronous flag-polling mechanism, allowing computation to proceed immediately once data is ready, rather than waiting for panel-level completion.
- (3) **Full Data Residency:** This single-kernel, tile-centric design enables a crucial optimization: each tile is loaded from global memory once, remains resident in on-chip memory (registers and shared memory) for its entire computational lifecycle, and is written back only once. This strategy drastically reduces global memory traffic compared to multi-kernel approaches.

By restructuring the factorization to follow GEMM’s execution principles rather than invoking GEMM only as a subroutine, our non-delayed formulation shows substantial performance advantages over both closed-source vendor libraries (e.g., cuSOLVER and rocSOLVER) and open-source frameworks (e.g., MAGMA). Specifically, when compared against the highly optimized official vendor baselines, our approach yields peak speedups of $1.51\times$ on NVIDIA A100, $1.85\times$ on H100, $2.28\times$ on RTX 5090, and $2.16\times$ on AMD R9700 GPUs. Furthermore, the relative speedups over MAGMA are even higher across these platforms. By largely eliminating coarse-grained scheduling overheads, this non-delayed paradigm effectively closes much of the long-standing performance gap between dense Cholesky factorization and GEMM.

2 Background

This section deconstructs the performance limitations of conventional Cholesky factorization on GPUs. We first analyze the ‘delayed update’ paradigm, a cornerstone of existing library implementations, to reveal its inherent overheads. We then present high-performance GEMM as an ideal non-delayed execution model, establishing a conceptual framework for the algorithm developed in this paper.

2.1 Delayed Update Paradigm in Tiled Cholesky

Given a symmetric positive definite matrix $A \in \mathbb{R}^{n \times n}$, Cholesky factorization computes a lower triangular matrix L such that $A = LL^T$.

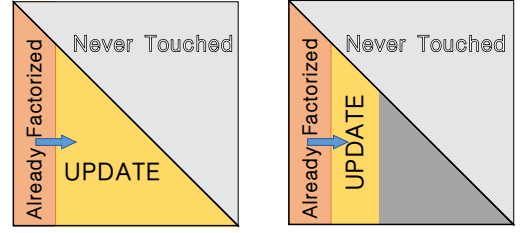


Figure 1: The delayed update paradigm in right-looking (left) and left-looking (right) algorithms. The orange and yellow regions abstractly represent groups of multiple matrix tiles whose computations are distributed across numerous GPU thread blocks. Both approaches share a fundamental bottleneck: the trailing yellow tiles are updated synchronously as a single lock-step group. This forced execution prevents individual tiles from progressing immediately when their dependencies are met, characterizing the ‘delayed update’ overheads shown in Definition 1.

While unblocked, column-wise algorithms are simple to express, their reliance on Level-1 and Level-2 BLAS operations leads to poor performance on modern GPUs, which are architected for high-throughput, data-parallel computation.

To improve data reuse and leverage the memory hierarchy, high-performance implementations universally adopt blocked algorithms. The matrix is partitioned into tiles, and the computation is organized as a sequence of operations on these tiles. Regardless of the specific variant (e.g., left-looking or right-looking), implementations in modern libraries are fundamentally built upon a common strategy: the **delayed update paradigm**.

This paradigm is a deliberate design choice motivated by the desire to offload computation to vendor-optimized, but monolithic, Level-3 BLAS kernels. As noted in early work on GPU-accelerated libraries [35], execution is typically managed by an external runtime that schedules a Directed Acyclic Graph (DAG) of tasks, where each task corresponds to launching a complete CUDA kernel (e.g., ‘potrf’, ‘trsm’ and ‘gemm’). The procedure involves: 1) executing kernels to factorize a panel of tiles, and 2) after waiting for their completion, launching a series of ‘gemm’ kernels to update the trailing matrix. While this allows for some overlap, the fundamental scheduling unit remains the kernel. This aggregation maximizes the work for each external GEMM call, but it comes at a significant cost. It imposes coarse-grained synchronization points and necessitates substantial data movement, as tiles are repeatedly read from and written back to global memory.

As illustrated in Figure 1, many tiles are forced to wait in lock-step even when their own data dependencies have already been satisfied. In earlier work, the term *delayed update* primarily referred to the accumulation of work into later Level-3 BLAS calls [35]. Here, however, we elevate and formalize the term from an implementation artifact to a broader algorithmic concept, focusing strictly on data-flow and block-level ready states. Under this definition, even fused single-kernel designs such as IA-Chol [13] remain delayed-update methods, because they still enforce block-level synchronization barriers that preserve the traditional panel-update ordering. Therefore,

we formally abstract and define this performance-limiting behavior from a tile-centric dependency perspective:

DEFINITION 1 (DELAYED UPDATE, TILE-CENTRIC). *An algorithm uses a **delayed update** scheme if there exists at least one tile whose update, although all its data dependencies are already satisfied, is not executed independently but is forced to start and finish together with operations on other tiles.*

This forced aggregation and synchronization is precisely what creates the performance bottleneck, especially for small to medium matrices where these overheads cannot be effectively amortized. Viewed from a data-flow perspective, both classical algorithmic variants introduce artificial synchronization boundaries. Traditional right-looking factorizations operate in a *producer-centric* manner: a newly factorized panel eagerly broadcasts its updates, forcing target tiles in the trailing matrix to be repeatedly loaded and stored across successive steps. Conversely, while traditional left-looking algorithms pull updates in a *consumer-centric* manner, they still synchronize at the coarse panel level.

2.2 GEMM as the Non-Delayed Ideal

In stark contrast to the delayed update model stands the execution pattern of a high-performance, blocked GEMM. Consider the computation $C \leftarrow C + AB$, where all matrices are partitioned into tiles. A well-designed GEMM implementation exhibits a purely **non-delayed** behavior at the tile level.

In this ideal model, a CUDA thread block is typically assigned to compute a single output tile C_{ij} . This tile is kept resident in the block's on-chip memory (registers and shared memory) for its entire computational lifetime. As the computation iterates through the tiled inner dimension k , the tile-level update $C_{ij} \leftarrow C_{ij} + A_{ik}B_{kj}$ is performed *immediately* after the corresponding operand tiles A_{ik} and B_{kj} are fetched. The progress of tile C_{ij} is completely independent of any other tile C_{lm} ; there is no algorithmic requirement for them to synchronize or wait for each other. This tile-wise independence is the key to GEMM's efficiency, as it maximizes parallelism and data reuse while minimizing stalls. This ideal behavior can be formally defined as:

DEFINITION 2 (NON-DELAYED UPDATE, TILE-CENTRIC). *An algorithm uses a **non-delayed update** scheme if, for every tile, each update whose data dependencies are satisfied is executed independently and applied to that tile as soon as it becomes ready.*

Blocking GEMM is the canonical example of a non-delayed algorithm. The central thesis of our work stems from a key observation: the trailing matrix update in Cholesky, which constitutes the bulk of the computation, is structurally isomorphic to a series of GEMM-like operations. This suggests that instead of treating Cholesky as a coarse-grained sequence of calls to external BLAS kernels, we can redesign it from the ground up to *behave* like a single, unified, non-delayed kernel. This perspective, which aims to import the execution philosophy of GEMM rather than merely calling it as a subroutine, forms the foundation of our proposed algorithm.

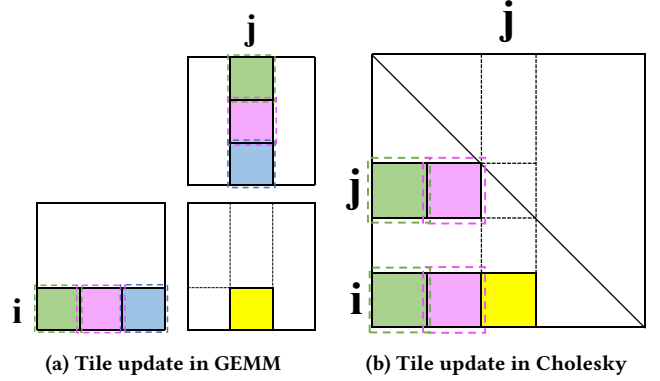


Figure 2: Tile update in (a) GEMM and (b) Cholesky. Both consume tile pairs from row/column panels to contribute local tile-level updates, yielding structurally identical dataflow.

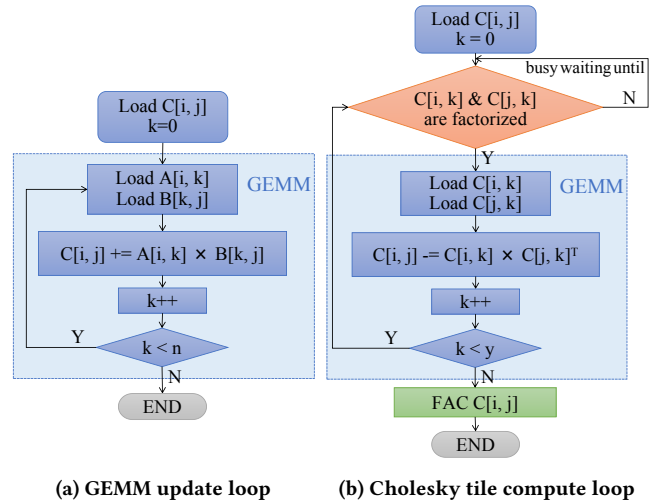


Figure 3: (a) GEMM repeatedly loads a tile pair and accumulates their product into the destination tile. (b) Cholesky executes the same identical loop structure, except tile pairs become available dynamically after factorization. Both share a GEMM-like compute pattern.

2.3 Structural Foundation: Why Cholesky Updates Are GEMM-Like

The starting point of our algorithmic design is the observation that the tile-level trailing update in Cholesky factorization shares the same computational structure as the update of a destination tile in GEMM, as shown in Figure 2. In a blocked GEMM, each output tile C_{ij} is formed by accumulating a sequence of local products. Each iteration selects one tile from the row panel and one from the column panel, multiplies them, and adds the result to C_{ij} . This process continues until all tile pairs along the shared dimension have been consumed. The update is therefore a regular accumulation that is both compute-bound and highly structured. Its flowchart is shown in Figure 3a.

The Cholesky trailing update follows the same pattern. For tile C_{ij} , each update step consumes tiles C_{ik} and C_{jk} , performs a small matrix multiplication, and subtracts the result from C_{ij} . Although the algebra differs from GEMM, the tile access pattern, the iteration structure, and the accumulation behavior are identical. The dashed boxes in Figure 3b illustrate this mapping; every contribution to C_{ij} originates from a corresponding pair of input tiles located in the i -panel and j -panel, perfectly mirroring the tile-wise dependency structure of GEMM.

This structural alignment provides the foundation of our algorithm. Once the trailing update is viewed as a GEMM-like accumulation, it becomes natural to redesign the factorization so that tiles are updated using the same principles that underlie high-performance GEMM kernels. The remainder of the algorithm follows directly from this perspective: tiles should remain on chip, updates should occur as soon as their data becomes available, and the entire computation should mirror the fine-grained, compute-bound execution model of GEMM.

3 Non-Delayed Cholesky Factorization

The goal of this section is to develop the algorithmic structure that enables Cholesky factorization to approach the near peak efficiency of GEMM on a single GPU. The central premise, established in Section 2, is that the trailing updates in Cholesky fundamentally share the same tile-level dataflow as blocked GEMM. Exploiting this structural similarity necessitates departing from the traditional approach of invoking external BLAS subroutines. Instead, it requires redesigning the factorization around a fused, GEMM-like execution model wherein all tile-level operators are mapped into a single persistent kernel.

The following subsections build this design systematically. First, we outline the *fused, tile-resident execution model*, an execution paradigm that keeps each target tile in on-chip memory throughout its lifetime and eliminates the rigid panel-update-panel synchronization barriers of traditional delayed-update algorithms. Next, we detail the register-level mapping, revealing how the update phase is mapped directly onto a high-performance SIMT GEMM microkernel to inherit its compute-bound characteristics. Finally, we discuss thread block configuration and memory alignment, noting how tile dimensions and shared memory footprints map to hardware resource limits.

3.1 A Fused, Tile-Resident Execution Model

The non-delayed tiled Cholesky factorization adopts a fully fused and tile-resident execution model in which every tile is permanently assigned to a persistent CUDA block. All computations associated with the tile—including its trailing updates, triangular solve, and diagonal factorization—are executed by the same block, using on-chip memory throughout. This eliminates kernel boundaries, avoids global synchronization, and allows fine-grained progress of tiles based solely on true data dependencies.

Dependency tracking through a global readiness array. To coordinate progress among tiles without global synchronization, the algorithm employs a global array `signal_fin[]` in device memory. For a matrix partitioned into $p \times p$ tiles, the lower triangular region contains $(p^2 + p)/2$ tiles, and `signal_fin[]` contains one entry

Algorithm 1 Non-Delayed Tiled Cholesky Factorization (kernel executing on a CUDA grid)

```

1: Input: matrix  $A \in \mathbb{R}^{n \times n}$  stored in device memory
2:   block size  $b$ , number of tiles per dimension  $p = n/b$ 
3:    $A$  is logically partitioned into tiles  $A_{ij}$ ,  $0 \leq j \leq i < p$ 
4: Global state: signal_fin[] array and d_id_extractor initialized to zero

5:  $id \leftarrow \text{atomicAdd}(\text{d\_id\_extractor}, 1)$  ▷ Dynamic task assignment
6: Map  $id$  to topological tile coordinates  $(i, j)$ 
7: load  $A_{ij}$  from global memory into registers

8: if  $i = j$  then ▷ Diagonal tile
9:   for  $k = 0$  to  $i - 1$  do
10:    while signal_fin[( $i, k$ )] = 0 do
11:      wait
12:    end while
13:    update  $A_{ii} \leftarrow A_{ii} - A_{ik}A_{ik}^\top$ 
14:  end for
15:  factorize  $A_{ii}$ 
16:  store final tile  $A_{ii}$  to global memory
17:  signal_fin[( $i, i$ )]  $\leftarrow 1$ 
18:  return
19: end if

20: if  $j < i$  then ▷ Off diagonal tile
21:   for  $k = 0$  to  $j - 1$  do
22:    while signal_fin[( $i, k$ )] = 0 or signal_fin[( $j, k$ )] = 0 do
23:      wait
24:    end while
25:    update  $A_{ij} \leftarrow A_{ij} - A_{ik}A_{jk}^\top$ 
26:  end for
27:  while signal_fin[( $j, j$ )] = 0 do
28:    wait
29:  end while
30:  solve  $A_{jj} \leftarrow A_{jj}(L_{jj}^\top)^{-1}$ 
31:  store final tile  $A_{ij}$  to global memory
32:  signal_fin[( $i, j$ )]  $\leftarrow 1$ 
33: end if

```

per tile. Each entry is initialized to zero and set to one exactly once when the corresponding tile has completed its final factorization step. Because every tile is exclusively owned by a single CUDA block, no write conflicts occur.

A diagonal tile (i, i) depends on the factorization of tiles (i, k) for $k < i$, while an off-diagonal tile (i, j) depends on the readiness of both (i, k) and (j, k) for $k < j$, and additionally on (j, j) before its triangular solve. These conditions translate into simple load checks of `signal_fin[]` that determine whether a block may proceed or must briefly wait. Crucially, the array `signal_fin[]` entails only one synchronization flag per matrix tile. Because the total size of these flags is extraordinarily small relative to the matrix data, the overhead of this spin-waiting is minimal. Asynchronous flag polling

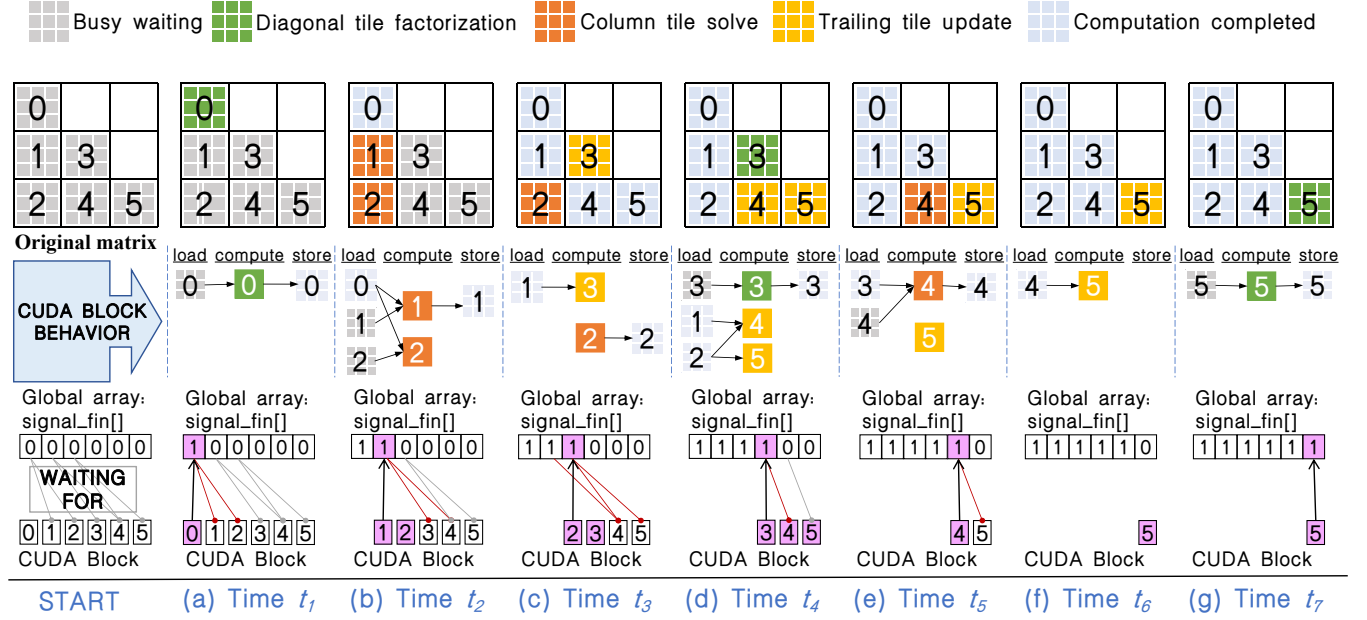


Figure 4: Execution over a 3×3 tiled matrix. Each tile is exclusively owned by a persistent CUDA block, remaining purely resident in on-chip memory. Asynchronous flags (signal_fin) track readiness, enabling an event-driven dataflow where updates proceed immediately when dependencies are met, without coarse-grained barriers.

merely generates highly localized memory traffic, thus preserving the continuous compute-bound pipelines of active warps without congesting the main memory bandwidth.

Tile residency and fused execution. A central efficiency property of this model is that **each tile is loaded from global memory exactly once**. All updates occur while the tile is kept in registers or shared memory, and the final diagonal factorization or triangular solve writes the tile back only once. This fused execution path minimizes memory traffic, avoids repeatedly streaming tiles through the memory hierarchy, and mirrors the behavior of a GEMM microkernel in which data remains hot for the duration of its use.

Deadlock-free dynamic task assignment. A critical challenge in executing a persistent, dependency-driven DAG within a single kernel is avoiding grid deadlock. If CUDA blocks were statically assigned to tiles based on hardware block IDs, scheduling unpredictability could stall active blocks, while the blocks they depend on are indefinitely delayed in the dispatch queue. To guarantee forward progress, our implementation employs a dynamic task-assignment scheme. Through an atomic counter (atomicAdd) maintained in global memory, active CUDA blocks dynamically fetch their tile assignments (acting as a ticket dispenser). This guarantees that tiles are claimed in a strictly monotonic sequence, matching the topological progression of the dependencies and completely eliminating the risk of grid deadlock.

Interpreting the execution timeline. Figure 4 illustrates the resulting asynchronous behavior across seven representative times t_1 – t_7 . Each column of the figure shows: (1) the state of the matrix tiles, (2) the load–compute–store sequence of the CUDA block operating on

a tile, and (3) the global readiness array and which blocks depend on which entries.

At the start, the matrix is partitioned into tiles 0–5, which are dynamically claimed by active CUDA blocks in topological index order. Initially, all entries of the global array signal_fin[] are set to zero, indicating that no tile has yet completed its computation. The gray lines connecting signal_fin[] to the CUDA blocks in the figure denote the readiness dependencies: each block must wait for the tiles on which its computation depends to be marked as completed before proceeding to the next stage. The timeline unfolds as follows:

- **At t_1 ,** tile 0 completes its diagonal factorization (Algorithm 1, lines 15–16) and sets its readiness flag in the global array (line 17). Upon completing all work associated with this tile, the corresponding CUDA block terminates and releases its occupied computational resources. At this point, tiles 1 and 2, which depend on tile 0, observe the update of signal_fin[0] and become eligible to proceed. More generally, all CUDA blocks whose computations depend on tile 0 are unblocked once its readiness flag is set.
- **At t_2 ,** tiles 1 and 2 begin their triangular solves (Algorithm 1, lines 27–30). Tile 1 completes its computation early and sets its readiness flag (lines 31–32), while tile 2 continues executing into time t_3 , illustrating the asynchronous progression of CUDA blocks. The next update of tile 3 depends only on tile 1. Once signal_fin[1] is set, tile 3 becomes eligible to proceed and is unblocked, allowing its CUDA block to advance. Tile 4 also depends on tile 1 and observes the update of the corresponding entry in the global array at this time. However, because its next update additionally depends on

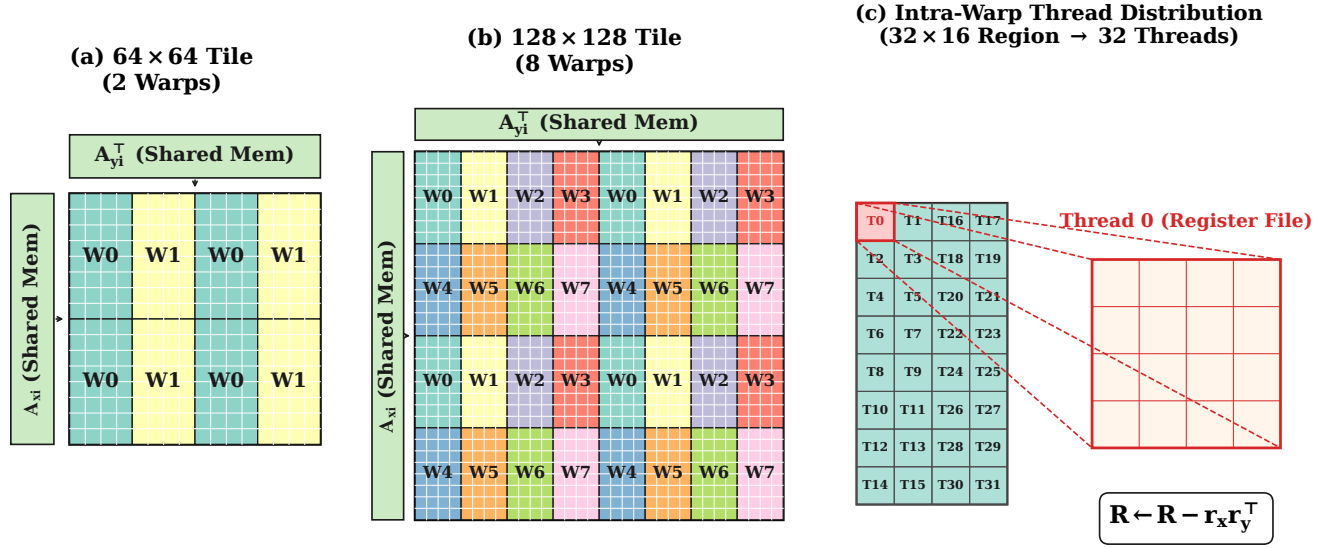


Figure 5: Hierarchical, GEMM-like mapping. (a, b) Spatial distribution of warps across 64×64 and 128×128 accumulator tiles. (c) At the intra-warp level, operands are fetched from shared memory into registers using a 2×2 interleaved thread layout to minimize bank conflicts, replicating optimal GEMM patterns.

tile 2, the completion of tile 1 alone is insufficient to satisfy its dependency conditions. As a result, tile 4 remains blocked and continues waiting.

- At t_3 , the completion of tile 1 enables tile 3 to begin its trailing update (Algorithm 1, lines 9–14), even though tile 2 is still active. This demonstrates that updates and solves naturally interleave under the non-delayed execution model. At this point, tile 2 finally completes its computation, thereby satisfying all dependency conditions for tile 4. As a result, tile 4 becomes eligible to proceed with its next update, which depends on both tiles 1 and 2. Similarly, the update of tile 5 depends only on tile 2, and its dependency condition is likewise satisfied, allowing tile 5 to advance to its update phase.
- At t_4 , tile 3 proceeds to its diagonal factorization immediately after completing its trailing updates (Algorithm 1, line 15), since this operation does not depend on any preceding tiles. In addition, because tile 2 had already completed its calculation in the previous time step, the dependency conditions for both tile 4 and tile 5 are fully satisfied. As a result, tile 4 (Algorithm 1, lines 21–26) and tile 5 (Algorithm 1, lines 9–14) are enabled to begin their trailing updates.
- At t_5 , tile 4 completes its trailing update. Because tile 3, on which its triangular solve depends, has already finished its diagonal factorization, tile 4 can immediately proceed to its triangular solve (Algorithm 1, lines 27–32). At the same time, the first trailing update of tile 5 has not yet completed, and tile 5 therefore remains in the update phase.
- At t_6 , tile 5 is still performing its trailing update (Algorithm 1, lines 9–14). However, unlike at times t_4 and t_5 , where its update is driven by the data produced by tile 2, the update at t_6 is now based on the triangular solve result of tile 4.

This transition reflects the progression of the update loop for tile 5 across successive dependency stages. The change in dependency can also be observed in the figure: after tile 5 initiates its first update at t_4 , the dependency indicator associated with its CUDA block shifts to point to the entry of `signal_fin[]` corresponding to tile 4, indicating that subsequent updates must wait for the completion of tile 4.

- At t_7 , tile 5 has completed all of its trailing updates. Because tile 5 lies on the diagonal and its final calculation does not depend on any other tiles, it can immediately proceed with its diagonal factorization (Algorithm 1, lines 15–17). This step completes the computation associated with tile 5 and marks the completion of the entire factorization pipeline.

Across the entire execution, the number of active CUDA blocks decreases as tiles complete their work and exit. Because every block progresses solely based on its local dependency conditions, the factorization exhibits naturally high concurrency, with diagonal factorizations, triangular solves, and trailing updates all overlapping whenever the dataflow permits. This fully asynchronous, dependency-driven behavior is a defining characteristic of the non-delayed formulation.

3.2 Mapping the Update to a GEMM-Like Kernel

The update applied to each trailing tile in the non-delayed Cholesky factorization is carried out by a microkernel that follows the same hierarchical organization used in high-performance GEMM implementations. The tile is decomposed into warp-level regions, and each warp is further partitioned into thread-level fragments (as shown in Figure 5). To optimize memory-level parallelism and minimize shared memory bank conflicts, threads are assigned using a strict 2×2 interleaved layout. Each thread computes a fixed 4×4

subfragment entirely in registers, so that a warp collectively produces a 32×16 microtile. This mapping is identical to the spatial decomposition employed by SIMT-based GEMM microkernels on CUDA cores. The only difference lies in the accumulation operator. Instead of evaluating $C \leftarrow C + AB$, the update performs

$$A_{xy} \leftarrow A_{xy} - A_{xi}A_{yi}^T,$$

while all data movement, reuse structure, and warp-level scheduling follow the same pattern as GEMM.

A key distinction from a conventional GEMM kernel is that the updated tile is not written back to global memory after each multiplication step. In a standard GEMM pipeline, the result fragments are accumulated in registers and eventually stored to global memory. In the Cholesky factorization, however, the updated tile must serve as the input for the subsequent triangular solve or diagonal factorization. The fragment layout produced by the GEMM microkernel is therefore not directly suitable for the next computation stage. To avoid writing the partially updated tile back to global memory and reloading it in a different layout, the kernel stores the updated tile into a shared memory buffer that is large enough to hold the entire tile. The tile is written in a column-major format so that the subsequent computation stages can load the required columns directly into registers without additional global memory traffic.

This organization allows the update kernel to inherit the computational efficiency of GEMM while preparing the data for the next phase of the factorization. The warp-level and thread-level decompositions follow the GEMM design, the intermediate results remain in registers during the accumulation loop, and the final tile is staged in shared memory in a layout natively suited for the subsequent factorization steps. The choice of tile sizes and shared memory allocation is discussed further in Section 3.3.

3.3 Thread Block Configuration and Memory Alignment

Translating our algorithmic model into a high-performance continuous kernel requires aligning the mathematical tile size b with the physical GPU thread block geometry. Based on classical GEMM tuning heuristics, we constrain our design space to $b = 64$ and $b = 128$. The larger 128×128 tile size maximizes arithmetic intensity and thoroughly amortizes register load instructions. Conversely, the 64×64 configuration yields four times the number of tiles, significantly increasing grid-level parallelism to saturate highly-scaled multiprocessor arrays during the trailing updates.

Furthermore, fusing the updating, solving, and factorization stages mandates that each $b \times b$ tile resides entirely in shared memory once fetched. For FP32 precision, this requires a baseline allocation of at least 16 KB (for $b = 64$) or 64 KB (for $b = 128$) of shared memory per thread block, with actual consumption being slightly higher depending on the specific microkernel buffering requirements. Although exceeding 64 KB is a substantial architectural allocation, empirical profiling via NVIDIA Nsight Compute (NCU) reveals that the theoretical thread occupancy is overwhelmingly bound by the strict register limits required to sustain the complex instruction pipelines of the fused matrix arithmetic. As a consequence, utilizing large shared memory capacity acts to stage

data and completely avoid the intermediate global memory round-trips that plague traditional delayed-update schemes, all without inducing any secondary occupancy penalties.

4 Experiments

This section evaluates our non-delayed Cholesky factorization against state-of-the-art implementations. Beyond demonstrating superior end-to-end throughput against industry-standard libraries, we quantify its structural efficiency by benchmarking its underlying GEMM microkernels. Finally, we analyze the micro-architectural execution profile—highlighting reduced off-chip memory traffic, eliminated kernel launch latency, and fine-grained synchronization resilience—and briefly discuss the design’s architectural portability.

4.1 Experimental Setup

We evaluate four GPUs representing recent NVIDIA and AMD architectures: the A100 PCIe with 19.5 TFLOP/s peak FP32 throughput and 1555 GB/s of memory bandwidth, the H100 SXM5 with 66.9 TFLOP/s and 3352 GB/s, the RTX 5090 with 104.8 TFLOP/s and 1793 GB/s, and the AMD Radeon AI PRO R9700 with 47.8 TFLOP/s and 640 GB/s. All experiments are strictly focused on single-device maximal throughput. For comprehensive baseline comparisons, we benchmark against the highly optimized vendor library cuSOLVER/rocSOLVER/hipSOLVER and the open-source MAGMA library. To ensure full clarity regarding the exact software interfaces invoked during our evaluation, both the hardware specifications and the targeted subroutines are explicitly detailed in Table 1.

Table 1: Hardware specifications and evaluated libraries.

Platform	A100	H100	RTX 5090	R9700
FP32 Peak (TFLOPS)	19.5	66.9	104.8	47.8
Bandwidth (GB/s)	1555	3352	1793	640
Evaluated FP32 Routines				
cuSOLVER (v12.9)	cusolverDnSpotrf		-	
rocSOLVER (v7.2)	-		rocsolver_spotrf	
hipSOLVER (v7.2)	-		hipsolverSpotrf	
MAGMA (v2.9)	magma_spotrf_native			
Ours	Non-Delayed Cholesky ($b = 64, 128$)			

4.2 End-to-End Performance and Structural Efficiency

Figure 6 presents the overarching results of our study, where we compare the FP32 Cholesky factorization performance against two highly-optimized baselines: NVIDIA’s cusolverDnSpotrf and MAGMA’s magma_spotrf_native. Across all platforms, the non-delayed paradigm demonstrates a consistent and significant performance advantage over the traditional delayed methodology.

On the RTX 5090, our $b = 128$ variant achieves a peak speedup of 2.28× over cuSOLVER at $n = 5632$. On the H100, it reaches the peak advantage at $n = 6400$ with a speedup of 1.85×. On the A100, where cuSOLVER is exceptionally well-tuned, our variant is still 1.51× faster at peak.

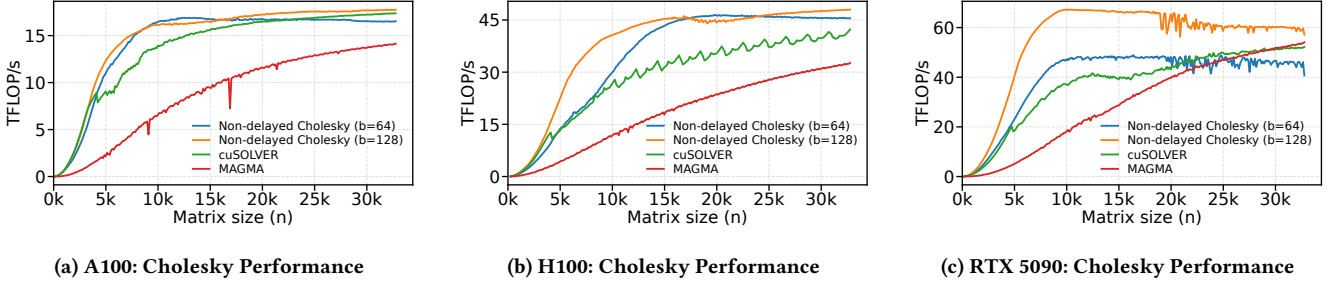


Figure 6: FP32 Overall Cholesky performance. The single persistent kernel robustly dominates traditional delayed methods across the board, peaking at $1.51\times$, $1.85\times$, and $2.28\times$ speedups relative to cuSOLVER on the A100, H100, and RTX 5090, respectively.

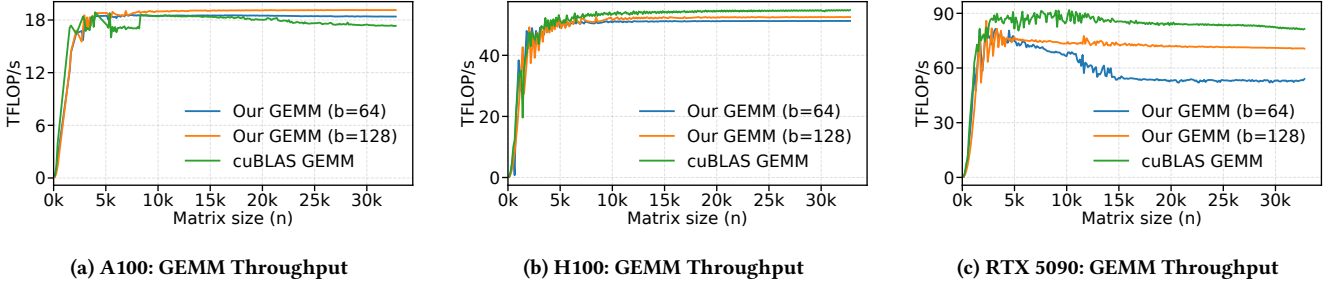


Figure 7: Baseline FP32 GEMM throughput. While our hand-written SIMT kernels match cuBLAS on A100 and H100, they fall behind on the RTX 5090. This performance-gap provides a critical baseline to evaluate algorithmic conversion efficiency.

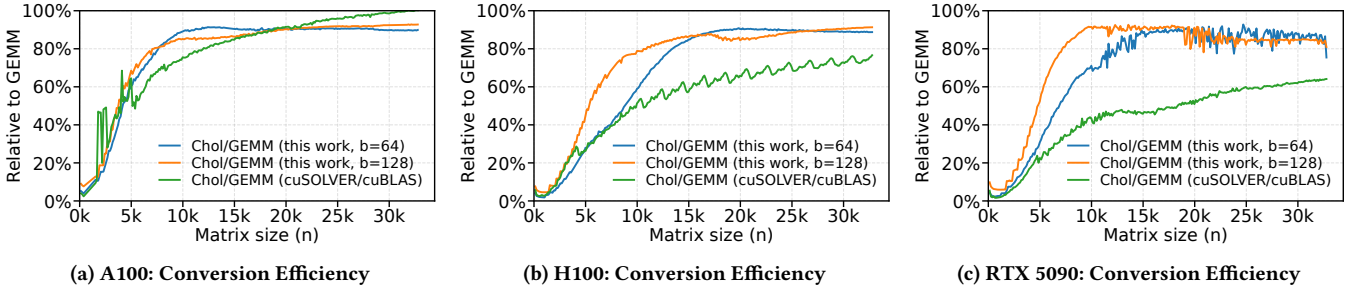


Figure 8: Conversion efficiency: the fraction of raw GEMM throughput strictly translated into overall Cholesky performance. Despite varying base GEMM speeds, our non-delayed structure universally preserves $> 90\%$ conversion. However, the heavy off-chip memory traffic from repeated kernel launches heavily degrades cuSOLVER’s efficiency on the bandwidth-constrained RTX 5090 architecture.

Structural Efficiency and the 5090 Anomaly. The absolute speedups alone do not fully capture the algorithmic power of the non-delayed structure. Since matrix factorization throughput is theoretically bounded by its underlying matrix multiplication capability, structural efficiency is best measured by how well an algorithm translates available raw GEMM performance into end-to-end throughput.

To quantify this, Figure 7 reports the performance of our functional, standalone custom SIMT GEMM against NVIDIA’s premier cuBLAS library. Figure 8 then plots the ratio of the end-to-end Cholesky throughput normalized by these respective GEMM baselines. Although closed-source libraries like cuSOLVER may internally employ distinct backends (e.g., CUTLASS) rather than in-

voking cuBLAS directly, normalizing their performance against cuBLAS provides a strictly fair and rigorous metric for how effectively they utilize the vendor’s peak theoretical limits. On the A100 and H100, our GEMM tightly tracks cuBLAS, allowing our algorithm to dominate entirely due to a consistently swift structural utilization rate.

However, the RTX 5090 scenario reveals our design’s most compelling advantage. As Figure 7 (right) shows, a noticeable performance gap exists between our unoptimized consumer-grade GEMMs and the heavily-tuned cuBLAS library. Despite this foundational disadvantage, our overall Cholesky implementation crushes cuSOLVER by $2.28\times$. The explanation lies entirely in Figure 8 (right): our single-kernel structure systematically converts $> 90\%$ of its

available GEMM capability into useful Cholesky factorization, by bypassing macroscopic global memory barrier costs. In contrast, the off-chip memory traffic in the delayed model causes cuSOLVER’s efficiency to plunge to roughly 60%.

This stark contrast forcefully proves that the single-kernel integration perfectly eliminates the inherent data movement stalls of traditional DAG-based schedulers. Rather than detailing an isolated hardware artifact, the remaining sections will explicitly isolate the architectural sources of this structural victory: profiling the precise elimination of launch overheads and fine-grained synchronizations (Section 4.3), evaluating memory I/O and spatial locality (Section 4.4), and validating architectural portability (Section 4.6).

4.3 Kernel Launch Overhead Analysis

Before analyzing the absolute throughput, we first highlight a fundamental architectural advantage of our approach: the elimination of kernel launch and synchronization overhead. As illustrated in Figure 9, traditional algorithms (e.g., cuSOLVER and MAGMA) require repeated host-device synchronizations that scale with the matrix dimension. In stark contrast, both of our non-delayed variants ($b = 64$ and $b = 128$) execute the entire factorization using exactly one kernel invocation, regardless of the matrix size. This single-kernel approach completely eradicates the host-side scheduling latency and repetitive global memory barriers, establishing the structural basis for our high sustained efficiency.

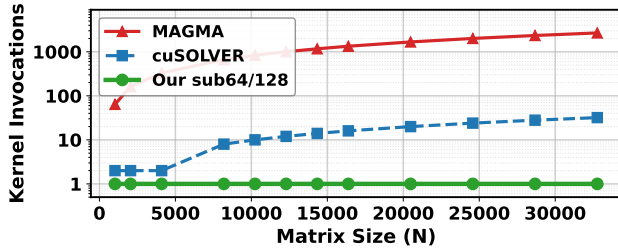


Figure 9: Kernel launch invocations required for a complete Cholesky factorization (y-axis is in log scale). While delayed approaches (MAGMA and cuSOLVER) require launch counts that scale with the number of matrix tiles—thereby incurring structural susceptibility to host-side latency—our algorithm handles the entire procedure within a singular, monolithic launch.

4.4 Memory I/O and Locality Optimization

To understand the memory access characteristics driving our algorithm’s significant performance advantage, we analyze the memory instruction breakdown across the GPU memory hierarchy. We quantify this traffic using SASS-level memory execution instructions, as natively tabulated by the NCU profiler. Relying on explicit instruction counts—rather than dynamic byte volumes that fluctuate with L1/L2 cache eviction policies—provides a deterministic measure of the algorithm’s intrinsic memory traffic. A fundamental bottleneck in traditional panel-based Cholesky factorization is the heavy reliance on the global memory system to exchange data

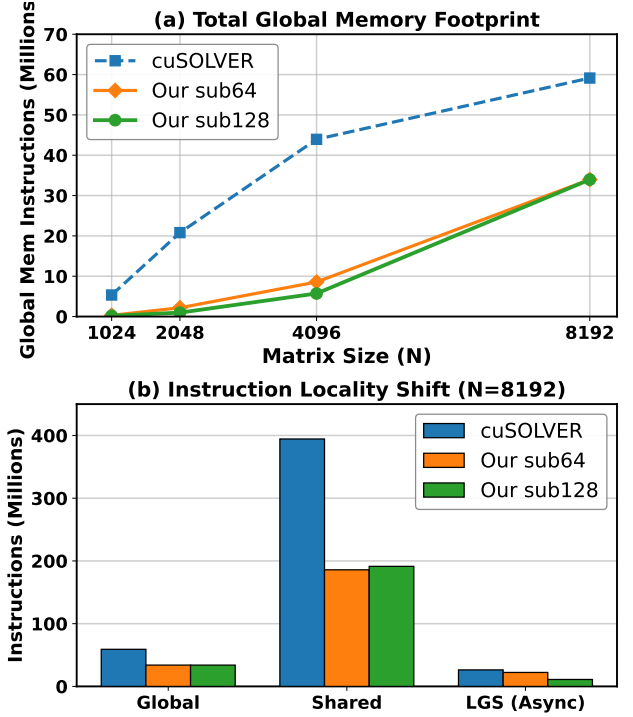


Figure 10: Memory I/O footprint and locality analysis. (a) Scaling of global memory instructions vs. matrix dimension N . (b) Traffic breakdown at $N = 8192$. By anchoring data residently on-chip rather than flushing intermediate panels, the fused implementation yields a roughly 50% reduction in both GMEM accesses and SMEM transactions.

across different algorithmic steps (e.g., panel factorization, trailing matrix updates). By executing the entire factorization within a single unified kernel (as demonstrated in Section 4.3), our algorithm keeps the active data stationary in the high-speed on-chip memory. In this micro-architectural breakdown, we specifically benchmark against cuSOLVER, as it represents the most heavily vendor-tuned, closed-source target and constitutes our primary performance competitor.

Figure 10 illustrates this memory locality optimization. Figure 10(a) shows the total executed global memory instructions as the matrix size N increases. While cuSOLVER’s global memory footprint scales rapidly due to the repeated flushing and reloading of intermediate panels across numerous kernel launches, our algorithm’s global memory traffic grows at a significantly slower rate, primarily bounded by the mandatory initial cold fetches and final out-of-core write-backs.

Figure 10(b) further breaks down the instruction locality at $N = 8192$. The data reveals that our algorithm does not merely shift global memory accesses to shared memory; it actively eliminates a massive amount of redundant data movement. Compared to cuSOLVER, our method cuts the executed global memory instructions by nearly half and simultaneously reduces shared memory access instructions by over 50%. This substantial compression of the

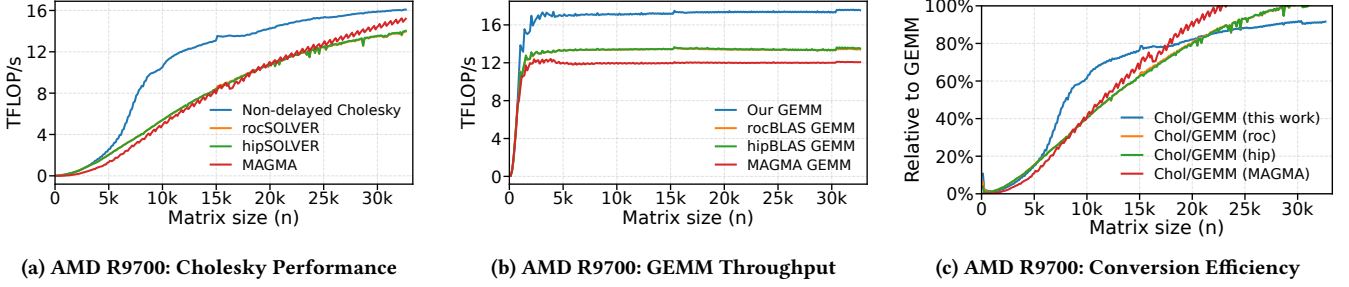


Figure 11: FP32 performance on the AMD Radeon AI PRO R9700. Our non-delayed implementation seamlessly transitions to the AMD ROCm software stack and surpasses MAGMA and both vendor mechanisms (rocSOLVER and hipSOLVER), achieving the highest absolute throughput without AMD-specific microkernel tuning. Furthermore, it preserves high conversion efficiency, maintaining relative independence from the underlying architecture’s characteristics.

overall instruction-level I/O footprint corroborates the superior end-to-end performance and GEMM-conversion efficiency presented in Section 4.2.

4.5 Fine-Grained Synchronization Overhead

While our unified kernel eliminates global memory and launch overheads, it introduces explicit dependency tracking via lightweight busy-waiting. To assess the pipeline impact of this synchronization, we performed instruction-level profiling using NCU under a heavy workload ($N = 8192$). Hardware scheduler sampling reveals that `stall_wait` cycles—which cover all control-flow spin-loops and map directly to our busy-wait design—constitute merely **4.81%** of the total warp pipeline stalls, comparable to standard barrier stalls (`__syncthreads()` at 3.55%). This low stall proportion indicates that these fine-grained synchronization events resolve quickly and are not heavily contended. Consequently, the transient spin-waiting consumes a negligible fraction of the SM’s instruction issue bandwidth, allowing the arithmetic and memory operations of concurrent active warps to proceed efficiently without severe pipeline interference.

4.6 Architectural Portability and Robustness

The performance of our non-delayed factorization across architecturally diverse GPUs strictly validates its profound algorithmic portability. Beyond the consistent behavior on NVIDIA platforms (A100, H100, and RTX 5090), we further deployed our algorithm on the latest AMD Radeon AI PRO R9700 GPU.

As shown in Figure 11, we additionally evaluate our non-delayed Cholesky implementation within the AMD ROCm ecosystem by porting it to HIP. To obtain the best baseline performance for the official manufacturer libraries (the ROCm rocSOLVER backend and its frontend hipSOLVER) and MAGMA, we manually set `ROCBLAS_USE_HIPBLASLT=0` in the testing environment. However, we believe that their underlying GEMM implementation might still be suboptimal. Against these fully optimized baselines, our algorithm continues to deliver superior throughput across all evaluated matrix dimensions, achieving a peak speedup of **2.16×**.

Perhaps the most significant finding from the end-to-end evaluation is not just the absolute performance scaling, but the consistency of the *relative* efficiency. As demonstrated in Section 4.2,

our non-delayed factorization consistently converts over 90% of its underlying GEMM kernel’s throughput into final performance, regardless of the vendor platform. This stands in stark contrast to traditional libraries whose efficiency relative to raw GEMM fluctuates significantly depending on the underlying hardware architecture’s memory bandwidth and scheduling latencies. By executing the entire factorization within a single unified kernel, our algorithm eliminates the variable host-side scheduling latencies, yielding a stable, hardware-agnostic efficiency profile that transfers seamlessly across different GPU generations and discrete vendors. This demonstrates that the extreme high performance of our approach is an intrinsic property of the algorithm itself, fully orthogonal to specific hardware pipelines.

5 Related Work

Task-Based DAG Scheduling and GPU Offloading. Historically, exploiting multicore processors and hybrid CPU-GPU systems for dense matrix factorizations heavily relied on viewing the algorithm as a DAG of BLAS tasks. Libraries such as PLASMA [8] and early versions of MAGMA [4, 32] dynamically scheduled these tasks across processing units to hide computational latency. This dynamic scheduling was further supplemented by pipelined execution schemes [28] and distributed approaches such as SLATE [19]. In these hybrid models, the host CPU typically manages the dependency resolution and dispatches heavy routines (like TRSM or GEMM) to the GPU [35]. While frameworks eventually migrated toward ‘native’ GPU execution [2, 24] to avoid slow PCIe transfers, they still retain a host-side DAG execution engine. This traditional ‘delayed update’ approach fundamentally requires multiple separate kernel launches, incurring inevitable host scheduling latency and repeated global memory synchronizations between task generations.

Memory Layouts and Algorithmic Blocking. Fundamentally, modern linear algebra relies on blocked iterations to convert memory-bound operations into compute-bound Level-3 BLAS routines [15, 22]. Various sophisticated execution schemes have been explored, inclusive of recursive formulations [6], and advanced batched paradigms for concurrent small-matrix processing [1, 3, 14, 20, 25, 27], which often process thousands of independent matrices simultaneously to saturate the GPU. Other advanced optimizations include mixed-precision with iterative refine-

ment [11, 43], and tile low-rank approximations [9, 10]. Hardware-specific adaptations also target modern accelerators like Xeon Phi [17, 26] and AI chips/Tensor Cores [33, 39, 46]. Additional refinements emphasize customized layouts like the Rectangular Full Packed format [23]. Although these works successfully optimize individual computational tiles and layouts, they generally operate under the underlying assumption that intermediate tiles must be routinely flushed to global memory before the next algorithmic step begins, leaving the macro-level memory I/O bottleneck unresolved.

Single-Kernel Execution and Synchronization-Free Algorithms. Within the high-performance computing domain, the overhead of host-driven synchronization has prompted research into GPU-centric execution models. The ‘Persistent Threads’ programming style [21] demonstrated the viability of bypassing the hardware scheduler by launching long-living thread blocks that fetch and synchronize work intrinsically on the GPU. Several solvers have since adopted single-kernel, resource-aggregation, or persistent designs [12, 16] to minimize dispatch overhead. In the specific context of dense matrix factorizations, recent efforts have begun exploring kernel fusion to avoid host-side scheduling. For instance, IA-Chol [13] successfully fuses Cholesky decomposition into a single GPU kernel, yet it still relies on explicit block-level synchronization barriers that inherently mimic the traditional ‘delayed update’ sequence. Additionally, Yang et al. [44] proposed a tiled-grained mixed precision single kernel conjugate gradient solver Mille-feuille. Our work advances this trajectory by demonstrating that a *pure non-delayed*, tile-resident Cholesky factorization can be seamlessly mapped into a persistent GPU kernel, replacing coarse-grained barrier synchronizations or explicit OS-level DAG execution with asynchronous, fine-grained on-chip memory dependencies. Philosophically, this tile-level asynchronous event-driven design echoes the success of modern synchronization-free paradigms originally pioneered in sparse triangular solves [30, 31, 34, 40, 45], translating these fine-grained concurrency concepts into dense matrix factorizations. Furthermore, since state-of-the-art sparse direct solvers frequently partition matrices into dense blocks and invoke localized dense computational kernels [18, 29, 42, 47], the high-throughput, non-delayed factorization introduced in this paper possesses strong potential for broader adoption as a core processing routine within these sparse solver frameworks.

6 Conclusion

We in this paper have proposed a non-delayed algorithm for accelerating Cholesky factorization on modern GPUs. Specifically, we developed a GEMM-like, single-kernel execution model to maximize on-chip data residency and eliminate kernel launch overheads. The experimental results on diverse GPU platforms (NVIDIA A100, H100, RTX 5090 and AMD R9700) show that our approach brings significant speedups over state-of-the-art vendor libraries.

Acknowledgments

We appreciate the valuable comments of all reviewers. Weifeng Liu is the corresponding author of this paper. This work has been partially supported by the National Natural Science Foundation of China (Grant No. U23A20301 and No. 62372467).

References

- [1] Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, and Jack Dongarra. 2016. Performance tuning and optimization techniques of fixed and variable size batched Cholesky factorization on GPUs. *Procedia Computer Science* 80 (2016), 119–130.
- [2] Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, and Jack Dongarra. 2017. Fast Cholesky factorization on GPUs for batch and native modes in MAGMA. *Journal of Computational Science* 20 (2017), 85–93.
- [3] Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, and Jack Dongarra. 2018. Optimizing GPU kernels for irregular batch workloads: a case study for cholesky factorization. In *HPEC '18*. 1–7.
- [4] Emmanuel Agullo, Olivier Beaumont, Lionel Eyraud-Dubois, and Suraj Kumar. 2016. Are static schedules so bad? a case study on cholesky factorization. In *IPDPS '16*. 1021–1030.
- [5] Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov. 2009. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *Journal of Physics: Conference Series* 180, 1 (2009), 012037.
- [6] Bjarne Stig Andersen, Jerzy Waśniewski, and Fred G Gustavson. 2001. A recursive formulation of Cholesky factorization of a matrix in packed storage. *ACM Transactions on Mathematical Software* 27, 2 (2001), 214–244.
- [7] Edward Anderson, Zhaojun Bai, Christian Bischof, L Susan Blackford, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, Alan McKenney, et al. 1999. *LAPACK Users' guide*.
- [8] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. 2009. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel computing* 35, 1 (2009), 38–53.
- [9] Qinglei Cao, Rabab Alomairi, Yu Pei, George Bosilca, Hatem Ltaief, David Keyes, and Jack Dongarra. 2022. A framework to exploit data sparsity in tile low-rank cholesky factorization. In *IPDPS '22*. 414–424.
- [10] Qinglei Cao, Yu Pei, Thomas Herault, Kadir Akbudak, Aleksandr Mikhalev, George Bosilca, Hatem Ltaief, David Keyes, and Jack Dongarra. 2019. Performance analysis of tile low-rank cholesky factorization using parsec instrumentation tools. In *ProTools '19*. 25–32.
- [11] Erin Carson, Tomáš Gergelits, and Ichitaro Yamazaki. 2022. Mixed precision s-step Lanczos and conjugate gradient algorithms. *Numerical Linear Algebra with Applications* 29, 3 (2022), e2425.
- [12] 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 Computing* 83 (2019), 73–92.
- [13] 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 *ICS '25*. 135–148.
- [14] Tingxing Dong, Azzam Haidar, Stanimire Tomov, and Jack Dongarra. 2014. A fast batched Cholesky factorization on a GPU. In *ICPP '14*. 432–440.
- [15] Jack Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain S Duff. 1990. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software* 16, 1 (1990), 1–17.
- [16] Joseph Dorris, Jakub Kurzak, Piotr Luszczek, Asim YarKhan, and Jack Dongarra. 2016. Task-based Cholesky decomposition on knights corner using OpenMP. In *ISC '16*. 544–562.
- [17] Joseph Dorris, Asim YarKhan, Jakub Kurzak, Piotr Luszczek, and Jack Dongarra. 2018. Task based Cholesky decomposition on Xeon Phi architectures using OpenMP. *International Journal of Computational Science and Engineering* 17, 3 (2018), 310–323.
- [18] Xu Fu, Bingbin Zhang, Tengcheng Wang, Wenhao Li, Yuechen Lu, Enxin Yi, Jianqi Zhao, Xiaohan Geng, Fangying Li, Jingwen Zhang, Zhou Jin, and Weifeng Liu. 2023. PanguLU: A Scalable Regular Two-Dimensional Block-Cyclic Sparse Direct Solver on Distributed Heterogeneous Systems. In *SC '23*. 1–14.
- [19] Mark Gates, Jakub Kurzak, Ali Charara, Asim YarKhan, and Jack Dongarra. 2019. SLATE: Design of a modern distributed and accelerated linear algebra library. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–18.
- [20] Mark Gates, Jakub Kurzak, Piotr Luszczek, Yu Pei, and Jack Dongarra. 2017. Autotuning batch Cholesky factorization in CUDA with interleaved layout of matrices. In *IPDPSW '17*. 1408–1417.
- [21] Kshitij Gupta, Jeff A Stuart, and John D Owens. 2012. A study of persistent threads style GPU programming for GPGPU workloads. In *InPAR '12*. 1–14.
- [22] Fred G Gustavson, Jerzy Waśniewski, Jack Dongarra, Jose R Herrero, and Julien Langou. 2013. Level-3 Cholesky factorization routines improve performance of many Cholesky algorithms. *ACM Transactions on Mathematical Software* 39, 2 (2013), 1–10.
- [23] Fred G Gustavson, Jerzy Waśniewski, Jack Dongarra, and Julien Langou. 2010. Rectangular full packed format for Cholesky's algorithm: factorization, solution, and inversion. *ACM Transactions on Mathematical Software* 37, 2 (2010), 1–21.
- [24] Azzam Haidar, Ahmad Abdelfattah, Stanimire Tomov, and Jack Dongarra. 2017. High-performance Cholesky factorization for GPU-only execution. In *GPGPU*

- '17. 42–52.
- [25] Azzam Haidar, Ahmad Abdelfattah, Mawussi Zounon, Stanimire Tomov, and Jack Dongarra. 2017. 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 (2017), 973–984.
 - [26] Azzam Haidar, Stanimire Tomov, Konstantin Arturov, Murat Guney, Shane Story, and Jack Dongarra. 2016. LU, QR, and Cholesky factorizations: Programming model, performance analysis and optimization techniques for the Intel Knights Landing Xeon Phi. In *HPEC '16*. 1–7.
 - [27] Jakub Kurzak, Hartwig Anzt, Mark Gates, and Jack Dongarra. 2015. Implementation and tuning of batched Cholesky factorization and solve for NVIDIA GPUs. *IEEE Transactions on Parallel and Distributed Systems* 27, 7 (2015), 2036–2048.
 - [28] Jakub Kurzak and Jack Dongarra. 2006. Pipelined Shared Memory Implementation of Linear Algebra Routines with arbitrary Lookahead-LU, Cholesky, QR. In *PARA '06*.
 - [29] Yida Li, Siwei Zhang, Yiduo Niu, Yang Du, Qingxiao Sun, Zhou Jin, and Weifeng Liu. 2026. Trojan Horse: Aggregate-and-Batch for Scaling Up Sparse Direct Solvers on GPU Clusters. In *PPoPP '26*. 369–383.
 - [30] Weifeng Liu, Ang Li, Jonathan Hogg, Iain S Duff, and Brian Vinter. 2016. A synchronization-free algorithm for parallel sparse triangular solves. In *Euro-Par '16*. 617–630.
 - [31] Weifeng Liu, Ang Li, Jonathan D Hogg, Iain S Duff, and Brian Vinter. 2017. Fast synchronization-free algorithms for parallel sparse triangular solves with multiple right-hand sides. *Concurrency and Computation: Practice and Experience* 29, 21 (2017), e4244.
 - [32] Hatem Ltaief, Stanimire Tomov, Rajib Nath, and Jack Dongarra. 2010. Hybrid multicore cholesky factorization with multiple gpu accelerators. *IEEE Transaction on Parallel and Distributed Systems* 48 (2010).
 - [33] 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.
 - [34] Zhengyang Lu and Weifeng Liu. 2023. Tilesptsv: a tiled algorithm for parallel sparse triangular solve on gpus. *CCF Transactions on High Performance Computing* 5, 2 (2023), 129–143.
 - [35] Rajib Nath, Stanimire Tomov, and Jack Dongarra. 2010. An improved magma gemm for fermi graphics processing units. *The International Journal of High Performance Computing Applications* 24, 4 (2010), 511–515.
 - [36] NVIDIA. 2024. cuBLAS Library. <https://developer.nvidia.com/cublas>.
 - [37] NVIDIA. 2024. cuSOLVER Library. <https://developer.nvidia.com/cusolver>.
 - [38] NVIDIA. 2024. CUTLASS: CUDA Templates for Linear Algebra Subroutines. <https://github.com/NVIDIA/cutlass>.
 - [39] Lu SHI, Gaoyuan ZOU, Siqi WU, and Shaoshuai ZHANG. 2025. High performance Cholesky factorization on emerging GPU architectures using Tensor Cores. *Computer Engineering & Science* 47, 7 (2025), 1170.
 - [40] Jiya Su, Feng Zhang, Weifeng Liu, Bingsheng He, Ruofan Wu, Xiaoyong Du, and Rujia Wang. 2020. CapelliniSpTRSV: a thread-level synchronization-free sparse triangular solve on GPUs. In *ICPP '20*. 1–11.
 - [41] Hemeng Wang, Yang Du, Sidu Li, Xiaowen Tian, Qingxiao Sun, and Weifeng Liu. 2025. KAMI: Communication-Avoiding General Matrix Multiplication within a Single GPU. In *SC '25*. 1572–1589.
 - [42] Tengcheng Wang, Wenhao Li, Haojie Pei, Yuying Sun, Zhou Jin, and Weifeng Liu. 2023. Accelerating Sparse LU Factorization with Density-Aware Adaptive Matrix Multiplication for Circuit Simulation. In *DAC '23*. 1–6.
 - [43] Ichitaro Yamazaki, Stanimire Tomov, and Jack Dongarra. 2015. Mixed-precision Cholesky QR factorization and its case studies on multicore CPU with multiple GPUs. *SIAM Journal on Scientific Computing* 37, 3 (2015), C307–C330.
 - [44] Dechuang Yang, Yuxuan Zhao, Yiduo Niu, Weile Jia, En Shao, Weifeng Liu, Guangming Tan, and Zhou Jin. 2024. Mille-feuille: A tile-grained mixed precision single-kernel conjugate gradient solver on gpus. In *SC '24*. 1–16.
 - [45] Feng Zhang, Jiya Su, Weifeng Liu, Bingsheng He, Ruofan Wu, Xiaoyong Du, and Rujia Wang. 2021. Yuenyeungsptsv: a thread-level and warp-level fusion synchronization-free sparse triangular solve. *IEEE Transactions on Parallel and Distributed Systems* 32, 9 (2021), 2321–2337.
 - [46] Shaoshuai Zhang, Vivek Karihaloo, and Panrui Wu. 2020. Basic Linear Algebra Operations on TensorCore GPU. In *Scala '20*. 44–52.
 - [47] Jianqi Zhao, Yao Wen, Yuchen Luo, Zhou Jin, Weifeng Liu, and Zhenya Zhou. 2021. SFLU: Synchronization-Free Sparse LU Factorization for Fast Circuit Simulation on GPUs. In *DAC '21*. 37–42.

Received 16 December 2025; accepted 24 February 2026