StructILU: Dependency-Preserving Incomplete LU with Hierarchical Parallelism for Structured Grid PDEs on GPUs

Hao Luo
School of Mathematical Sciences
Peking University
Beijing, China
lhsms@pku.edu.cn

Qianchao Zhu
School of Mathematical Sciences
Peking University
Beijing, China
dysania@pku.edu.cn

Xiaochen Hao
School of Computer Science
Peking University
Beijing, China
xiaochen.hao@stu.pku.edu.cn

Chunxi Lei
School of Mathematical Sciences
Peking University
Beijing, China
leichunxi@stu.pku.edu.cn

Chengdi Ma
School of Mathematical Sciences
Peking University
Beijing, China
mcd2020@stu.pku.edu.cn

Chenchen Zhang
School of Mathematical Sciences
Peking University
Beijing, China
zhangchenchen@stu.pku.edu.cn

Yun Liang
School of Integrated Circuits
Peking University
Beijing, China
ericlyun@pku.edu.cn

Chao Yang*
School of Mathematical Sciences
Peking University
Beijing, China
PKU-Changsha Institute for
Computing and Digital Economy
Changsha, China
chao_yang@pku.edu.cn

Abstract

The Incomplete LU (ILU) computation is a crucial component for solving large-scale sparse linear systems arising from partial differential equations (PDEs), many of which are discretized on structured grids. However, due to inherent loop-carried data dependencies in ILU computation, implementing it on GPUs with massive computing units poses significant challenges. Existing methods either experience a lower convergence rate or suffer from insufficient parallelism, and few efforts have been made to exploit the geometric information of the PDE grid. In this work, we introduce StructILU for dependency-preserving structured grid ILU computations on GPUs. StructILU constructs hierarchical parallelism with three tiers: grid, intra-row, and intra-element. An acyclic dependency task mapping maximizes grid parallelism, while latency-aware pipeline methods optimize intra-row parallelism, and Tensor Core-optimized matrix operations enhance intra-element parallelism. Aligned with hierarchical parallelism, memory transformations are designed to utilize high memory bandwidth. Experiment results show that StructILU demonstrates up to 6.5x and 17.7x (average 2.11x and 3.88x) speedups compared to existing general methods for triangular solve and factorization, respectively, and can achieve an average speedup of 2.02x in end-to-end ILU-preconditioned iterative solvers.

*Corresponding author.

ICS '25, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-1537-2/25/06
https://doi.org/10.1145/3721145.3725745

CCS Concepts

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

Keywords

Incomplete LU, GPU, Structured Grid, PDE, Iterative Solver, Sparse Triangular Solve

ACM Reference Format:
Hao Luo, Qianchao Zhu, Xiaochen Hao, Chunxi Lei, Chengdi Ma, Chenchen Zhang, Yun Liang, and Chao Yang. 2025. StructILU: Dependency-Preserving Incomplete LU with Hierarchical Parallelism for Structured Grid PDEs on GPUs. In *2025 International Conference on Supercomputing (ICS '25), June 08–11, 2025, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3721145.3725745

Table 1: Comparison with existing parallel ILU work.

	Factorization	Triangular Solve	Dependency-Preserving	Structured Grid Optimized	Target Platform
ParILU [12, 13]	✔	✘	✘	✘	GPU
ParILUT [4]	✔	✘	✘	✘	GPU
MC-ILU [33, 48]	✔	✔	✘	✘	GPU
Sync-Free [37]	✘	✔	✔	✘	GPU
cuSPARSE [46]	✔	✔	✔	✘	GPU
DBSR [72]	✔	✔	✘	✔	X86/ARM
Fast Sparse [63]	✘	✔	✔	✔	Sunway
This work	✔	✔	✔	✔	GPU

1 Introduction

Sparse iterative solvers play a vital role in a wide range of scientific and engineering applications [15, 52, 55, 74], particularly for solving large-scale systems arising from the discretization of partial differential equations (PDEs). In a sparse iterative solver, the incomplete LU (ILU) computation [45, 52] is frequently employed as the preconditioner [8, 26, 30, 65] or the subdomain solver [5, 24, 36, 68, 69] to accelerate the convergence speed; however, its complexity often throttles the performance of overall solving process.

The ILU computation consists of a factorization phase, which is usually performed once, and a sparse triangular solve (SpTRSV) phase, which is often repeatedly applied. Both phases introduce complex loop-carried data dependencies, posing significant challenges in identifying inherent parallelism and fully utilizing hardware resources. And the challenges are even more severe on modern GPUs with massive parallelization capabilities. In addition to that, a large variety of problems rely on solving PDEs defined on structured grids [23, 49, 71], where the grid points are arranged in a regular geometric pattern, and their topology is predetermined. This feature makes it possible for more efficient parallel execution [10, 73]; however, it remains unexplored how the geometric information of the structured grids can be effectively leveraged to accelerate ILU on GPUs.

There have been extensive efforts on ILU parallelization, which are summarized in Table 1. These approaches can be categorized into two classes based on whether they preserve the data dependency ordering. The first class exploits high parallelism through matrix reordering [33, 51] and inexact factorization [4, 12, 13], which trade off accuracy for computational efficiency. As a result, they may experience lower convergence efficiency due to the disruption of data dependencies [13, 16]. The second class completely maintains the original data dependencies and produces the same result as a serial implementation. The level-set method [17, 27, 50, 60], which is used by cuSPARSE [46, 47], constructs a dependency graph and divides it into levels, where tasks in the same level can be parallelized. This method may incur substantial synchronization overhead between levels. In contrast, synchronization-free methods [18, 37, 42] avoid global synchronization through point-to-point spin waits, but may suffer from insufficient parallelism.

For structured grid problems, the data dependency patterns become more explicit, and could help overcome the limitations of existing methods, thus achieving both high convergence and computational efficiency. Specifically, the geometric information enables static optimizations on task mapping and scheduling strategies to maintain dependencies. These strategies continue to leverage parallelism through the level-set method while allowing fine-grained synchronization via a stencil-based spin-wait scheme. Additionally, simplifying matrix indexing facilitates further optimizations in memory access. Several works are focused on structured grid ILU parallelization on SIMD CPUs [58, 72] and the Sunway processor [63]. However, it is hard for them to extend to GPU platforms. The massive, hierarchically organized threads of GPU present a challenge in aligning computations with the SIMT model. The data dependency may incur warp divergence and uncoalesced memory access, and even result in deadlocks due to cyclic dependencies between thread blocks. Furthermore, leveraging powerful Tensor Cores requires elaborate adaptations in both computations and memory layouts [10, 38, 40].

In this paper, we propose StructILU, an efficient structured grid ILU framework to leverage GPUs. StructILU can harness the fine-grained parallelism within ILU computations. Previous works on CPUs [58, 63, 72] focus on parallelizing grid point computations within each level of the level-set method, a scheme referred to as **grid parallelism**. A coarse-grained stencil computation task is assigned to each grid point, failing to occupy threads on GPUs that are tailored for fine-grained operations. To address this issue, we further investigate the **intra-row parallelism** of single grid point computations, which is achieved by parallelizing across multiple nonzero elements within a matrix row. For multiphysics problems, the **intra-element parallelism** can be extracted

from the dense block elements in the matrix, which is formed by the correlation of multiple variables.

The StructILU framework incorporates all three tiers of parallelism, collectively forming **hierarchical parallelism**. For grid parallelism, we introduce an *acyclic dependency task mapping* technique from a level-set perspective. This approach decomposes dependencies between levels while minimizing synchronization overhead. For intra-row parallelism, we develop *latency-aware pipeline method* to reduce computation latency within a single row. In factorization phase, we analyze the data dependency between nonzero elements, and develop a two-substage pipelining technique to improve load balancing. For intra-element parallelism, we leverage the *Tensor Cores* in modern GPUs to significantly enhance the efficiency of computationally intensive matrix operations. Furthermore, aligned with hierarchical parallelism, we design *memory transformations*, including a vector-oriented matrix layout and level-based transformations, to ensure coalesced memory access. These optimizations fully leverage the massive parallel computing power and high memory bandwidth of GPUs.

Overall, our paper makes the following contributions:

- We propose StructILU, an efficient GPU framework to parallelize dependency-preserving ILU for structured grid PDEs.
- We exploit the available parallelism in ILU by decoupling it into hierarchical tiers: grid, intra-row, and intra-element.
- We develop optimizations for hierarchical parallelism to enhance both computation and memory access efficiency.

The experiment results on NVIDIA A100 and RTX 3090 GPUs demonstrate that StructILU can achieve up to 6.5x and 17.7x (on average 2.11x and 3.88x) speedups in the triangular solve and factorization phases, respectively, compared to existing GPU implementations. Furthermore, evaluations of end-to-end ILU preconditioned iterative solvers on multiple real-world applications reveal that StructILU outperforms the best baselines, achieving an average overall speedup of 2.02x. Finally, a convergence test with different ILU preconditioner variations shows that StructILU achieves both the highest convergence rate and the shortest runtime.

2 Background

2.1 Incomplete LU on Structured Grids

The ILU decomposition, derived from LU decomposition, uses a sparse approximation of the original matrix A. It drops some elements in predetermined non-diagonal positions to obtain lower (L) and upper (U) triangular matrices such that $A \approx LU$ [52]. For symmetric positive definite (SPD) matrices, it simplifies to $A \approx LL^T$, known as incomplete Cholesky (IC). The ILU method is valuable as a preconditioner or subdomain solver for solving PDEs on a structured grid. The particular sparse structure of matrix A is dictated by the PDEs' discretization pattern, known as the **stencil**. Examples of various stencil patterns and their corresponding sparse matrices are illustrated in Figure 1.

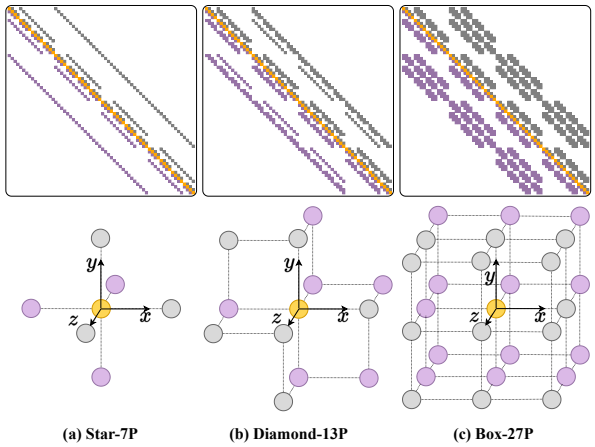


Figure 1: Three different stencil patterns and their corresponding sparse matrices. A central orange grid point, along with purple lower and gray upper neighboring grid points on its sides, represents a row of nonzero elements in matrix.

In detail, for a 3D structured grid with size $N_x \times N_y \times N_z$, the stencil is defined as $\mathcal{S} = \{S_i\}, S_i \in \mathbb{Z}^3$. In the matrix, the nonzero elements of a row (i, j, k) match columns $(i + x, j + y, k + z)$, where each (x, y, z) is a coordinate in $\mathcal{S}$. The total stencil size is noted by $s := |\mathcal{S}|$, with $\mathcal{S}_L$ and $\mathcal{S}_U$ denoting the lower and upper stencil subsets, s_L and s_U similarly. The term Degree of Freedom (DoF) refers to the number of variables at each grid point, and each nonzero matrix entry forms a $DoF \times DoF$ dense block.

The application of ILU consists of two fundamental stages. ❶ The **factorization phase** occurs only once during the setup stage of the iterative solver, which involves complicated arithmetic. Algorithm 1 shows the process of ILU factorization with zero fill-in (ILU(0)) on a structured grid, with matrices L and U stored in place. ❷ The **triangular solve** phase is repeatedly executed during each iteration of the iterative solver. Algorithm 2 illustrates the process for the lower triangular solve step on a structured grid. Note that both phases involve inherent loop-carried data dependencies, posing significant challenges for parallelization.

Algorithm 1: Inplace ILU(0) on a structured grid.

Input: Original matrix A with stencil $\mathcal{S}$.
Output: Factorized matrices L, U store in A, Diagonal inversion D_{inv}.

```
for row in grid_points:  # LOOP1: Grid
  for pt_l in S_L:       # LOOP2: Intra-row
    if (row + pt_l) in grid_points:
      A[row][S.index(pt_l)] *= D_inv[row + pt_l]
      for pt_u in S_U:   # LOOP3
        pt_elim = pt_l + pt_u
        if pt_elim in S:
          A[row][S.index(pt_elim)]   # Intra-element
            -= A[row][S.index(pt_l)]
              * A[row + pt_l][S.index(pt_u)]
  D_inv[row] = inv(A[row][S.index((0,0,0))])
```

Algorithm 2: Lower triangular solve.

Input: Lower triangular matrix L with identity diagonal, RHS vector b.
Output: Solution vector x.

```
for row in grid_points:    # LOOP1: Grid
  result = 0
  for pt in S_L:           # LOOP2: Intra-row
    col = row + pt
    if col in grid_points:
      result +=
        L[row][S_L.index(pt)] * x[col]  # Intra-element
  x[row] = b[row] - result
```

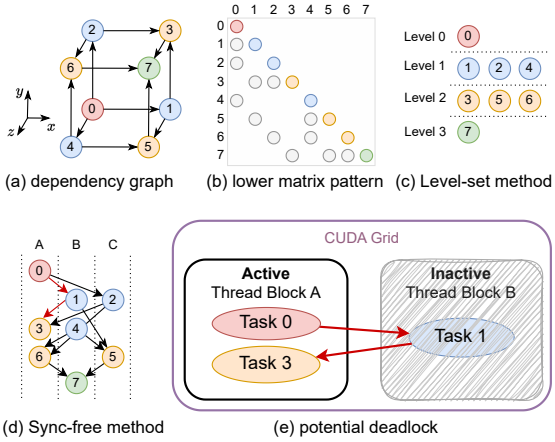


Figure 2: (a) Dependency graph for the lower triangular solve of Star-7P stencil. (b) Sparse pattern of the lower matrix L. (c) The classical level-set method. (d) The sync-free method. (e) Potential deadlock can be caused by cyclic dependency between thread blocks if the number of thread blocks exceeds the maximum active block count: Thread block A cannot exit until B starts, while B lacks the resources to start until A exits, resulting in a deadlock.

2.2 Level-set and Sync-free Methods

Despite the intrinsic loop-carried data dependencies and the massive parallelism of GPUs, several dependency-preserving methods have been developed for ILU and SpTRSV. We discuss the implementation of different existing methods using the Star-7P stencil as an example in Figure 2.

Level-set Method. The ILU computation is inherently sequential, yet the sparse nature of the matrix allows for a certain degree of parallelism in LOOP1 of Algorithms 1 and 2. To manage data dependencies effectively, the level-set method [2, 53] has been employed. This method categorizes the solution process into distinct levels, with each level consisting of tasks that can be executed independently. Specifically, dependencies among grid points are depicted using a directed acyclic graph (DAG). By analyzing the longest path from any vertex with zero incoming edges and assigning levels accordingly, nodes at the same level can be executed concurrently. As the classical level-set method processes each level sequentially using separate GPU kernels [33], high global synchronization overheads are introduced.

Sync-free Method. Targeting triangular solve, synchronization-free (sync-free) methods on GPUs have been developed to mitigate synchronization overhead. These methods employ a flag array to monitor the completion status of each row, enabling warps or threads to actively check the flag array while awaiting the completion of dependent rows, thereby eliminating the need for global synchronization. Specifically, the thread-level sync-free methods [56] allocate different rows to individual threads, while the warp-level approach [19, 37] assigns the computation of each row (LOOP2 in Algorithm 2) to the collaborative effort of threads within a single warp [25, 75].

3 Motivation and Overview

Modern GPUs are characterized by their massive parallelism, featuring thousands of core units designed for various types of computations. For example, the NVIDIA A100 GPU provides 10368 CUDA Cores for floating point and integer operations and 432 Tensor Cores for accelerating matrix operations. To fully harness the powerful computational capabilities of GPUs to parallelize Algorithms 1 and 2, we introduce the **hierarchical parallelism** in StructILU, which encompasses three distinct tiers: grid, intra-row, and intra-element. In addition, in terms of memory, to align with computations and ensure efficient coalesced access, we also optimize the memory layout through transformation.

Grid Parallelism. Using the stencil pattern associated with a structured sparse matrix, level information can be analytically determined without traversing the matrix's nonzeros. Therefore, grid parallelism from LOOP1 in Algorithms 1

and 2 can be maximized by applying the level-set method, whose global synchronization needs further optimization. For example, the level-set-based implementation in cuSPARSE [46, 47] launches separate kernels for different levels. Conversely, Sync-free methods break down synchronization into point-to-point interactions, but their parallelism is limited by task mapping. Figure 2(d) demonstrates a scenario where sync-free methods fail to maximize parallelism. Moreover, deadlocks may occur when bidirectional dependencies exist between thread blocks, as shown in Figure 2(e). These observations highlight a trade-off between achieving higher grid parallelism and mitigating the synchronization costs. Therefore, we designed an *acyclic dependency task mapping* method using the level-set philosophy, discussed in Section 4.1, while adopting a *block-to-block waiting scheme* inspired by the Sync-free method [37] to minimize synchronization costs between levels.

Intra-Row Parallelism. During the triangular solve phase, each row requires a dot product operation (Line 6 in Algorithm 2). The classical level-set methods [46] and thread-level sync-free approaches [56] neglect this form of parallelism. In contrast, warp-level methods [75] assign all threads within a warp to work collaboratively on a single row using warp reduction. However, this strategy often results in low warp utilization, as the number of nonzeros per row is relatively small for most stencils (e.g., Diamond-13P has $s_L = 6$, which uses only 18.8% of the 32-thread warp size) [25]. By noting that some of the dependent elements are available before the previous grid level completes, we introduced *latency-aware pipeline dot product* in Section 4.2.1.

Intra-row parallelism for ILU factorization is more intricate than that in SpTRSV, as multiple updates to the current row's nonzero elements (Line 4, 8 in Algorithm 1) must adhere to the specific order dictated by `LOOP2`. This results in a computational complexity of $O(s_L^2)$, which significantly surpasses that of SpTRSV. For instance, a Box-27P stencil row requires 114 updates and one diagonal inversion. For stencil-based problems, dependency analysis can be performed with negligible cost during the initial setup phase, enabling our proposal of *Intra-row level-set parallelization.* Additionally, a *two-substage pipeline* is also introduced in Section 4.2.2, which improves load balancing and reduces block synchronization overhead during the factorization of each row.

Intra-Element Parallelism. Finally, in multi-physics problems, each nonzero element in the sparse matrix becomes a $DoF \times DoF$ dense block, necessitating further intra-element parallelism. During the triangular solve phase, this parallelism can be fully utilized by assigning DoF threads to a single grid point. For ILU factorization, dense block computations involve matrix multiplication and matrix inversion operations, both with a complexity of $O(DoF^3)$. Although the block sparse row (BSR) interface in cuSPARSE and other studies [28, 43, 44] support these computations, the accelerated Tensor Cores on modern GPUs remain underutilized. To fully leverage these hardware units, we introduce *Tensor Core utilized matrix operations* to maximize intra-element parallelism; details are provided in Section 4.3.

Memory Transformation. In addition to maximizing parallelism across three hierarchical levels, we also adapt the memory layout to align with computations and ensure coalesced memory access, particularly during the memory-intensive SpTRSV phase. Existing methods for general SpTRSV fail in achieving coalesced access for stencil-based problems. For example, in the classic level-set method, threads process grid points of the same level concurrently. However, in a non-degraded 3D stencil, successive grid points along the x-axis never belong to the same level. Similarly, in warp-level sync-free methods, threads within a warp access solution vector entries correspond to non-successive columns (`x[col]` in Algorithm 2), resulting in uncoalesced access patterns. However, the fixed pattern of the stencil enables the chance to design specific memory layouts. Therefore, delicately matching the computation pattern of hierarchical parallelism, we design a *vector-oriented matrix layout* combined with *level-based transformation* to ensure coalesced access for both matrix and vector data, as detailed in Section 4.4.

4 Detailed Design of StructILU

In this section, we introduce the detailed design to utilized grid, intra-row, and intra-element parallelism in Section 4.1, 4.2, 4.3 respectively, and memory transformations in 4.4.

4.1 Grid: Acyclic Dependency Task Mapping

As discussed in Section 3, existing methods involve a trade-off between achievable grid parallelism and synchronization overhead. Considering both factors, we propose an *acyclic dependency task mapping* method. This method divides tasks into slice hyperplanes based on geometric information, adopting a level-set perspective to ensure maximum parallelism. Building upon the principles of sync-free methods, we then propose a *block-to-block waiting scheme* that allows points at subsequent levels to enter iterations rapidly, while maintaining dependencies between levels. This task mapping is inherently deadlock-free, eliminating the limit on total thread block count and thereby enhancing hardware utilization. Furthermore, by preserving dependencies across multiple levels within a single thread block, our approach also enhances data reuse. In the subsequent, we delve into the specifics of this approach, beginning with a simple Star-7P stencil and then moving to a general case, as illustrated in Figure 3.

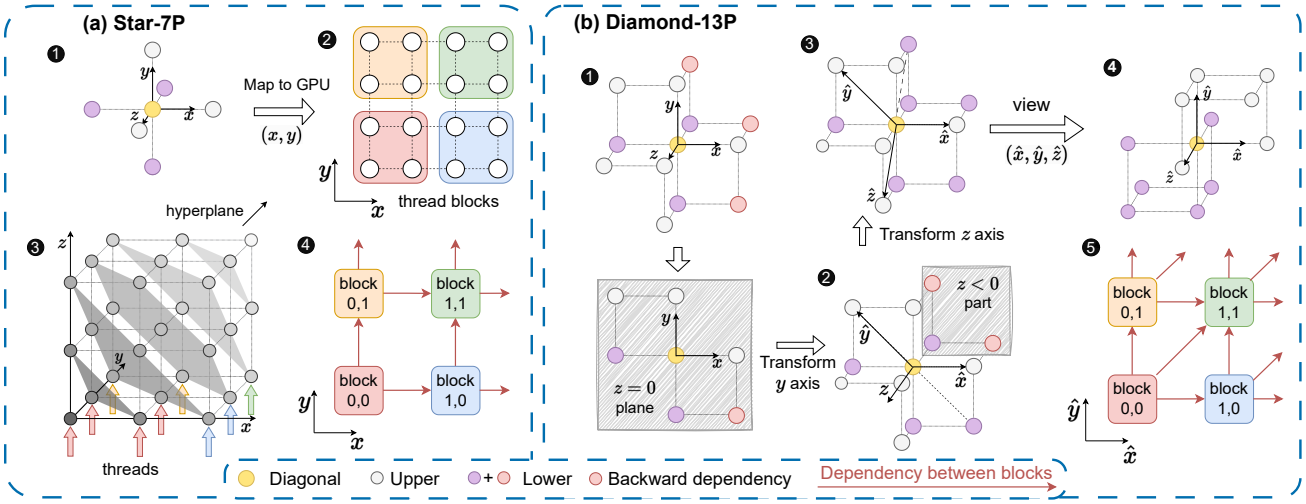


Figure 3: Acyclic dependency task mapping on GPUs. (a) Star-7P stencil. (b) General stencil cases, taking Diamond-13P stencil as an example.

Star-7P stencil. ❶ Data dependencies exist solely in the positive directions of the x, y, and z axes. The level of a grid point (i, j, k) is $i+j+k$, resulting in a total of $N_x+N_y+N_z-2$ levels. This arrangement allows points on the same level to be processed concurrently. ❷ We partition the grid in the xy-plane, assigning each point on this plane to a thread. These threads are then organized into thread blocks that are orthogonal to the coordinate axes. ❸ Each thread performs computations along the z-axis, ranging from $z = 0$ to $N_z - 1$. Threads within a block execute tasks synchronously at the same level, forming a wavefront hyperplane.

A *block-to-block waiting scheme* is proposed to preserve dependencies between thread blocks. This scheme involves a flag array in device memory, where each entry tracks the number of levels completed by the corresponding thread block. Before starting level l, a thread block $(\text{bid}_x, \text{bid}_y)$ spin-waits until the blocks below it $(\text{bid}_x, \text{bid}_y - 1)$ and to its left $(\text{bid}_x - 1, \text{bid}_y)$ complete level $l - 1$. Upon completing level l and performing a block synchronization (block-sync) with a memory fence, the flag for the current block is incremented. Figure 3(a)❹ shows that direct data dependencies between thread blocks occur only in the positive x and y directions. This design handles dependencies without global synchronization and effectively avoids deadlocks.

General stencil cases. The major difference between star-shaped stencils and other cases lies in the existence of nonzero lower points in $\{(x, y, z) \in \mathcal{S}_L | x > 0 \wedge y < 0 \wedge z = 0\}$ and $\{(x, y, z) \in \mathcal{S}_L | (x > 0 \vee y > 0) \wedge z < 0\}$, which cause backward dependencies. For Diamond-13P stencil, these points are colored red in Figure 3(b)❶. Here, points $(1, -1, 0)$, $(1, 0, -1)$, and $(0, 1, -1)$ can potentially cause backward dependencies. If we directly map the xy-plane to the thread grid, thread block (bid_x, bid_y) depends on $(bid_x + 1, bid_y)$ and $(bid_x, bid_y + 1)$ for computation at the next level. This results in a cyclic dependency, which can cause potential deadlocks when the number of thread blocks exceeds the maximum active block count of the GPU. A global synchronization would be required to prevent this, as illustrated in Figure 2(e).

To resolve this issue, we perform an affine transformation on the coordinate system. Let the original basis vector be $\boldsymbol{e}_x, \boldsymbol{e}_y, \boldsymbol{e}_z$. We first define $\boldsymbol{e}_{\hat{x}} := \boldsymbol{e}_x$ as the $\hat{x}$ direction. Then, we check the slice plane $z = 0$ of $\mathcal{S}_L$ and use the upper-rightmost red point in the lower half plane to define the $\hat{y}$ direction as:

$$-\boldsymbol{e}_{\hat{y}} \in \underset{(x,y,0)\in\mathcal{S}_L, x>0, y<0}{\text{argmax}} \frac{|x|}{|y|}. \tag{1}$$

❷ After applying the affine transformation base on $(\boldsymbol{e}_{\hat{x}}, \boldsymbol{e}_{\hat{y}})$, we obtain the $\hat{x}\hat{y}$-coordinate. Thus all lower points in slice plane $z = 0$ satisfy $\hat{x} \leq 0$ and $\hat{y} \leq 0$. Next, we consider points where $z < 0$, and derive $\boldsymbol{e}_{\hat{z}}$ as follows:

$$-\boldsymbol{e}_{\hat{z}} \in \underset{(x,y,z)\in\mathcal{S}_L, (x>0\vee y>0)\wedge z<0}{\text{argmax}} \frac{|\hat{x}| + |\hat{y}|}{|z|}. \tag{2}$$

Take Diamond-13P stencil as an example, the lower point $(1, -1, 0)$ satisfies the condition in (1) leading to $\boldsymbol{e}_{\hat{y}} := \boldsymbol{e}_y - \boldsymbol{e}_x$. Then, the lower point $(0, 1, -1)$ has $\hat{x} = 1, \hat{y} = 1, z = 1$, which satisfies the maximum condition in (2). This leads to the transformation $\boldsymbol{e}_{\hat{z}} := \boldsymbol{e}_z - \boldsymbol{e}_y$. Actually, the choice of $\boldsymbol{e}_{\hat{y}}$ and $\boldsymbol{e}_{\hat{z}}$ may not be unique. ❸ By applying affine transformation based on $\boldsymbol{e}_{\hat{x}}, \boldsymbol{e}_{\hat{y}}, \boldsymbol{e}_{\hat{z}}$, we transform the xyz-coordinate to $\hat{x}\hat{y}\hat{z}$-coordinate. This results in the set $\{(x, y, z) \in \mathcal{S}_L | (\hat{x} > 0 \vee \hat{y} >$

$0) \wedge \hat{z} < 0\} = \emptyset$, which implies that no points are causing backward dependencies. ❹ A view of the stencil in the $\hat{x}\hat{y}\hat{z}$ coordinate system is shown.

For the allocation of tasks to threads, we employ an approach similar to the one utilized in the Star-7P case, with the new coordinates $(\hat{x}, \hat{y}, \hat{z})$. Specifically, we map the $\hat{x}\hat{y}$-plane onto 2D thread blocks for the Diamond-13P stencil, assigning each thread a line of points along the $\hat{z}$ direction. Here, a grid point (i, j, k) in xyz-coordinate corresponds to $(\hat{i}, \hat{j}, \hat{k}) = (i+j+k, j+k, k)$ in $\hat{x}\hat{y}\hat{z}$-coordinate. Thus the level under this transformation is given by $\hat{i} + \hat{j} + \hat{k} = i + 2j + 3k$. Figure 3(b)❺ illustrates that there are no cyclic dependencies between thread blocks. Each thread block (bid_x, bid_y) waits for the completion of the block $(bid_x - 1, bid_y - 1)$ due to the diagonal dependencies. Finally, we sort the thread blocks by their minimum effective level for GPU kernel launch to further improve scheduling efficiency.

Similarly, we can obtain $\boldsymbol{e}_{\hat{x}} := \boldsymbol{e}_x, \boldsymbol{e}_{\hat{y}} := \boldsymbol{e}_y - \boldsymbol{e}_x, \boldsymbol{e}_{\hat{z}} := \boldsymbol{e}_z - \boldsymbol{e}_y - \boldsymbol{e}_x$ for the Box-27P stencil. The grid point (i, j, k) in xyz-coordinate has a new coordinate of $(\hat{i}, \hat{j}, \hat{k}) = (i + j + 2k, j + k, k)$ in $\hat{x}\hat{y}\hat{z}$-coordinate, with level $\hat{i}+\hat{j}+\hat{k} = i+2j+4k$. Let us denote N_{levels} as the number of levels and $P_{grid} := \frac{N_x N_y N_z}{N_{levels}}$ as the average grid parallelism. Then, N_{levels} for the Star-7P case and Box-27P case are given by $N_x + N_y + N_z - 2$ and $N_x + 2N_y + 4N_z - 6$, respectively. For N^3 grid case, the P_{grid} is about $\frac{N^2}{3}$ and $\frac{N^2}{7}$, implying that the grid parallelism for the Box-27P case is only 43% of that for the Star-7P case, necessitating further fine-grain parallelization.

4.2 Intra-Row: Latency-Aware Pipeline Methods

For cases with many levels, grid parallelism becomes insufficient, making it crucial to fully exploit fine-grained intra-row parallelism to reduce latency at each level. Parallel computation of a single row reduces the number of floating point operations N_{fp} along the critical path, although it may introduce additional synchronization overhead due to N_{bs} block-syncs. We propose latency-aware pipeline parallelization strategies for both the triangular solve and factorization phases, to parallelize intra-row computations, corresponding to `LOOP2` in Algorithms 1 and 2. The *grid level* definition adheres to that in Section 4.1, and the pipeline methods are demonstrated using a Diamond-13P stencil in Figure 4. Nonzeros are sorted by their *intra-row levels*, where each *intra-row level* for a nonzero in $\mathcal{S}_L$ is calculated as the sum of its transformed $\hat{x}\hat{y}\hat{z}$-coordinates. For example, shown in (a), the lower points in Diamond 13P have intra-row levels ranging from -3 to -1. This paper introduces pipeline optimizations for the lower triangular solve phase in (b), and the ILU factorization in segments (c) through (f).

4.2.1 Triangular solve phase. ***Intra-row pipelined dot product.*** For the lower triangular solve phase, the task of each row, i.e. `LOOP2` in Algorithm 2 is an inner product with length s_L. To parallelize it, the nonzeros are partitioned into P_{row} stages. The choice of P_{row} is not unique, but the maximum reasonable value to ensure load balance is determined by the following rule: The level of nonzeros in each stage does not exceed $stage\_id - P_{row}$, where $0 \le stage\_id < P_{row}$. Afterward, these stages are mapped to P_{row} groups within a thread block and processed in parallel using a pipeline manner. Figure 4(b) shows the pipeline for the Diamond-13P stencil with $P_{row} = 2$. For a grid point with grid level l, the summation corresponds to three nonzero elements with intra-row level -1 is performed at the grid level l; However, the other three elements with intra-row level -2 and -3 are summarized earlier during the execution of grid level $l - 1$. Assuming sufficient free resources in SM, the N_{fp} is reduced from 6 to 3. It is worth noting that compared to warp-level methods, whose warp reduction gives an overhead of an addition, N_{fp} is reduced by 25%. Specifically, the dependencies between pipeline stages are maintained by the *block-to-block waiting scheme* described in Section 4.1, with no additional synchronization cost, i.e. $N_{bs} = 0$. For Star-7P whose grid parallelism is much greater, this pipeline method is not applicable due to its compact dependencies that row with level l depends only on rows with level $l - 1$. For other stencils, the effect of pipelining is summarized in Table 2.

4.2.2 ILU factorization phase. ***Intra-row level-set parallelization.*** Since the computational cost of the factorization is typically higher than that of the triangular solve phase, it is necessary to utilize finer-grained parallelism when eliminating each matrix row. As stated in Algorithm 1, during the elimination, each lower triangular nonzero (`pt_l` in `LOOP2`) performs several updates to subsequent nonzeros (`pt_elim` in `LOOP3`). Simply parallelizing `LOOP3` is sub-optimal for complex stencils, as the latency is strictly limited by s_L of `LOOP2`. To address this issue, we construct an *intra-row dependency graph* for the elements within rows based on dependencies, as shown in Figure 4(c). The dependency graph can then be executed using level-set parallelization with P_{row} thread groups. The P_{row} can be chosen according to the load balancing result. Unlike the original serial algorithm, the updates of each nonzero `pt_elim` are merged, at the node corresponding to `pt_elim` in the graph. The computation cost of each nonzero includes multiple updates from previously completed elements (line 8 in Algorithm 1), coupled with the multiply for the lower part (line 4) and the inversion of diagonal (line 11).

Figure 4(d) illustrates the level-set parallelization of Diamond-13P using $P_{row} = 4$, the length of bricks showing the computational cost of each nonzero. Assuming that the

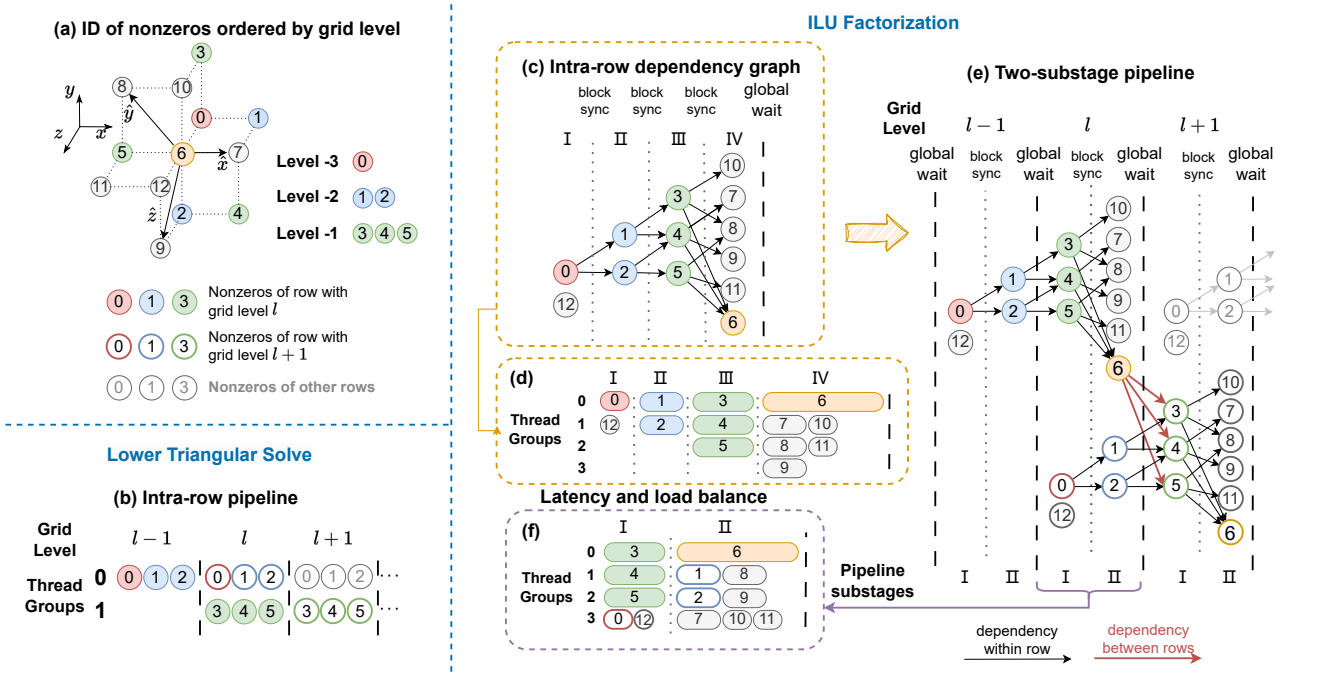


Figure 4: Latency aware pipeline parallelization. (a) Diamond-13P Stencil with points ordered by grid level. (b) Intra-row pipeline for the lower triangular solve. (c) Intra-row dependency graph for factorization, with dependencies across multiple levels hidden for simplicity. (d) Latency and load balance of Level-set parallelization. (e) Two-substage pipeline. (f) Latency and load balance of Two-substage pipeline.

cost of fused multiply-add (FMA) or inverse is 1, the latency per row involves a computational cost $N_{fp} = 13$, along with 3 block-syncs between levels. This is a notable reduction compared to the straightforward parallelization of LOOP3, which incurs 14 computational costs and 12 block-syncs. However, it can be observed in (d) that workload imbalance between thread groups still prevents full utilization of parallelism.

Two-substage pipeline method. Although intra-row elimination has been parallelized, the multiple block synchronizations and exacerbated load imbalance within levels still result in high overhead. Additionally, the red arrows in Figure 4(e) denote dependencies between nonzero elements in rows with adjacent grid levels (across LOOP1), which prohibit direct pipelining of intra-row stages. Therefore, we adopt a two-substage pipeline to allow the overlapping of rows in successive grid levels. After the first two intra-row levels of a row at grid level l are completed, the computation for the row at grid level $l + 1$ begins. This approach ensures that only one additional block-sync between two substages is involved, and other dependencies are implicitly handled by the *block-to-block waiting scheme* between grid levels. Also, (f) shows the load balancing improvement due to the more elements in each substage (from rows of successive grid levels) to be parallelized compared to the original intra-row level set parallelization. For Diamond-13P stencil, the floating point cost N_{fp} is reduced to 10, and N_{bs} is reduced to 1.

Table 2: Computation latency with pipeline parallelization on a single row, under typical choices of P_{row}.

Stencil	Triangular Solve			Factorization		
	Serial N_{fp}	P_{row}	Parallel N_{fp}	Serial N_{fp}	P_{row}	Parallel N_{fp}
Star-13P	6	2	3	19	2	11
Diamond-13P	6	2	3	29	4	10
Diamond-25P	12	4	3	95	4	25
Box-27P	13	4	4	115	4	30

As observed in Table 2, the pipeline method reduces latency for both triangular solve and factorization phases. Moreover, with compact updates of nonzero elements, segmenting multiple pipeline stages within a thread block also improves the spatial locality of data.

4.3 Intra-Element: Tensor Core Utilization

For the case with $DoF > 1$, the introduction of $DoF \times DoF$ dense matrix operations significantly increases the arithmetic intensity of the ILU factorization. It is essential to fully exploit the acceleration units provided by the hardware, such

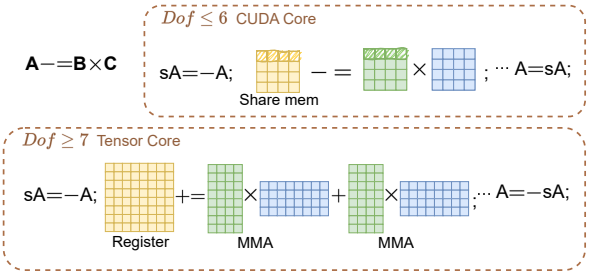


Figure 5: Tensor Core utilized dense matrix operations for multi-*DoF* cases.

as Tensor Cores in the A100 GPU, to maximize intra-element parallelism. For $A = A - B \times C$, shown in Figure 5, when the DoF is sufficiently large, we utilize the FP64 Tensor Cores to perform matrix multiply-add (MMA) operations. In this case, the warp directly corresponds to the parallelism across the entire DoF dimension. During elimination, the original matrix is negated and stored in registers, and after completing all MMA operations, the result is negated again and written back to device memory. In this case the intra-element parallelism $P_{DoF} = 32$. For $A = A^{-1}$, we inverse the matrix using Gauss-Jordan elimination, where threads in a warp collaborate using warp shuffle.

In other cases where Tensor Cores cannot be utilized, each thread is responsible for the computation of one row during matrix multiplication and inversion, storing intermediate results in shared memory, and $P_{DoF} = DoF$. For the triangular solve with multiple DoFs, we assign successive DoF threads to each variable in the solution vector, and no warp divergence is incurred since the computation is independent of other variables within the current grid point.

4.4 High Throughput Memory Transformation

With the exploration of hierarchical parallelism to fully exploit the massive parallelism of GPUs, adapting the memory layout becomes essential to achieve high bandwidth utilization. We apply transformations to the matrix and vector storage formats to align with the computational patterns, thereby ensuring coalesced memory access.

Vector-oriented matrix layout. Sparse matrices from structured grid PDEs are shaped by the underlying stencil, enabling an optimized storage scheme through zero-padding and avoidance of extra indices. In StructILU, we represent the sparse matrix as a 6D tensor with dimensions $DoF \times DoF \times s \times M \times N \times P$, where the physical arrangement of tensor dimensions critically influences memory-access efficiency. Figure 6(a) illustrates different storage formats using the lower matrix from Star-7P as an example. Common formats,

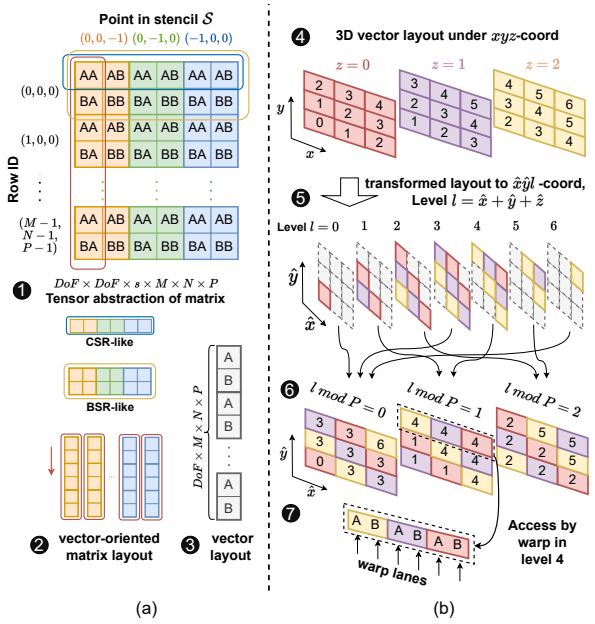


Figure 6: (a) The tensor abstraction and vector-oriented matrix layout. (b) The level-based memory layout transformation for coalesced access. A and B denote 2 *DoF*s.

such as CSR and BSR, store nonzero elements in row-major order, failing to achieve coalesced memory access under our task mapping. To address this, we propose a *vector-oriented matrix layout*, which splits each block column-wise. Each column of length DoF is stored contiguously along the grid axes, resulting in multiple $DoF \times M \times N \times P$ arrays that align with vector-based storage.

Level-based memory layout transformation. In our task mapping, threads corresponding to the same level are successive along the grid's x axis. However, they do not access adjacent locations. Taking Star-7P as an example, when computing level l, adjacent threads (i, j) and $(i+1, j)$ in the thread grid process $(i, j, l-i-j)$ and $(i+1, j, l-i-j-1)$, respectively, leading to non-coalesced access due to different z coordinates. To resolve this, we employ an affine transformation, similar to subsection 4.1, but elevate $\hat{z}$ to $l = \hat{x} + \hat{y} + \hat{z}$. Figure 6(b) shows this on a $3 \times 3 \times 3$ grid. Under this scheme, the vector data for (i, j, k) is stored at $(\hat{i}, \hat{j}, l)$, ensuring adjacent threads access contiguous data and achieving coalesced memory access. For matrices, we apply a corresponding offset transformation to the proposed vector-oriented layout, also ensuring coalesced access. Furthermore, any storage overflow from increased level numbers can be mitigated by applying a modulus to each dimension, preventing excessive memory usage.

5 Evaluation

5.1 Setup

We implement StructILU with CUDA C++. StructILU is compiled with Clang 18.0 and CUDA 12.2. Our experiments run on two machines. The first one has an Intel Xeon Gold 6326 CPU and an RTX 3090 GPU. The second features a same CPU along with one A100 PCIe GPU [11]. An RTX 3090 GPU has 24GB device memory, offering a theoretical memory bandwidth of 936 GB/s and an FP64 performance of 556 Gflop/s, with a boost clock of 1950MHz. Conversely, an A100 GPU has 80GB memory, providing a theoretical memory bandwidth of 1935 GB/s, with a 1410MHz GPU boost clock. The FP64 CUDA Core performance for the A100 GPU is 9.7 Tflop/s, and 19.5 Tflop/s when utilizing the FP64 Tensor Core units. All tests use the double precision.

5.2 Single Kernel Performance

To our best knowledge, no existing GPU implementation is specifically tailored for dependent-preserving structured grid ILU. Consequently, we compare our StructILU approach with several baselines designed for general sparse matrices, including the latest cuSPARSE 12.1, AG-SpTRSV [25], and Recursive Block (Rec-Block) [42]. Among these, only cuSPARSE provides implementations for ILU factorization. We evaluate both CSR and BSR formats in cuSPARSE, enabling level information to ensure enhanced kernel performance. For AG-SpTRSV, an exhaustive search is conducted to optimize performance outcomes, while Rec-Block is implemented using its default blocking parameters. AG-SpTRSV and Rec-Block are variants of Sync-free methods. We evaluate the single kernel performance of the lower triangular solve and ILU factorization stages using five different stencil patterns under various DoF settings, as in Table 3.

The results are shown in Figure 7. For the lower triangular solve phase, StructILU achieves an average speedup of 2.16x on the RTX 3090 and 2.06x on the A100 GPUs compared to the fastest baseline. It is evident that StructILU delivers consistently high performance across various stencils and DoFs, ranging from 99 Gflop/s to 296 Gflop/s. In contrast, the baselines face a dramatic performance degradation for small DoF and s, possibly due to the few nonzeros in a row. Therefore, StructILU achieves an average speedup of 4.1x for $DoF = 1$. Notably, StructILU achieves an average bandwidth of 700.4 GB/s on the RTX 3090, which is 74.8% of the theoretical peak. This indicates that StructILU effectively utilizes parallelism but is limited by hardware memory bandwidth.

For the ILU factorization phase, StructILU achieves average speedups of 4.22x on the RTX 3090 and 3.56x on the A100 GPUs compared to the best baseline. The trend in speedup is similar to that observed in the triangular solve phase. With the Diamond-25P stencil using $DoF = 8$, StructILU sustains a performance of 385 Gflop/s on the RTX 3090, 69.2% peak performance. This indicates that the achievable performance of the RTX 3090 is limited by its peak FP64 performance under such a complicated case, in which case the A100 GPU achieves 959 Gflop/s, thanks to FP64 Tensor Core units.

Stencil	DoF=1	DoF=2	DoF=4	DoF=6	DoF=8
Star-7P	512 | 560	336 | 416	208 | 256	160 | 192	128 | 160
Star-13P	432 | 544	272 | 336	160 | 208	128 | 160	96 | 128
Diamond-13P	432 | 544	272 | 336	160 | 208	128 | 160	96 | 128
Diamond-25P	336 | 416	208 | 256	128 | 160	96 | 128	80 | 104
Box-27P	336 | 416	208 | 256	128 | 160	96 | 128	80 | 104

Table 3: The configuration of single kernel tests. To make full use of the device memory, we choose appropriate N^3 grids for RTX 3090 | A100 GPU.

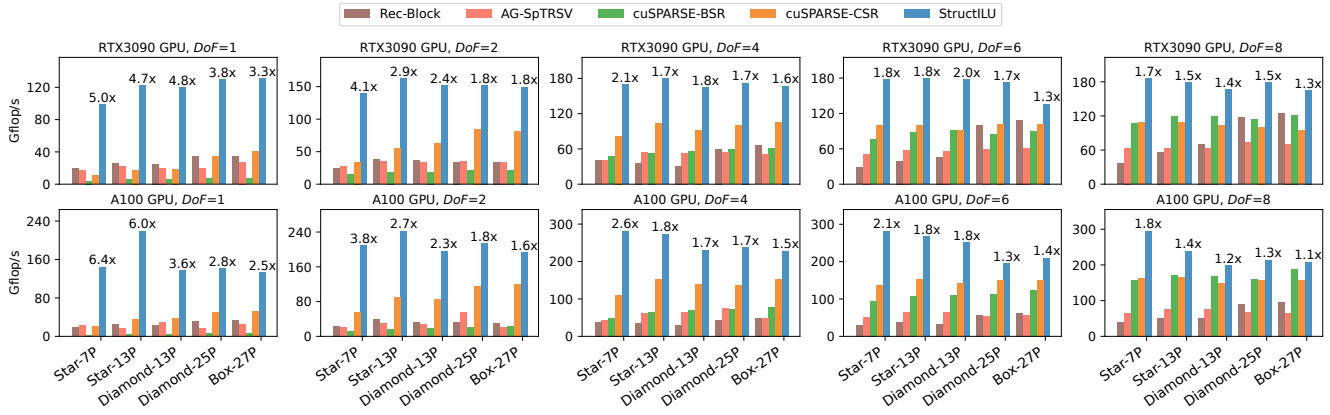
5.3 Performance Breakdown

In this subsection, we explore how StructILU benefits from the proposed techniques. Figure 8 details the results on an A100 GPU. We use acyclic dependency task mapping as the baseline. By applying level-based memory layout transformation, the performance of the four cases improves by 2.8x, 1.7x, 3.8x, and 2.2x, respectively. Next, we implement pipeline optimization for intra-row dot products. The speedups achieved are 1.1x, 1.8x, 1.8x and 2.9x, respectively. This technique offers slight acceleration for the Star-13P case as its grid parallelism is nearly sufficient to utilize GPU resources.

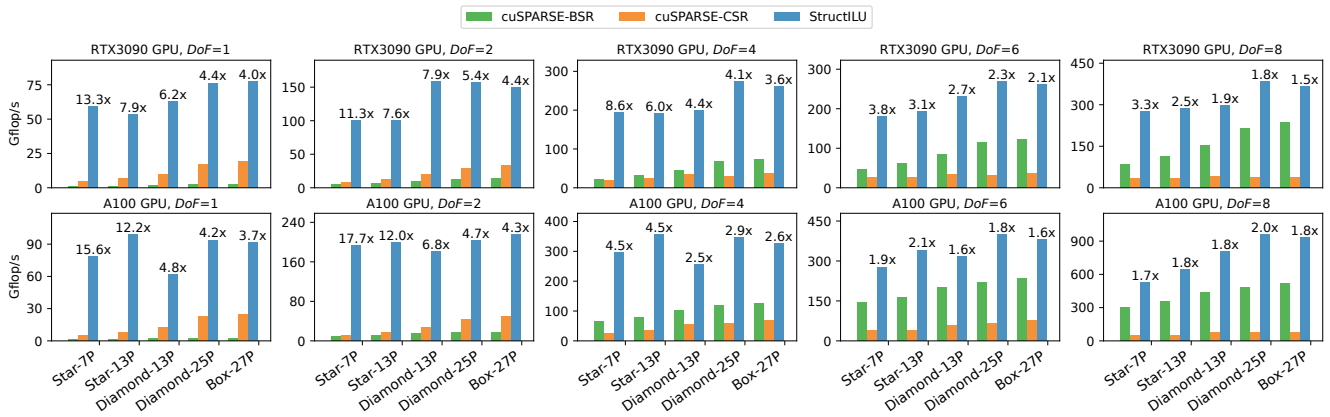
For the ILU decomposition, using task mapping as the baseline, memory transformation provides speedups of 5.1x, 3.4x, 1.6x and 1.5x for four cases. By applying the two-substage pipelining method, performance improves by 16%, 26%, 31%, and 47%, respectively, to exploit the intra-row parallelism and reduce latency. This performance gain is less than desired P_{row} due to the limitation of SM resources. Finally, by enabling Tensor Core when $DoF = 8$, we observe performance improvements of 81% and 82% for Diamond-25P-8 and Box-27P-8, respectively, demonstrating a performance gain matching the hardware feature.

5.4 Parallelism Under Different Configuration

We also scale problem sizes across different GPUs to observe performance changes. Figure 9 compares the lower triangular solve performance for the Star-13P and Diamond-13P stencils. In both cases and on both devices, StructILU achieves a significant performance gain over cuSPARSE, especially for large N. Additionally, by leveraging the geometric structure, we eliminate the need for index arrays and preprocessing storage, enabling StructILU to handle larger problems on memory-limited GPUs.



(a) Lower triangular solve



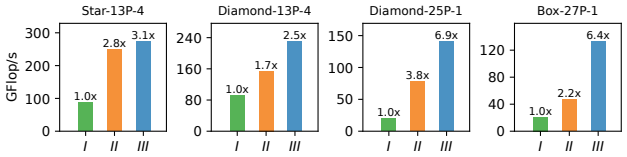
(b) Factorization

Figure 7: Performance comparison of StructILU and baselines on the lower triangular solve and ILU factorization phase, excluding the time for preprocessing and symbolic factorization. The speedup of StructILU over the fastest baseline is marked.

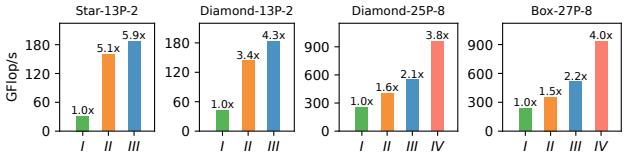
Moreover, parallelism analysis can be performed by comparing the two stencils. Both Star-13P and Diamond-13P stencils use a two-stage pipeline with similar memory and computation footprints. However, Star-13P has fewer levels, $N_{levels} = 3N - 2$, compared to Diamond-13P, which has $N_{levels} = 6N-5$, nearly doubling the number of levels. On the A100, for $N \leq 192$, the performance of Star-13P is roughly twice that of Diamond-13P, indicating that performance is primarily limited by grid parallelism $P_{grid} = \frac{N^3}{N_{levels}}$. The average overall parallelism $P_{grid}P_{row}P_{DoF} \approx \frac{2N^2}{3}$ of Diamond-13P is half that of Star-13P, which is $\frac{4N^2}{3}$. Therefore, Diamond-13P, with a grid size of $\sqrt{2}N$, achieves similar performance to Star-13P with an N grid due to comparable parallelism. Additionally, it is noteworthy that RTX 3090 outperforms A100 for very small N due to insufficient parallelism, and its performance is linearly related to the GPU core frequency. As N increases, A100 performs better owing to its superior hardware parallelism and higher memory bandwidth.

5.5 ILU Preconditioned Iterative Solver Test

Furthermore, six real-world applications are selected to compare the performance of different ILU implementations as preconditioners for iterative solvers. Table 4 and Figure 10 present the configuration and time breakdown results for each problem, respectively. Ensuring numerically consistent solutions, StructILU achieves an average speedup of 2.35x and 1.73x on RTX 3090 and A100 GPUs, respectively. For example, in the Cahn-Hilliard problem with 22 iterations on RTX 3090, cuSPARSE-CSR spends 25.6% of the time on the factorization phase, while StructILU only spends 9.8%,



(a) Lower triangular solve



(b) Factorization

Figure 8: Performance breakdown on A100 GPU. Star-13P-2 represents Star-13P with $DoF = 2$. *I*: Task mapping; *II*: +Memory layout optimizations; *III*: +Pipeline parallel; *IV*: +Tensor core for large DoF.

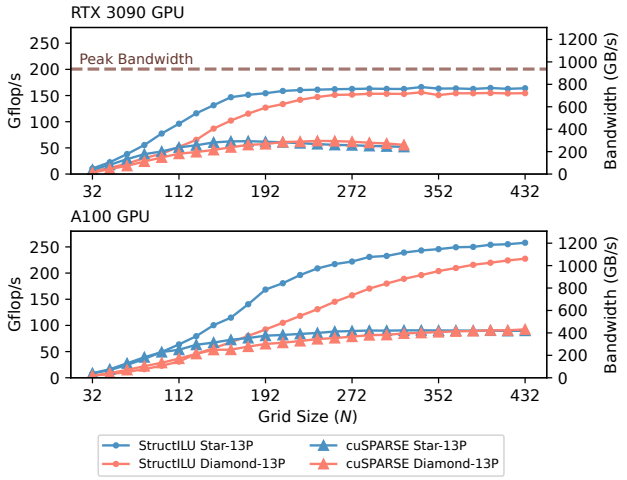


Figure 9: Performance of the lower triangular solve with two stencils with $DoF = 2$, using StructILU and cuSPARSE-CSR on A100 and RTX 3090 GPUs. The grid size is N^3.

indicating a significant reduction in ILU factorization cost. Additionally, StructILU outperforms cuSPARSE in matrix-vector multiplication due to its vector-oriented matrix layout, which ensures coalesced accesses and eliminates indexing overhead.

Specifically, due to the typically high cost of the analysis and symbolic factorization phases in ILU, StructILU directly leverages geometric information within the kernel, eliminating this overhead. Therefore, it is excluded from our comparison. For example, in the Poisson-IC(0) case on RTX 3090, cuSPARSE's preprocessing takes 1.16 seconds, while the linear solve time is only 1.76 seconds. In contrast, the memory layout transformation in StructILU takes just 0.002 seconds, negligible compared to the total linear solve time of 0.53 seconds.

Table 4: Problem configuration for iterative solver tests. The grid size is N^3. # LI denotes the number of linear iterations.

PDEs	Solver	Stencil	DoF	N	# LI
Poisson [6]	CG-IC(0)	Star-7P	1	304	46
Poisson [6]	CG-IC(1)	Diamond-13P	1	304	29
Convection-Diffusion [31]	GMRES-ILU(0)	Star-7P	1	304	23
Navier-Stokes [29]	GMRES-ILU(0)	Box-27P	4	104	12
Euler [64]	GMRES-ILU(0)	Star-13P	5	112	11
Cahn-Hilliard [35]	GMRES-ILU(0)	Diamond-25P	2	160	22

5.6 Convergence Test of Preconditioners

Finally, we are interested in the efficiency of different variations of ILU preconditioners, especially non-dependency-preserving methods, including multi-color ILU (MC-ILU) [33], ParILU [13] and ParILUT [4], whose IC variants are used. We use the `csrcolor` routine in cuSPARSE [48] for MC-IC, and Ginkgo [3] 1.9.0 for ParIC and ParICT. We conduct the experiments using the Poisson equation in Table 4 on an A100 GPU with the conjugate gradient (CG) solver. The default parameters are used for baselines, where ParIC(0) performs 5 sweeps during factorization, and the maximum fill-in level of ParICT is 2. For StructILU, IC(0) and IC(1) are tested.

Figure 11 shows the iteration and timing results. The dependency-preserving IC(0) algorithm requires 223 iterations to converge to a relative residual norm of 10^{-5}, achieving a total CG iteration time 3.68x faster than ParIC(0), which necessitates 246 iterations. This reduction in iteration count by approximately 10.3% is due to maintaining data dependencies. In contrast, the ParICT algorithm converges in only 179 iterations; however, due to threshold constraints, its convergence rate is slower than that of the IC(1) method using the Diamond-13P stencil, which requires only 148 iterations, achieving a 2.78x speedup relative to ParICT. MC-IC(0) converges after 295 iterations, due to the drop of extra nonzeros resulting from multi-color reordering.

Furthermore, regarding decomposition time, the numerical factorization for IC(0), IC(1), ParIC(0), and MC-IC(0) takes 0.012s, 0.045s, 0.014s, and 0.081s respectively. These results show that StructILU not only converges faster but also offers greater decomposition efficiency. In contrast, ParICT performs numerical and symbolic factorizations together,

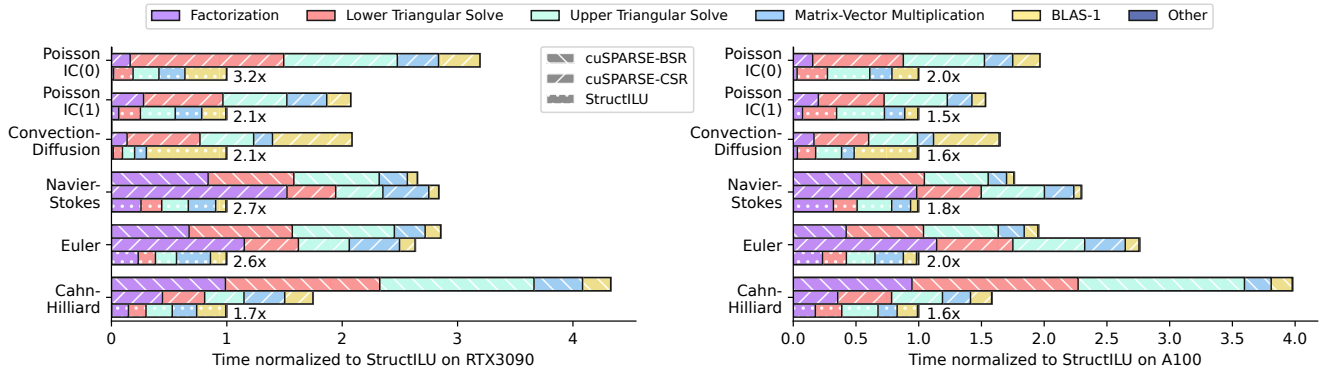


Figure 10: Results of ILU preconditioned iterative methods on RTX3090 and A100 GPUs. Different colors show the time breakdown. StructILU and two baselines are dependency-preserving and converge with the same iteration number. Symbolic factorization and sparse pattern analysis phases of baselines are excluded.

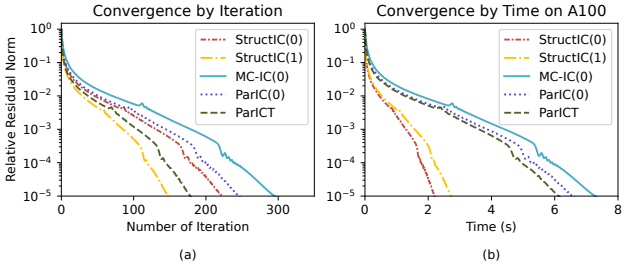


Figure 11: Comparison of iteration and time of different preconditioners for Poisson equation. StructIC denotes the IC version of StructILU.

leading to a total decomposition time of 2.16s. Besides, the coloring of MC-IC(0) takes 0.21s.

6 Related work

Non-dependency-preserving Methods. These algorithms can be classified into two categories. One category is based on reordering the matrix A into a new matrix by using some multi-coloring schemes [52]. Compared to the level-set method, this method captures more parallelism but the resulting ILU preconditioner may have a significant reduction in convergence rate than the serial ILU since many more nonzero elements are dropped. GPU performance of the method can be found in [33, 48]. The other category is inexact methods. ParILU [12, 13] is an inexact ILU preconditioner in which nonzeros in L and U are updated asynchronously without any data dependency for several sweeps during factorization. Convergence degradation can also be faced when using inexact preconditioners [13]. ParILU and its threshold-based version ParILUT [4] are implemented in Ginkgo linear solver library [3]. Blocked versions of asynchronous methods are also designed [28, 44].

Dependency-preserving parallelization of SpTRSV. For parallelizing dependency-preserving ILU on GPUs, much of the focus has been on the triangular solve. A level-set method leveraging multiple SpMV kernels for SpTRSV is implemented [33]. A synchronization-free algorithm [37] eliminates global synchronizations between level sets using the Compressed Sparse Column (CSC) format. A similar approach, offering warp-level sync-free operations for the CSR format, is introduced [19]. To enhance performance in matrices with a low average number of nonzeros per row, a thread-level sync-free method is developed [56]. RecBlock [42] employs a recursive blocking strategy to leverage the 2-D spatial structure of matrices, improving SpTRSV performance. AG-SpTRSV [25] applies kernel templates to optimize performance search. Numerous other advancements in triangular solve optimization have been made, including data formats [34, 41, 54, 62, 70], adaptive scheduling [1, 20, 21, 57, 75], the preprocessing stage [18, 22], and multi-GPU systems [39, 66]. cuSPARSE [46, 47] is an NVIDIA-developed library designed for sparse matrix computations, supporting both ILU(0) and triangular solves, with CSR and BSR formats.

Optimization of ILU-like preconditioners for structured grids. A block ILU scheme using a Star-7P stencil is implemented through a level-set method on GPUs [43]. On Sunway systems, structured grid triangular solve employs a producer-consumer model [32, 63]. DBSR [72] is a data format for vectorizing structured grid sparse triangular solve on CPUs, using block multi-color ordering. Mixed precision preconditioners in structured multigrid on CPUs are also studied [78]. Further research extends to other preconditioners with similar loop-carried dependencies, like the

Gauss-Seidel and SOR smoothers in multigrid methods, both on Sunway processors [76] and multi-core CPUs [7, 59, 77].

Other work with related massive parallel optimization techniques. CuMF_SGD [67] adopts multi-level parallelism for matrix factorization on GPUs using stochastic gradient descent. Centauri [9] improves communication partitioning and hierarchical scheduling for overlap in large language model training. An FPGA framework enabling automatic parallelism for stencil computation is proposed [61]. Tensor Cores in modern GPUs are also utilized for the numerical solution of PDEs [10, 14, 40].

7 Concluding remarks

In this work, we propose StructILU, an efficient GPU framework to parallelize dependency-preserving ILU for structured grid PDEs. We decompose the parallelism inherent in ILU computation into three hierarchical tiers: grid, intra-row, and intra-element. By leveraging the geometric properties of structured grids, StructILU incorporates acyclic dependency task mapping, latency-aware pipeline parallelization, Tensor Core utilization for multiphysics problems, and high-throughput memory transformations, thereby fully exploiting the massive parallelism of GPUs.

We remark that although this work focuses on structured grid problems, the techniques and performance model we propose can be further extended to ILU and other similar preconditioners on unstructured grids, to exploit parallelism while reducing overhead based on geometric information. Additionally, although the experiments presented in this paper were conducted solely in double precision and on NVIDIA GPUs, the proposed approach is applicable to other precision formats and GPUs from different vendors as well.

Acknowledgments

This work was supported in part by National Key R&D Program of China (No. 2023YFA1011703), National Natural Science Foundation of China (No. 12131002, 12288101), Changsha Science and Technology Bureau (No. KH2301001), and Shanghai Municipal Science and Technology Major Project. In addition, special thanks to Prof. Weifeng Liu for valuable suggestions on the experimental design.

References

[1] Najeeb Ahmad, Buse Yilmaz, and Didem Unat. 2020. A prediction framework for fast sparse triangular solves. In *European Conference on Parallel Processing*. Springer, 529–545.

[2] Edward Anderson and Youcef Saad. 1989. Solving sparse triangular linear systems on parallel computers. *International Journal of High Speed Computing* 1, 01 (1989), 73–95.

[3] Hartwig Anzt, Terry Cojean, Goran Flegar, Fritz Göbel, Thomas Grützmacher, Pratik Nayak, Tobias Ribizel, Yuhsiang Mike Tsai, and Enrique S Quintana-Ortí. 2022. Ginkgo: A modern linear operator algebra framework for high performance computing. *ACM Transactions on Mathematical Software (TOMS)* 48, 1 (2022), 1–33.

[4] Hartwig Anzt, Tobias Ribizel, Goran Flegar, Edmond Chow, and Jack Dongarra. 2019. ParILUT-a parallel threshold ILU for GPUs. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 231–241.

[5] Michele Benzi, Wayne Joubert, and Gabriel Mateescu. 1999. Numerical experiments with parallel orderings for ILU preconditioners. *Electronic Transactions on Numerical Analysis* 8 (1999), 88–114.

[6] Ed Bueler. 2020. *PETSc For partial differential equations: Numerical solutions in C and Python*. SIAM.

[7] Huanqi Cao, Shizhi Tang, Qianchao Zhu, Bowen Yu, and Wenguang Chen. 2023. Mat2Stencil: A Modular Matrix-Based DSL for Explicit and Implicit Matrix-Free PDE Solvers on Structured Grid. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2 (2023), 686–715.

[8] Arielle Carr, Eric de Sturler, and Serkan Gugercin. 2021. Preconditioning parametrized linear systems. *SIAM Journal on Scientific Computing* 43, 3 (2021), A2242–A2267.

[9] Chang Chen, Xiuhong Li, Qianchao Zhu, Jiangfei Duan, Peng Sun, Xingcheng Zhang, and Chao Yang. 2024. Centauri: Enabling efficient scheduling for communication-computation overlap in large model training via communication partitioning. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 178–191.

[10] Yuetao Chen, Kun Li, Yuhao Wang, Donglin Bai, Lei Wang, Lingxiao Ma, Liang Yuan, Yunquan Zhang, Ting Cao, and Mao Yang. 2024. ConvStencil: Transform Stencil Computation to Matrix Multiplication on Tensor Cores. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 333–347.

[11] Jack Choquette, Wishwesh Gandhi, Olivier Giroux, Nick Stam, and Ronny Krashinsky. 2021. Nvidia a100 tensor core gpu: Performance and innovation. *IEEE Micro* 41, 2 (2021), 29–35.

[12] Edmond Chow, Hartwig Anzt, and Jack Dongarra. 2015. Asynchronous iterative algorithm for computing incomplete factorizations on GPUs. In *High Performance Computing: 30th International Conference, ISC High Performance 2015, Frankfurt, Germany, July 12-16, 2015, Proceedings 30*. Springer, 1–16.

[13] Edmond Chow and Aftab Patel. 2015. Fine-grained parallel incomplete LU factorization. *SIAM journal on Scientific Computing* 37, 2 (2015), C169–C193.

[14] Cu Cui. 2024. Acceleration of tensor-product operations with tensor cores. *ACM Transactions on Parallel Computing* 11, 4 (2024), 1–24.

[15] James W Demmel. 1997. *Applied numerical linear algebra*. SIAM.

[16] Shun Doi and Takumi Washio. 1999. Ordering strategies and related techniques to overcome the trade-off between parallelism and convergence in incomplete factorizations. *Parallel Comput.* 25, 13-14 (1999), 1995–2014.

[17] Xin Dong and Gene Cooperman. 2011. A bit-compatible parallelization for ILU (k) preconditioning. In *Euro-Par 2011 Parallel Processing: 17th International Conference, Euro-Par 2011, Bordeaux, France, August 29-September 2, 2011, Proceedings, Part II 17*. Springer, 66–77.

[18] Ernesto Dufrechou and Pablo Ezzatti. 2018. A new GPU algorithm to compute a level set-based analysis for the parallel solution of sparse triangular systems. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 920–929.

[19] Ernesto Dufrechou and Pablo Ezzatti. 2018. Solving sparse triangular linear systems in modern GPUs: a synchronization-free algorithm. In *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. IEEE, 196–203.

[20] Ernesto Dufrechou, Pablo Ezzatti, Manuel Freire, and Enrique S Quintana-Ortí. 2021. Machine learning for optimal selection of sparse

triangular system solvers on GPUs. *J. Parallel and Distrib. Comput.* 158 (2021), 47–55.

[21] Ernesto Dufrechou, Pablo Ezzatti, and Enrique S Quintana-Ortí. 2019. Automatic selection of sparse triangular linear system solvers on GPUs through machine learning techniques. In *2019 31st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 41–47.

[22] Manuel Freire, Juan Ferrand, Franco Seveso, Ernesto Dufrechou, and Pablo Ezzatti. 2023. A GPU method for the analysis stage of the SPTRSV kernel. *The Journal of Supercomputing* 79, 13 (2023), 15051–15078.

[23] Tom Goodale, Gabrielle Allen, Gerd Lanfermann, Joan Massó, Thomas Radke, Edward Seidel, and John Shalf. 2003. The cactus framework and toolkit: design and applications: invited talk. In *High Performance Computing for Computational Science—VECPAR 2002: 5th International Conference Porto, Portugal, June 26–28, 2002 Selected Papers and Invited Talks 5*. Springer, 197–227.

[24] William D Gropp, LC McInnes, MD Tidriri, and DE Keyes. 1997. Parallel implicit PDE computations: Algorithms and software. In *Proceedings of Parallel CFD*, Vol. 97. Citeseer, 333–344.

[25] Zhengding Hu, Jingwei Sun, Zhongyang Li, and Guangzhong Sun. 2024. AG-SpTRSV: An Automatic Framework to Optimize Sparse Triangular Solve on GPUs. *ACM Transactions on Architecture and Code Optimization* (2024).

[26] Carlo Janna, Massimilano Ferronato, and Giuseppe Gambolati. 2010. A block FSAI-ILU parallel preconditioner for symmetric positive definite linear systems. *SIAM Journal on Scientific Computing* 32, 5 (2010), 2468–2484.

[27] Wayne Joubert and Thomas Oppe. 1994. Improved SSOR and incomplete Cholesky solution of linear equations on shared memory and distributed memory parallel computers. *Numerical Linear Algebra with Applications* 1, 3 (1994), 287–311.

[28] Aditya Kashi and Sivakumaran Nadarajah. 2021. An asynchronous incomplete block LU preconditioner for computational fluid dynamics on unstructured grids. *SIAM Journal on Scientific Computing* 43, 1 (2021), C1–C30.

[29] Cetin C Kiris, Dochan Kwak, and Stuart E Rogers. 2003. Incompressible Navier-Stokes solvers in primitive variables and their applications to steady and unsteady flow simulations. *Numerical Simulations of Incompressible Flows* (2003), 3–34.

[30] Byungjoon Lee and Chohong Min. 2021. Optimal preconditioners on solving the Poisson equation with Neumann boundary conditions. *J. Comput. Phys.* 433 (2021), 110189.

[31] Randall J LeVeque and Randall J Leveque. 1992. *Numerical methods for conservation laws*. Vol. 214. Springer.

[32] Jianjiang Li, Jiabi Liang, Wei Xue, Zhengding Hu, Lin Li, and Jinliang Shi. 2024. Toward efficient structured-grid triangular solver on sunway many-core processors. *The Journal of Supercomputing* 80, 8 (2024), 10610–10636.

[33] Ruipeng Li and Yousef Saad. 2013. GPU-accelerated preconditioned iterative linear solvers. *The Journal of Supercomputing* 63 (2013), 443–466.

[34] Ruipeng Li and Chaoyu Zhang. 2020. Efficient parallel implementations of sparse triangular solves for GPU architectures. In *Proceedings of the 2020 SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, 106–117.

[35] Yibao Li, Lujing Zhang, Qing Xia, Qian Yu, and Junseok Kim. 2021. An unconditionally energy-stable second-order time-accurate numerical scheme for the coupled Cahn–Hilliard system in copolymer/homopolymer mixtures. *Computational Materials Science* 200 (2021), 110809.

[36] Zhongze Li and Yousef Saad. 2006. SchurRAS: A restricted version of the overlapping Schur complement preconditioner. *SIAM Journal on Scientific Computing* 27, 5 (2006), 1787–1801.

[37] 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 2016: Parallel Processing: 22nd International Conference on Parallel and Distributed Computing, Grenoble, France, August 24-26, 2016, Proceedings 22*. Springer, 617–630.

[38] Xiaoyan Liu, Yi Liu, Hailong Yang, Jianjin Liao, Mingzhen Li, Zhongzhi Luan, and Depei Qian. 2022. Toward accelerated stencil computation by adapting tensor core unit on gpu. In *Proceedings of the 36th ACM International Conference on Supercomputing*. 1–12.

[39] Yang Liu, Nan Ding, Piyush Sao, Samuel Williams, and Xiaoye Sherry Li. 2023. Unified Communication Optimization Strategies for Sparse Triangular Solver on CPU and GPU Clusters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[40] Yuechen Lu, Lijie Zeng, Tengcheng Wang, Xu Fu, Wenxuan Li, Helin Cheng, Dechuang Yang, Zhou Jin, Marc Casas, and Weifeng Liu. 2024. Amgt: Algebraic multigrid solver on tensor cores. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–16.

[41] Zhengyang Lu and Weifeng Liu. 2023. Tilesptrsv: a tiled algorithm for parallel sparse triangular solve on gpus. *CCF Transactions on High Performance Computing* 5, 2 (2023), 129–143.

[42] Zhengyang Lu, Yuyao Niu, and Weifeng Liu. 2020. Efficient block algorithms for parallel sparse triangular solve. In *Proceedings of the 49th International Conference on Parallel Processing*. 1–11.

[43] Lixiang Luo, Jack R Edwards, Hong Luo, and Frank Mueller. 2015. A fine-grained block ILU scheme on regular structures for GPGPUs. *Computers & Fluids* 119 (2015), 149–161.

[44] Wenpeng Ma and Xiao-Chuan Cai. 2021. Point-block incomplete LU preconditioning with asynchronous iterations on GPU for multiphysics problems. *The International Journal of High Performance Computing Applications* 35, 2 (2021), 121–135.

[45] J Andvandervorst Meijerink and Henk A Van Der Vorst. 1977. An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix. *Math. Comp.* 31, 137 (1977), 148–162.

[46] Maxim Naumov. 2011. Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the GPU. *NVIDIA Corp., Westford, MA, USA, Tech. Rep. NVR-2011* 1 (2011).

[47] Maxim Naumov. 2012. Parallel incomplete-LU and Cholesky factorization in the preconditioned iterative methods on the GPU. *Nvidia Technical Report NVR-2012-003* (2012).

[48] Maxim Naumov, Patrice Castonguay, and Jonathan Cohen. 2015. Parallel graph coloring with applications to the incomplete-lu factorization on the gpu. *Nvidia White Paper* (2015).

[49] Natalia K Nikolova, Helen W Tam, and Mohamed H Bakr. 2004. Sensitivity analysis with the FDTD method on structured grids. *IEEE Transactions on Microwave Theory and Techniques* 52, 4 (2004), 1207–1216.

[50] Jongsoo Park, Mikhail Smelyanskiy, Narayanan Sundaram, and Pradeep Dubey. 2014. Sparsifying synchronization for high-performance shared-memory sparse triangular solver. In *Supercomputing: 29th International Conference, ISC 2014, Leipzig, Germany, June 22-26, 2014. Proceedings 29*. Springer, 124–140.

[51] Eugene L Poole and James M Ortega. 1987. Multicolor ICCG methods for vector computers. *SIAM J. Numer. Anal.* 24, 6 (1987), 1394–1418.

[52] Yousef Saad. 2003. *Iterative methods for sparse linear systems*. SIAM.

[53] Joel H Saltz. 1990. Aggregation methods for solving sparse triangular systems on multiprocessors. *SIAM J. Sci. Statist. Comput.* 11, 1 (1990), 123–144.

[54] Barry Smith and Hong Zhang. 2011. Sparse triangular solves for ILU revisited: Data layout crucial to better performance. *The International*

Journal of High Performance Computing Applications 25, 4 (2011), 386–391.

[55] Barry F Smith. 1997. Domain decomposition methods for partial differential equations. In *Parallel numerical algorithms*. Springer, 225–243.

[56] 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 *Proceedings of the 49th International Conference on Parallel Processing*. 1–11.

[57] Brad Suchoski, Caleb Severn, Manu Shantharam, and Padma Raghavan. 2012. Adapting sparse triangular solution to GPUs. In *2012 41st International Conference on Parallel Processing Workshops*. IEEE, 140–148.

[58] Kengo Suzuki, Takeshi Fukaya, and Takeshi Iwashita. 2023. A novel ILU preconditioning method with a block structure suitable for SIMD vectorization. *J. Comput. Appl. Math.* 419 (2023), 114687.

[59] Ruhollah Tavakoli. 2010. Parallelizing Sequential Sweeping on Structured Grids–Fully Parallel SOR/ILU preconditioners for Structured n-Diagonal Matrices. *arXiv preprint arXiv:1008.3699* (2010).

[60] Henk A van der Vorst. 1989. High performance preconditioning. *SIAM J. Sci. Statist. Comput.* 10, 6 (1989), 1174–1185.

[61] Shuo Wang and Yun Liang. 2017. A comprehensive framework for synthesizing stencil algorithms on FPGAs using OpenCL model. In *Proceedings of the 54th Annual Design Automation Conference 2017*. 1–6.

[62] Xinliang Wang, Weifeng Liu, Wei Xue, and Li Wu. 2018. swSpTRSV: A fast sparse triangular solve with sparse level tile layout on sunway architectures. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 338–353.

[63] Xinliang Wang, Ping Xu, Wei Xue, Yulong Ao, Chao Yang, Haohuan Fu, Lin Gan, Guangwen Yang, and Weimin Zheng. 2018. A fast sparse triangular solver for structured-grid problems on sunway many-core processor SW26010. In *Proceedings of the 47th International Conference on Parallel Processing*. 1–11.

[64] Andrew M Wissink, Anastasios S Lyrintzis, and Roger C Strawn. 1996. Parallelization of a three-dimensional flow solver for Euler rotorcraft aerodynamics predictions. *AIAA Journal* 34, 11 (1996), 2276–2283.

[65] Yuanzhe Xi and Yousef Saad. 2017. A rational function preconditioner for indefinite sparse linear systems. *SIAM Journal on Scientific Computing* 39, 3 (2017), A1145–A1167.

[66] Chenhao Xie, Jieyang Chen, Jesun Firoz, Jiajia Li, Shuaiwen Leon Song, Kevin Barker, Mark Raugas, and Ang Li. 2021. Fast and scalable sparse triangular solver for multi-gpu based hpc architectures. In *Proceedings of the 50th International Conference on Parallel Processing*. 1–11.

[67] Xiaolong Xie, Wei Tan, Liana L Fong, and Yun Liang. 2017. CuMF_SGD: Parallelized stochastic gradient descent for matrix factorization on GPUs. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*. 79–92.

[68] Tianshi Xu, Ruipeng Li, and Daniel Osei-Kuffuor. 2023. A two-level GPU-accelerated incomplete LU preconditioner for general sparse linear systems. *arXiv preprint arXiv:2303.08881* (2023).

[69] Ichitaro Yamazaki, Sivasankaran Rajamanickam, Erik G Boman, Mark Hoemmen, Michael A Heroux, and Stanimire Tomov. 2014. Domain decomposition preconditioners for communication-avoiding Krylov methods on a hybrid CPU/GPU cluster. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 933–944.

[70] Ichitaro Yamazaki, Sivasankaran Rajamanickam, and Nathan Ellingwood. 2020. Performance portable supernode-based sparse triangular solver for manycore architectures. In *Proceedings of the 49th International Conference on Parallel Processing*. 1–11.

[71] Chao Yang, Wei Xue, Haohuan Fu, Hongtao You, Xinliang Wang, Yulong Ao, Fangfang Liu, Lin Gan, Ping Xu, Lanning Wang, et al. 2016. 10M-core scalable fully-implicit solver for nonhydrostatic atmospheric dynamics. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 57–68.

[72] Xiaojian Yang, Shengguo Li, Fan Yuan, and Dezun Dong. 2024. DBSR: An Efficient Storage Format for Vectorizing Sparse Triangular Solvers on Structured Grids. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–14.

[73] Xin You, Hailong Yang, Zhonghui Jiang, Zhongzhi Luan, and Depei Qian. 2021. DRStencil: Exploiting data reuse within low-order stencil on GPU. In *2021 IEEE 23rd Int Conf on High Performance Computing & Communications; 7th Int Conf on Data Science & Systems; 19th Int Conf on Smart City; 7th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys)*. IEEE, 63–70.

[74] David M Young. 2014. *Iterative solution of large linear systems*. Elsevier.

[75] Feng Zhang, Jiya Su, Weifeng Liu, Bingsheng He, Ruofan Wu, Xiaoyong Du, and Rujia Wang. 2021. Yuenyeungsptrsv: a thread-level and warp-level fusion synchronization-free sparse triangular solve. *IEEE Transactions on Parallel and Distributed Systems* 32, 9 (2021), 2321–2337.

[76] Qianchao Zhu, Hao Luo, Chao Yang, Mingshuo Ding, Wanwang Yin, and Xinhui Yuan. 2021. Enabling and scaling the HPCG benchmark on the newest generation Sunway supercomputer with 42 million heterogeneous cores. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13.

[77] Yi Zong, Xinliang Wang, Haopeng Huang, Chensong Zhang, Xiaowen Xu, Jian Sun, Bowen Yan, Qin Wang, Sicong Li, Zhaohui Ding, et al. 2024. POSTER: StructMG: A Fast and Scalable Structured Multigrid. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 478–480.

[78] Yi Zong, Peinan Yu, Haopeng Huang, and Wei Xue. 2024. FP16 Acceleration in Structured Multigrid Preconditioner for Real-World Applications. In *Proceedings of the 53rd International Conference on Parallel Processing*. 52–62.