Loop-Carried Dependence Transformation for Parallel Sparse Solvers on GPUs

Da Ma[1,†], Mohammad Mahdi Salehi Dezfuli[1,†], Amir Mohammad Tavakkoli[2], Samirasadat Jamalidinan[1], Hossein Albakri[1], Mary Hall[2], and Kazem Cheshmi[1]
[1]McMaster University, Hamilton, ON, Canada [2]University of Utah, Salt Lake City, UT, USA
{mad29, salehm32, jamalids, albakrih, cheshmi}@mcmaster.ca, {amir.tavakkoli, mhall}@cs.utah.edu

***Abstract*—Loop-carried dependences create significant computational bottlenecks in scientific solvers, hindering the effective utilization of massive GPU parallelism. Existing libraries are typically restricted to specific numerical methods or matrix types, making them difficult to scale, while previous compiler techniques often target only fully parallel loops or multicore CPUs. We propose a loop-carried dependence parallelism (LCD) transformation that automatically enables GPU parallelism for loops within sparse solvers. LCD lowers identified loops from a broad range of sparse operations, mapping them to thread-blocks and warps while implementing efficient and correct synchronization strategies. This framework utilizes a thread-block manager and a thread manager, both orchestrated by a central scheduler. Implemented as a Python just-in-time (JIT) compiler, the code generated by LCD achieves a geometric mean speedup of $1.60\times$-$4.56\times$ over the cuSPARSE library on H100 GPUs across various operations.**

***Index Terms*—Parallel processing, Parallel programming, Sparse matrices, Directed acyclic graph, Dynamic compiler.**

I. Introduction

Several scientific solvers [1] involve loop-carried dependences, occuring when there are two accesses to the same memory location in different loop iterations, with at least one write [2]. Because a dependence is an assumption that two memory accesses may point to the same location, running full loop iterations in parallel under this conservative assumption can lead to incorrect results. These dependences limit the scalability of scientific solvers across parallel architectures, particularly GPUs, where abundant parallelism exists. Due to the nature of sparse matrices, loop-carried dependences in these solvers are typically sparse, exposing opportunities to execute some iterations concurrently. However, this available parallelism is input-sensitive, which makes task decomposition and mapping highly challenging given the vast range of input possibilities and the complex hierarchy of parallelism in GPUs across thread-blocks, warps, and threads.

To parallelize loop-carried dependences on GPUs, major synchronization strategies include wavefront techniques [3], [4] and barrier-free (or synchronization-free) methods [5]. Wavefront parallelism employs preprocessing to identify independent iterations and separates their execution using global barriers; each barrier typically translates to a separate GPU kernel launch. Conversely, barrier-free methods use finer-grained interactions, such as point-to-point synchronization, to improve load balance and reduce the number of kernel launches to one [6]. Existing libraries [7]–[9] benefit from these techniques by tailoring implementations to specific applications or matrix classes. However, these architecture-specific solutions introduce significant complexity for the developer, often necessitating entirely new implementations to maintain high performance across diverse platforms.

Compilers offer the flexibility to support cross-platform execution, and can provide code specialization based on runtime information from the application. REPTILE tensor compiler [10] and Sympiler [11], [12] generate sequential tiled code for Cholesky for single thread on CPU. Inspector-executor methods such as sparse polyhedral framework [13], [14] and ParSy [15] generate a Directed Acyclic Graph (DAG) to represent dependence between iterations. Then they inspect the DAG to run independent iterations in parallel. These compilers predominantly rely on wavefront parallelism. These techniques typically provide coarse-grained thread parallelism, sufficient for CPUs. However, constructing workloads with hierarchical granularity is essential for efficient GPU execution while ensuring proper interaction to guarantee correctness.

This paper introduces the LCD compiler transformation to enable partial parallelism for sparse solver operations on GPUs. The transformation decomposes loop iterations into hierarchical tasks for thread-blocks, warps, and threads using block and thread manager steps. To ensure correctness, dependence analysis identifies necessary synchronization points, which are then implemented at the thread-block and warp levels for scalability. Because the optimal task decomposition and thread mapping depend on the input matrix, LCD employs a scheduler to navigate this search space. Implemented as a compiler directive within a Pythonic JIT framework, the transformation leverages runtime sparsity patterns and Pythonic expressivity to generate high-performance code. Evaluated on an H100 GPU using SuiteSparse [16] matrices, the JIT-generated code achieves geometric mean speedups over cuSPARSE of $1.62\times$ for sparse lower triangular solve (SpTRSV) with compressed sparse column (CSC), $1.60\times$ for SpTRSV with compressed sparse row (CSR), $2.81\times$ for Sparse ILU0 (SpILU0) factorization, $4.56\times$ for SpILU0-solve, $3.09\times$ for Gauss-Seidel, and $1.70\times$ for supernodal SpTRSV.

[†]Da Ma and Mohammad Mahdi Salehi Dezfuli contributed equally.

SC26, November 15-20, 2026, Chicago, Illinois, USA
979-8-3195-4789-7/26/$31.00

II. Motivation

We use the sparse triangular solver (SpTRSV) code shown in Listing 1 to illustrate the parallelism challenges posed by a loop-carried dependence on GPUs. The SpTRSV code solves a linear system $Ax = b$, where A is a given lower triangular matrix, b is a given right-hand side vector, and x is the unknown solution vector. The code uses the CSC format, where nonzero elements are stored column by column in array `Ax`. The corresponding row indices and column pointers are stored in `Ai` and `Ap`, respectively. In Listing 1, the read operation in line 5 (reading `x[i]`) and the write operation at line 7 (writing to `x[Ai[j]]`) reference the same memory location when indices i and `Ai[j]` are equal in different iterations; i.e., a loop-carried dependence. To maintain correctness when running iterations in parallel, these dependence relations must not be violated. While the dependence-governing arrays `Ai` and `Ap` are known components of the storage format, the specific values within them, which define the matrix sparsity and memory access patterns, are only available at runtime.

Listing 1: Sparse lower triangular solve (SpTRSV) CSC CPU

```
1 @Parallel(block_map='i', thread_map='j')
2 def sptrsv_csc(n, Ap, Ai, Ax, b):
3  x = np.zeros(n) # zero initialization
4  for i in range(n):
5   x[i] = (b[i] - x[i]) / Ax[Ap[i]]
6   for j in range(Ap[i]+1, Ap[i+1]):
7    x[Ai[j]] += Ax[j] * x[i]
8  return x
```

Consequently, the dependence DAG should be constructed at runtime based on these contents. Each vertex of the DAG represents an iteration of the loop that is annotated with `block_map` within the `Parallel` decorator, as shown in line 1 in Listing 1. This process necessitates a compile-time analysis to uncover dependences, along with a runtime module to materialize the DAG. Once the DAG is created, independent iterations are mapped to GPU thread-blocks to ensure coarse-grained parallelism by carefully transforming the loop. Listing 2 shows an example of the transformed GPU kernel for the SpTRSV code shown in Listing 1. As shown in line 2, one iteration of the specified loop `i` is mapped to one thread-block. Multiple iterations can also be mapped to one thread-block, depending on the granularity of the tasks. While the DAG dictates the safe execution order between dependent loop iterations across thread-blocks, concurrent memory updates by parallel threads writing to `x` in line 7 of Listing 1 also need to be made mutually exclusive, enforced by the `atomic` operation in line 9 of Listing 2.

To use all threads within a thread-block, fine-grained parallelism is essential to map innermost loops to threads. The innermost loop `j` designated for fine-grained parallelism is specified in the directive in line 1 in Listing 1. When mapping the loop iterations to threads, the loop body is split into parallel and thread-invariant (sequential) regions. Extracting these two, needs synchronization between them, as shown in line 7, as well as thread mapping in the parallel region, shown in line 8 in Listing 2. In Listing 2, two iterations are mapped to each thread to improve data locality. We assume the loop's trip count is a multiple of UF for conciseness. Selecting efficient UF depends on the matrix sparsity pattern, specifically the number of nonzeros in column `i`. These parameters, and their dependence on the sparsity of `A`, create a search space.

Listing 2: One GPU thread in SpTRSV CSC

```
1 def sptrsv_csc_kernel(Ap, Ai, Ax, x, b):
2  i = cuda.blockIdx.x
3  tid = cuda.threadIdx.x
4  blockDim, UF = cuda.blockDim.x, 2
5  if tid == 0:
6   x[i] = (b[i] - x[i]) / Ax[Ap[i]]
7  cuda.syncthreads()
8  for j in range(Ap[i]+1+tid, Ap[i+1],UF*blockDim):
9   cuda.atomic.add(x, Ai[j], Ax[j] * x[i])
10  cuda.atomic.add(x, Ai[j+1], Ax[j+1] * x[i])
11 return x
```

Figure 1 shows how the resulting search space from thread granularity parameters, i.e., `blockDim` and UF, can grow across three different matrix code and across all symmetric positive definite (SPD) matrices from SuiteSparse [16]. For Figure 1a, the illustrated search space specifically sweeps the value of `blockDim` and UF in line 8 in Listing 2. A larger `blockDim` creates more threads in a block, and a larger UF makes the threads coarser. The color intensity shows how frequent a schedule is chosen to be optimal, which varies significantly across matrices. Consequently, the generated code must change dynamically as the input sparsity changes. As shown in Figure 1b and 1c, in addition to sparsity, the matrix data structure and kernel operation also affect the computational pattern of the kernel and, consequently, the optimal schedule. Our proposed LCD transformation explores this space and provides gmean speedups of $1.62\times$, $1.60\times$, and $3.09\times$ over cuSPARSE for SpTRSV-CSC, SpTRSV-CSR, and SpILU0, respectively, on H100 GPUs.

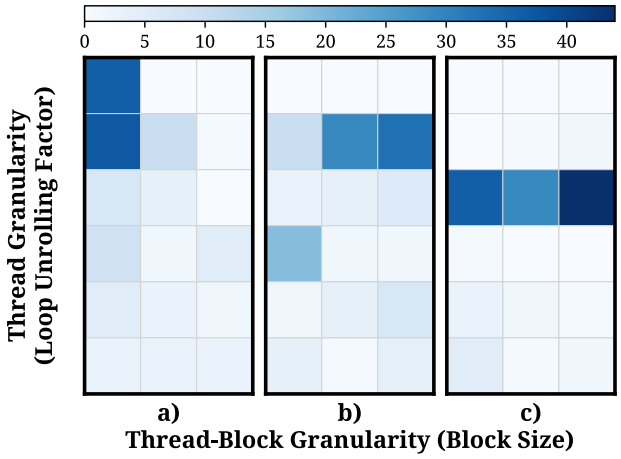
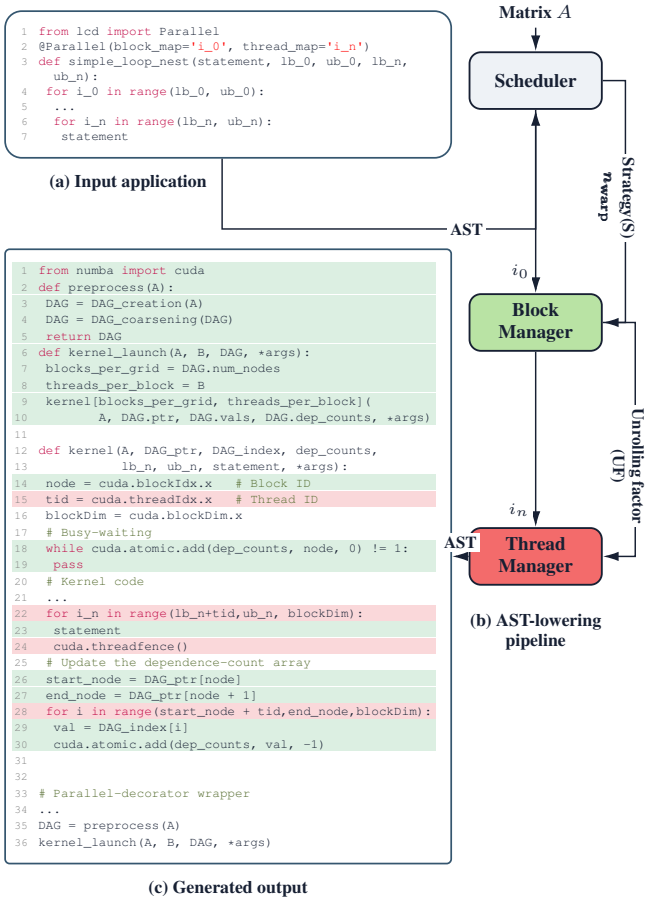


Fig. 1: Search space of the LCD transformation for a) SpTRSV CSC, b) SpTRSV CSR, c) Sparse ILU0. x axis and y axis show different thread and thread-block granularity across a limited schedule space. The color intensity shows the frequency of a schedule to be optimal across matrices.

III. Loop-Carried Dependence Parallelism (LCD) Transformation

This section provides an overview of LCD (Figure 2), followed by discussing its two lowering steps and scheduler.




```
1  from lcd import Parallel
2  @Parallel(block_map='i_0', thread_map='i_n')
3  def simple_loop_nest(statement, lb_0, ub_0, lb_n,
     ub_n):
4   for i_0 in range(lb_0, ub_0):
5    ...
6    for i_n in range(lb_n, ub_n):
7     statement
```

```
1  from numba import cuda
2  def preprocess(A):
3   DAG = DAG_creation(A)
4   DAG = DAG_coarsening(DAG)
5   return DAG
6  def kernel_launch(A, B, DAG, *args):
7   blocks_per_grid = DAG.num_nodes
8   threads_per_block = B
9   kernel[blocks_per_grid, threads_per_block](
10          A, DAG.ptr, DAG.vals, DAG.dep_counts, *args)
11
12 def kernel(A, DAG_ptr, DAG_index, dep_counts,
13             lb_n, ub_n, statement, *args):
14  node = cuda.blockIdx.x    # Block ID
15  tid = cuda.threadIdx.x    # Thread ID
16  blockDim = cuda.blockDim.x
17  # Busy-waiting
18  while cuda.atomic.add(dep_counts, node, 0) != 1:
19   pass
20  # Kernel code
21  ...
22  for i_n in range(lb_n+tid,ub_n, blockDim):
23   statement
24   cuda.threadfence()
25  # Update the dependence-count array
26  start_node = DAG_ptr[node]
27  end_node = DAG_ptr[node + 1]
28  for i in range(start_node + tid,end_node,blockDim):
29   val = DAG_index[i]
30   cuda.atomic.add(dep_counts, val, -1)
31
32
33 # Parallel-decorator wrapper
34 ...
35 DAG = preprocess(A)
36 kernel_launch(A, B, DAG, *args)
```

Fig. 2: LCD compiler pipeline. In (c), lines generated by the Block Manager are highlighted in green, whereas those generated by the Thread Manager are highlighted in red.

A. Overview

The LCD transformation takes the abstract syntax tree (AST) of a loop nest and a matrix as its inputs and generates a parallel GPU code. The input AST for LCD is a loop nest comprising $l_0, l_1, \ldots, l_n$, as demonstrated in Figure 2a. The output code shown in Figure 2c is generated after passing through the thread-block manager and the thread manager, guided by the input-sensitive scheduler as illustrated in Figure 2b. The `@Parallel` decorator in line 2 in Figure 2a is used to annotate the loops to be used for the different stages of the AST lowering. Formally, the parallel decorator is defined as `transformed_code = parallel(func, block_map, thread_map)`. Here, `func` is a Python function containing a loop with potential loop-carried dependences, `block_map` identifies the loop mapped to thread-blocks for the block manager, and `thread_map` identifies the loop mapped to threads for the thread manager. The output is a compiled function with a transformed AST containing both the device kernel and the host-side kernel-launch code.

As illustrated in Figure 2c, the host component of the generated code includes a preprocessing step alongside the kernel launch. The preprocessing part generates a dependence DAG to safely order the execution of iterations based on the sparsity pattern, as illustrated in Figure 3, followed by a coarsening step to improve data locality when iterations densely depend on each other. Once computed, the DAG is passed to the kernel-launch code to coordinate execution on the device. The generated kernel and the launch kernel are managed to execute the computation with different levels of granularity. This adaptability allows for various parallelization strategies sensitive to the input sparsity pattern.

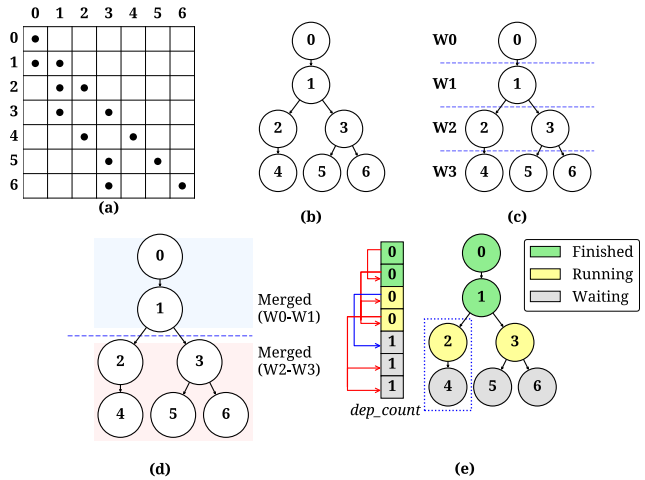


Fig. 3: (a) A sparse lower triangular matrix. (b) The dependence DAG of loop `i` in Listing 1 for the matrix in Figure 3a. (c) A wavefront parallelism execution schedule, where dashed blue lines delineate the sequential execution steps W0–W3. (d) A coarsened schedule achieved by merging wavefronts to improve locality; the shaded blue and red regions highlight the grouped execution levels of W0–W1 and W2–W3, respectively. (e) A barrier-free execution using atomic dependence counting with a single kernel launch. Node colors denote real-time execution states: green for finished, yellow for actively running, and grey for waiting. The `dep_count` array tracks unsatisfied dependences, which are updated dynamically as parent nodes complete.

Figure 3 shows how the DAG is generated for an input matrix and the code in Listing 1, and how the resulting parallel kernel can be launched with a varying number of barriers (i.e., kernel launches). Figure 3a shows a lower triangular sparse matrix represented in a 2D array for better illustration. The DAG for the SpTRSV loop `i` in Listing 1 is shown in Figure 3b. Figure 3c illustrates wavefront parallelism with four wavefronts, e.g. `wavefronts[2]=[2,3]`. In Figure 3c, each iteration of a wavefront is assigned to a thread-block, requiring four kernel launches (barriers). Figure 3d shows how thread-blocks are coarsened to run two iterations, re-

sulting in fewer launches (two). Finally, Figure 3e shows the minimum number of launches, which is one, achieved using a barrier-free method. Figure 3e also demonstrates the barrier-free implementation, where dependences are handled using a `dep_count` array that holds the count of remaining dependences for each node at any given time of execution.

B. Block Manager

The Block Manager lowers the code to map iterations of the specified loop to GPU thread-blocks. Algorithm 1 details the internal steps of the block manager. The inputs to this algorithm are the AST of the input code, the block synchronization strategy S, and the iterator variable of the mapped loop $loopvar$. The outputs of this process are $hAST$ and $dAST$, representing the abstract syntax trees of the preprocessing and output code, respectively.

Code Analysis: In order to ensure safety of the parallel code after mapping iterations of `i1` in Figure 2b to thread-blocks, Algorithm 1 analyzes the AST of the given loop. Line 1 finds the loop with the given $loopvar$ and stores an intermediate AST to $iAST$. In line 2 of Algorithm 1, `extract_deps` function extracts the governing relations of the DAG by finding RAW dependences from the code. Each dependence is described using Presburger arithmetic. Three factors are involved in the constraints of a dependence relation. Loop boundaries specify the free variable bounds. Indices of accesses describe equalities or inequalities, in case of block accesses, between free variables or functions of free variables. Lastly, control structures produce more equality or inequality constraints for each relation. Equations 1 and 2 show the dependence extracted from Listing 1. After extracting all dependences, the function excludes unsatisfiable dependences based on properties of the access functions [17].

Algorithm 1: Block Manager

Input : AST, S, $loopvar$, n_{warp}
Output: $dAST$,$hAST$

/* Code analysis */
1 $iAST \leftarrow find\_loop(loopvar, AST)$
2 $\mathcal{C} \leftarrow extract\_deps(iAST)$
3 **if** $\mathcal{C}.has\_reduction()$ **then** $iAST \leftarrow$ add_atomic$(iAST, \mathcal{C})$
/* Preprocessing code generation */
4 $hAST \leftarrow generate\_DAG\_creation(\mathcal{C})$
5 **if** $S = BF$ **then** $hAST \leftarrow add\_dep\_count(hAST)$
6 **if** $S = WF$ **then** $hAST \leftarrow add\_create\_waves(hAST)$
7 $hAST \leftarrow add\_coarsen(hAST, n_{warp})$
/* Output code generation */
8 $dAST \leftarrow convert\_to\_device(iAST)$
9 **if** $S = WF$ **then** $dAST \leftarrow map\_to\_wave(dAST)$
10 **if** $S = BF$ **then**
11 $\quad dAST \leftarrow add\_flag\_update(dAST)$
12 $\quad dAST \leftarrow add\_flag\_check(dAST)$
13 **end**
14 $dAST \leftarrow generate\_launch\_code(dAST, S)$

$$D_1 = \{i \rightarrow k \mid \exists j_1, j_2 \; Ap[i]+1 \leq j_1 < Ap[i+1] \wedge Ap[k]+1 \leq j_2 < Ap[k+1] \wedge Ai[j_1] = Ai[j_2]\} \quad (1)$$

$$D_2 = \{i \rightarrow k \mid \exists j \; (Ap[i]+1 \leq j < Ap[i+1] \wedge Ai[j] = k)\} \quad (2)$$

Line 3 in Algorithm 1 checks if atomic operation is needed for code generation. Some dependences exhibit reduction pattern. It's crucial to distinguish them because the involved operation should be executed as an atomic operation. The `has_reduction` method checks if a dependence exhibits this property. Then `add_atomic` marks statements in the $iAST$ with atomic. A reduction operation accumulates values into a memory location using an associative and commutative operator (e.g., +), manifesting as a RAW dependence pattern [2]. Equation 1 is a dependence that indicates a reduction.

Preprocessing Code Generation: To utilize the sparsity-based dependences, `generate_DAG_creation` on line 4 generates a code based on the relations to materialize the DAG from the sparsity structure of the input. The code consists of `for`-loops, iterating over the variable space of the dependence relations and checking the dependence constraints. We describe the DAG using CSR format. `DAG.index` holds the index of neighbors of each node and `DAG.ptr` points to the start and end indices that indicate the list of neighbors of each node in `DAG.index`. Another auxiliary array that is used in the barrier-free strategy is `DAG.dep_count`, which stores the number of dependences that should be met for each node of the DAG. Hence, its values are calculated by counting the number of incoming edges to each node the DAG. Additionally, to support mapping multiple iterations per thread-block, a built-in call returns a coarsened graph. Listing 3 shows the code generated to extract both the baseline and coarsened DAGs for the `i1` loop in Listing 1.

Listing 3: DAG creation for SpTRSV CSC automatically generated for the code in Listing 1

```
def create_DAG(Ap, Ai, Ax, b, n, nwarp):
 DAG_list = [[] for _ in range(n)]
 for i1 in range(n):
  for j1 in range(Ap[i1] + 1, Ap[i1+1]):
   DAG_list[i1].append(Ai[j1])
 dep_count = [0 for i in range(n)]
 for neighs in DAG_list:
  for node in neighs:
   dep_count[node] += 1
 DAG = DAGBuilder(DAG_list, dep_count)
 c_DAG = coarsen(DAG, nwarp)
 return DAG.ptr, DAG.index, DAG.dep_count, c_DAG
```

Output Code Generation: Lines 8-14 of Algorithm 1 generate the output code based on the strategy. Two different block mapping strategies, barrier-free and wavefront implementations require different approaches for kernel launch and kernel code generation. The wavefront strategy synchronizes the thread-blocks on the host during kernel launches. The grid size is equal to the number of iterations in each wave.

Listing 4: Barrier-free SpTRSV Kernel

```
def barrier_free_sptrsv_csc_kernel(Ap, Ai, Ax, x, b
    , DAG_ptr, DAG_index, dep_count):
    col = cuda.blockIdx.x ## Block ID
    tid = cuda.threadIdx.x
    bdim = cuda.blockDim.x
    while tid == 0 and cuda.atomic.add(dep_count,
        col, 0) != 0: pass
    cuda.syncthreads()
    if tid == 0: x[col]=(b[col]-x[col])/Ax[Ap[col]]
    cuda.syncthreads()
    for j in range(Ap[col]+1+tid, Ap[col+1], bdim):
        cuda.atomic.add(x, Ai[j], Ax[j] * x[col])
    cuda.thread_fence()
    start, end = DAG_ptr[col], DAG_ptr[col + 1]
    for i in range(start + tid, end, bdim):
        val = DAG_index[i]
        cuda.atomic.add(dep_count, val, -1)
```

Lines 6–10 in Figure 2c display the `kernel_launch` function for barrier-free parallelization. A busy-waiting mechanism preserves dependences by monitoring the `dep_count` array; a thread-block proceeds only when its corresponding count reaches zero. After completing its primary computation, the block updates the DAG by decrementing the dependence count of its children. Listing 4 shows the generated kernel for Listing 1. `cuda.thread_fence()` is used to ensure that computed solution values are globally visible before a block signals completion via the dependence update.

C. Thread Manager

The thread manager lowers kernel code to enable thread-level and warp-level parallelism. Thread-level parallelism is achieved by mapping an iterator variable to GPU threads specified with the `thread_map` argument. Algorithm 2 details this process, taking the kernel AST, an unrolling factor UF, and the loop variable $loopvar$ as inputs. Line 1 identifies the AST subtree corresponding to $loopvar$. To assign specific iterations to each thread, the `thread_mapping` function applies a block-strided approach: it offsets the loop lower bound by the thread ID and increments by the block size B. Here, B corresponds to `blockDim.x` and the thread ID corresponds to `threadIdx.x`. To address potential memory coordination challenges, line 3 extracts loop dependences to determine if the mapped loop contains carried dependences.

Line 4 of Algorithm 2 identifies reduction patterns and generates a new AST using `__shfl_down_sync` for fast warp-level reductions. This leverages hardware-level shuffle operations to handle iterations with large trip counts through cooperative execution rather than independent lane processing. Also, when a decomposed task incorporates multiple iterations, each thread processes a row sequentially and dependence between them is handled at the warp-level.

Lines 5 and 6 isolate thread-invariant statements and apply control structures to limit their execution to a single thread. Since GPU kernels execute instructions across all threads, redundant execution of code outside a nested loop can cause race conditions. To prevent this, line 5 extracts thread-invariant

Algorithm 2: Thread Manager

Input : AST, UF, $loopvar$
Output: $iAST$

1 $tAST \leftarrow find\_loop(loopvar, AST)$
2 $tAST \leftarrow thread\_mapping(tAST)$
3 $\mathcal{C} \leftarrow extract\_deps(tAST)$
/* Parallel reduction */
4 **if** $\mathcal{C}$.`has_reduction()` **then**
add_parallel_reduction($iAST, \mathcal{C}$)
/* Thread invariant */
5 $rAST \leftarrow AST - tAST$
6 $rAST \leftarrow thread\_invariant(rAST)$
/* Unrolling */
7 $tAST \leftarrow unroll(tAST, UF)$
8 $iAST = rAST \cup tAST$

memory writes by excluding the $tAST$ from the input AST. Finally, line 7 unrolls the loop by factor UF, increasing the memory load per thread to reduce branch overhead.

D. Scheduler

We introduce an analytic scheduler to automatically select execution strategies and optimizations per matrix. The scheduler receives key characteristics extracted during the initial inspection that constructs the baseline dependence graph: wavefront levels L, nonzero count N_{nz}, average in-degree $\overline{\deg}_{in}$, and matrix dimension n, representing the number of rows. The scheduler evaluates candidate schedules (S, B, UF, n_{warp}), where $S \in \{\text{WF}, \text{BF}\}$ is the synchronization strategy, B is block size, UF is unrolling factor, and n_{warp} is task mapping granularity. Crucially, n_{warp} dictates the hardware mapping: $n_{warp} > 1$ yields standard hierarchical kernels mapped via UF, whereas $n_{warp} = 1$ triggers the intra-warp routing kernel.

To navigate this search space, the scheduler analytically models the execution time T:

$$T(S, B, UF, n_{warp}) = T_{launch}(S) + T_{sync}(S, n_{warp}) + T_{mem} + T_{comp}(B, UF) \quad (3)$$

The analytical parameters are divided into two categories. The first consists of architectural constants, which reflect fixed physical hardware specifications and microbenchmarks: kernel launch latency (λ_{launch}), atomic operation cost (γ_{atomic}), and hardware memory bandwidth (BW_{arch}). The second category comprises the empirical workload characteristics, such as the memory footprint (MEM_{usage}) and the total floating-point operations (FLOPs).

1) Launch Overhead and Synchronization Cost: The wavefront strategy requires L launches, with launch overhead $T_{launch}(\text{WF}) = L \cdot \lambda_{launch}$, but absorbs synchronization into kernel boundaries, resulting in a synchronization cost $T_{sync}(\text{WF}) = 0$. Conversely, the barrier-free strategy uses a single persistent launch where $T_{launch}(\text{BF}) = \lambda_{launch}$, but incurs atomic contention from data dependences $\overline{\deg}_{in}$ and global task dispatch over n rows. This yields a synchronization cost $T_{sync}(\text{BF}, n_{warp}) = \gamma_{atomic} \cdot \left(\overline{\deg}_{in} + \frac{n}{n_{warp}}\right)$.

2) Compute and Memory Time: The memory access time is modeled as $T_{mem} = MEM_{usage}/BW_{arch}$, capturing the baseline data movement cost. The compute time is $T_{comp}(B, UF) = \text{FLOPs}/\min(B \cdot UF, \overline{deg}_{in})$. This bounds the theoretical processing capacity ($B \cdot UF$) by the average available parallelism per row ($\overline{deg}_{in}$), preventing idle thread allocation. Furthermore, the bounded search space implicitly avoids hardware limits like excessive register demand. Since FLOPs are directly proportional to the nonzero count N_{nz}, the analytical costs simplify to evaluating these constrained resources against architectural limits.

To determine the optimal configuration, the scheduler evaluates this unified Equation 3 across the search space of available parameters. It sweeps combinations such as $S \in \{\text{WF}, \text{BF}\}$, $B \in \{32, 64, 128, 256\}$, $UF \in \{1, 2, 4, 8\}$, and $n_{warp} \geq 1$ to select the configuration that yields the minimal T.

IV. Implementation

The LCD transformation is implemented within a JIT compiler to enable adaptive code generation with changes in the sparsity pattern. Once the code is generated, the DAG creation and coarsening build the runtime order of iterations, compatible with the generated code. While JIT is used for this work, LCD can be integrated into compilers, but the recompilation should happen explicitly; otherwise, the code performs suboptimally. We discuss the implementation details of AST lowering in the JIT pipeline stages as well as supported DAG and block coarsening in LCD.

Listing 5: JIT Compiler

```
1 import ast
2 import inspect
3 from functools import wraps
4 from numba import cuda
5
6 def parallel(**kwargs):
7  def decorator(func):
8   block_loop_var = kwargs.get("block_map", None)
9   thread_loop_var = kwargs.get("thread_map", None)
10  source = inspect.getsource(func)
11  cur_AST = ast.parse(source)
12  def wrapper(*args, **kwargs):
13   cached = jit_cache(*args, **kwargs)
14   if not cached:
15    S,B,UF,nwarp=scheduler.schedule(cur_AST,*args)
16    dAST, hAST = block_manager(cur_AST, S,
          block_loop_var, nwarp)
17    dAST = thread_manager(dAST,UF,thread_loop_var)
18    preprocess = func_from_AST(hAST)
19    DAG = preprocess(*args)
20    kernel, launcher = cuda_compile(dAST, hAST,
          mode)
21    dev_args = transfer_to_device(*args, DAG, B)

22    result = launcher(kernel, dev_args)

23   else:
24    dev_args = transfer_to_device(*args, cached.DAG
          )
25    result = cached.launcher(cached.kernel,
          dev_args)
26   return transfer_to_host(result)

27  return wrapper
28 return decorator
```

JIT Compiler Implementation: The LCD transformation is implemented as a JIT compiler in Python. As shown in the import statements of Listing 5, the implementation leverages Python's built-in `ast` and `inspect` modules for parsing, `functools` for decorator creation, and the `numba` [18] library for CUDA code generation. To enable input-sensitive parallelism, LCD performs loop-carried dependence analysis and transformations prior to code generation. First, lines 10–11 extract and parse the source code of the target function into a Python AST. During execution, lines 15–17 apply a sequence of domain-specific lowering steps to this AST. Specifically, line 15 evaluates the AST alongside runtime matrix inputs via the `scheduler` to determine the proper synchronization strategy and hierarchical mapping parameters. Guided by these parameters, the `block_manager` and `thread_manager` in lines 16–17 transform the original AST into distinct host (`hAST`) and device (`dAST`) syntax trees. Lines 18–19 compile and invoke the host AST to construct the runtime dependence `DAG`. Next, the JIT pipeline executes the core device operations: line 20 compiles the device AST and launcher segment of the host AST into an optimized Numba CUDA callable; line 21 transfers the necessary execution arguments, including the `DAG` and scheduling metadata, from the CPU to the GPU; and line 22 launches the generated CUDA kernel on the device.

Furthermore, LCD is designed to accommodate dynamic sparsity situations where the nonzero structure changes across executions. When sparsity is dynamic, the optimal schedule changes, necessitating both re-inspection and new code generation. While traditional libraries typically only redo the inspection phase and reuse generic kernels due to their static technical limitations, LCD leverages its JIT compiler to generate new sparsity-specific code on the fly. To prevent this compilation overhead from occurring in every iteration when the sparsity remains static, a `cached` variable is evaluated in line 13. If the cache is valid, the framework bypasses the JIT pipeline and directly executes the previously generated kernel using the cached `DAG` in the `else` branch. Finally, line 26 transfers the computed result to host memory and returns it.

DAG Coarsening: As discussed in Section III-B, LCD supports coarsening strategies that map multiple iterations to a thread-block to mitigate synchronization overhead and improve hardware utilization. This section details three advanced approaches. First, coarsened wavefronts merge adjacent wavefronts. Second, iteration coarsening aggregates dependent rows into coarser tasks. Finally, block coarsening maps dense blocks to BLAS kernels using CUDA streams.

I: Wavefront Coarsening. To reduce synchronization overhead, the LCD framework supports coarsening adjacent wavefronts (Figure 3c) and a DAG coarsening strategy that aggregates dependent nodes into coarser sub-trees or tasks.

II: Iteration Coarsening. Tasks are constructed during runtime preprocessing using a window-based heuristic. To manage complexity, the DAG is partitioned into bands spanning multiple wavefront levels (Figure 4a). Within each band, dependent nodes are identified and merged to form discrete tasks. Kernel execution varies by strategy. The wavefront strategy employs a

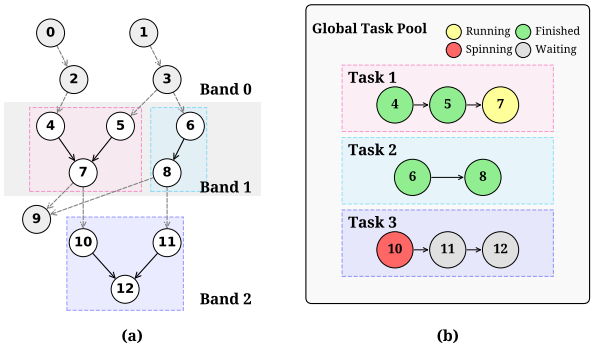


Fig. 4: (a) The DAG is partitioned into bands for iteration coarsening. Colored dashed boxes highlight how specific dependent nodes within these bands are grouped together to form discrete tasks. (b) Nodes in each task are flattened to enforce sequential execution, and all tasks are aggregated into a global task pool. Node colors indicate their real-time execution status: green for finished, yellow for actively running, red for actively waiting for dependences to resolve, and grey for waiting.

multi-launch method, issuing a separate kernel for each band. In the barrier-free strategy, bands dissolve into a unified global task pool (Figure 4b). Tasks are flattened into topologically sorted arrays, allowing persistent warps to fetch chains via atomic counters. This decouples execution from global barriers, enabling tasks to proceed across band boundaries once dependences are met.

III: Block Coarsening. Coarsening iterations into matrix blocks is common in supernodal sparse solvers to leverage BLAS operations. Since parallel BLAS calls within wavefront or barrier-free schemes are often inefficient due to the CPU-GPU communication typically required for each individual call, LCD leverages the CUDA Graphs API. By pre-defining the entire sequence of operations and their dependences, CUDA Graphs eliminate this per-call communication overhead, while CUDA Events act as lightweight synchronization points to enforce the correct execution order between different streams. To support this, LCD identifies BLAS calls in the AST, constructs a task-based DAG, and utilizes Algorithm 3 with the CUDA Graphs and CUDA Event APIs to manage execution. Algorithm 3 shows how different kernel launches are mapped to different streams. A DAG of tasks $\mathcal{T}$ and tuning parameter pool size or PS are passed as inputs, and schedule $\mathcal{S}$ is the output. First a stream pool is initialized to handle a set of streams. Then tasks in $\mathcal{T}$ are sorted topologically and stored in $\mathcal{O}$. After that, in line 3-7 of the Algorithm, for each task t in the $\mathcal{O}$ a stream is pulled from the pool. Then task t is submitted to schedule $\mathcal{S}$ by passing parent events followed by a signal as a dependence handle for t. While the focus of LCD is a single GPU, Algorithm 3 enables multi-GPU extension through the NCCL library. Communication tasks using `ncclSend` and `ncclRecv` are added to points of connection in the partitioned DAG, and are captured by CUDA Graphs during run-time.

Algorithm 3: Stream Manager

Input : $\mathcal{T}$,PS
Output: $\mathcal{S}$
1 $stream\_pool \leftarrow$ initialize(PS)
2 $\mathcal{O} \leftarrow$ topological_order$(\mathcal{T})$
3 **for** $t \in \mathcal{O}$ **do**
4 $\quad s \leftarrow stream\_pool$.next()
5 $\quad deps \leftarrow$ events$(t.parents)$
6 $\quad \mathcal{S}$.submit$(s, t, deps)$
7 $\quad \mathcal{S}$.signal(s, t)
8 **end**

V. Experimental Results

We evaluate the performance of the code generated by our LCD framework against their representative alternatives. The evaluation covers a range of sparse primitives and solvers.

A. Setup

1) Operations: Table I details the sparse matrix operations and storage formats evaluated in this work. We examine a suite of kernels strictly bottlenecked by complex, input-sensitive loop-carried dependences: forward and backward SpTRSV in both CSC and CSR formats, SpILU0 Factorization and its solve phase, Gauss-Seidel iterations, and Supernodal SpTRSV. These operations are covered because they form the computational backbone for linear solvers. As primary execution bottlenecks in numerical simulations, they serve as essential benchmarks for evaluating synchronization and parallel execution models on GPUs.

TABLE I: List of operations with formats and descriptions.

Operation	Format	Description
SpTRSV-fwd	CSC, CSR	Forward substitution
SpTRSV-bwd	CSC, CSR	Backward substitution
SpILU0	CSR	ILU0 factorization
SpILU0-solve	CSR	Solve with ILU0 factors
Gauss-Seidel	CSR	Gauss-Seidel iteration
SpTRSV-fwd	Supernodal CSC	Solve with Cholesky factors

2) Dataset: We evaluate all kernels using a benchmark of 139 SPD matrices from the SuiteSparse Matrix Collection, excluding cases that exhibit numerical instability or timeouts. This dataset provides a consistent evaluation foundation across all experiments. For SpTRSV, we extend this to a 232-matrix set, adding 93 large matrices with 10^6 to 10^7 nonzeros, to provide a more rigorous stress test against baselines such as CuPy [19], cuSPARSE [8], and YuenyeungSpTRSV [6]. For other operations, we maintain the 139-matrix dataset.

3) Environment: Experiments are conducted on server compute nodes equipped with NVIDIA H100 SXM GPUs (80 GB HBM3 memory) and AMD EPYC 9655 (Zen 5) host processors. For cross-architecture portability analysis, we extend the evaluation on a local workstation equipped with an NVIDIA GeForce RTX 4060 Ti GPU (8 GB GDDR6) and an Intel Core

i7-14700F processor. The software environment operates on Enterprise Linux 9 with NVIDIA Driver 570.17 and the CUDA 12.6 Toolkit. The proposed framework is implemented in Python 3.11.4 using Numba 0.61.2, which generates optimized PTX code via the NVVM backend.

4) Evaluation Methodology: We evaluate performance using single-precision floating-point arithmetic. Across all operations, CuPy 13.6.0 serves as our primary baseline, which often dispatches tasks to vendor libraries, such as cuSPARSE. For SpTRSV, we include direct comparisons with cuSPARSE, and the specialized YuenyeungSpTRSV implementation, excluding their inspection time. Benchmarking native C++ cuSPARSE against its Python wrapper showed a median overhead of 3.84% (gmean 6.3%), confirming the Python API overhead is negligible for these throughput-bound kernels. All measurements are conducted via the Triton [20] framework, reporting median throughput after a warm-up phase for reproducibility.

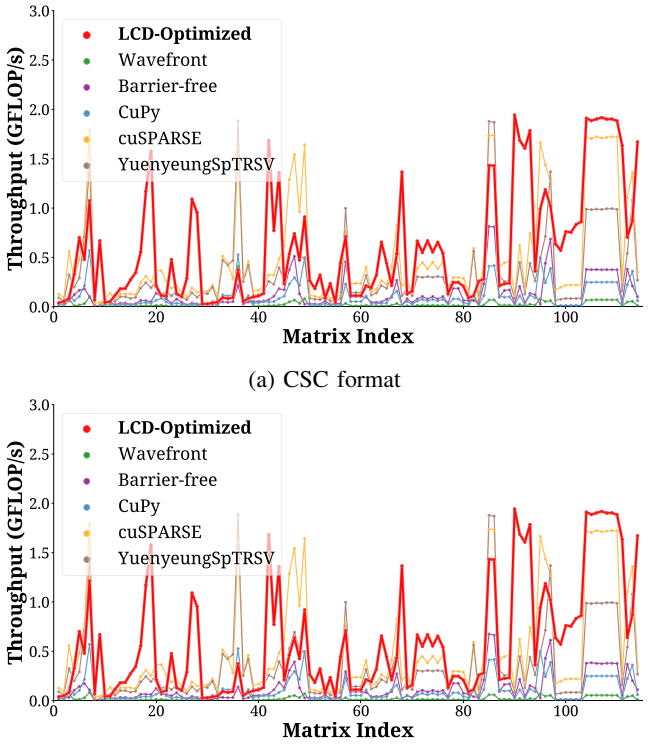
B. Kernel-Level Performance Analysis

1) Sparse Triangular Solve: In this section, we analyze the performance of the generated kernels for SpTRSV across both CSC and CSR formats. The evaluation focuses on throughput in GFLOP/s and relative speedup against cuSPARSE, YuenyeungSpTRSV, CuPy, and standard wavefront and barrier-free baselines.

For the CSC format, LCD-optimized kernels achieve a median throughput of 1.686 GFLOP/s, delivering gmean speedups of 1.62×, 2.73×, and 10.33× over cuSPARSE, YuenyeungSpTRSV, and CuPy, respectively. As shown in Fig. 5a, LCD outperforms these baselines on 57.8%, 68.2%, and 94.2% of the test cases. The performance gap is even starker when compared to basic structural baselines, with LCD achieving 37.67× and 10.15× speedups over standard wavefront and barrier-free variants. This disparity highlights that simply eliminating global barriers is insufficient for the column-centric updates inherent to CSC; rather, LCD's hierarchical task mapping is crucial to exploit fine-grained intra-column parallelism while effectively keeping accumulated updates localized in fast registers.

Similarly, for the CSR format, LCD achieves a median throughput of 1.684 GFLOP/s, yielding gmean speedups of 1.60×, 2.69×, and 10.16× over cuSPARSE, YuenyeungSpTRSV, and CuPy, respectively. As depicted in Fig. 5b, LCD outperforms these libraries on 57.8%, 66.8%, and 93.3% of the matrices. Against the generated structural baselines, LCD delivers massive 44.86× and 10.28× speedups over standard wavefront and barrier-free implementations. Building on the CSC results, this gap highlights the performance penalty of unmanaged atomic contention and load imbalance across varying row lengths, which LCD mitigates through task coarsening.

Taken together, while cuSPARSE remains the most competitive baseline library, LCD outperforms it in a majority of cases through matrix-specific kernel specialization. The substantial performance gaps between LCD and the remaining baselines highlight a critical insight: exposing the underlying dependence structure and load imbalance directly to the code generator unlocks automated optimization opportunities that static libraries inherently struggle to exploit.



(a) CSC format

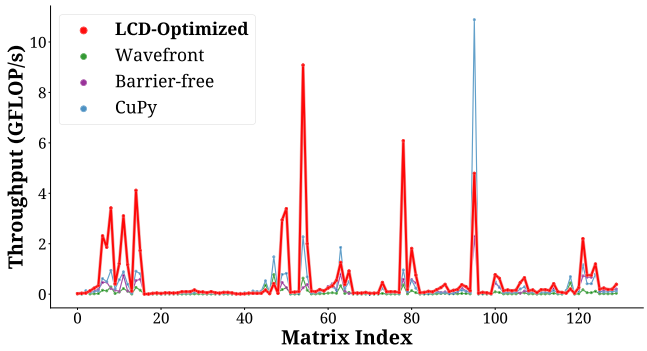
(b) CSR format

Fig. 5: Performance distribution for SpTRSV forward passes in CSC and CSR formats. The trend of SpTRSV backward pass performance remains similar. The plot is restricted to the 0–3 GFLOP/s range to preserve detail visibility.

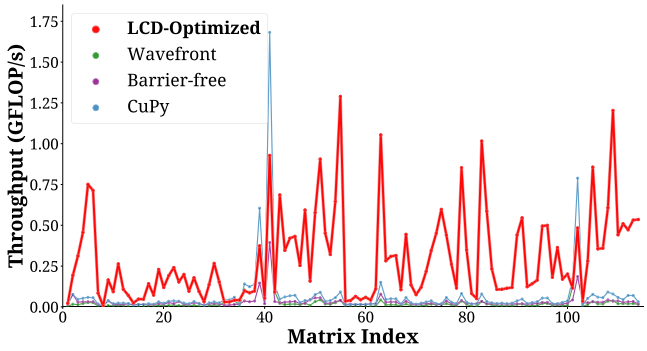
2) Sparse Incomplete LU Preconditioning: LCD optimizations are then evaluated across the preconditioning workflow including both factorization and the subsequent triangular solve. We first assess sparse incomplete LU factorization with zero fill-in (SpILU0). Our evaluation focuses on the CSR format, as row-wise elimination inherently requires row-major storage to ensure optimal data locality during updates.

The results demonstrate that the LCD-optimized kernels achieve a median throughput of 0.149 GFLOP/s, delivering a 2.81× gmean speedup over CuPy and outperforming it on 89.8% of the test cases (Fig. 6a). Unlike triangular solves, factorization requires repeated matrix accesses to accumulate partial results. Basic parallelization lacks locality awareness, forcing these frequent updates through the global memory hierarchy and causing immediate bandwidth saturation, yielding a mere 0.002 GFLOP/s. In contrast, LCD automates thread-level unrolling and tiling to promote this data reuse within the register file. By converting expensive global memory transactions into fast local operations, LCD robustly manages the complex dependence chains inherent to factorization.

In the subsequent triangular solve using the generated ILU0 factors, LCD sustains a robust median throughput of 0.192 GFLOP/s, delivering a 4.56× gmean speedup over



(a) ILU0 Factorization



(b) Solve with ILU0 factors

Fig. 6: Performance distribution for SpILU0 factorization and the subsequent solves with the factors in CSR format.

CuPy (Fig. 6b). Notably, CuPy's throughput degrades from 0.063 GFLOP/s on raw matrices to just 0.034 GFLOP/s on the L-factors. This performance drop exposes the structural challenge of preconditioned matrices, which frequently feature deep, serialized dependence chains typical of discretized PDEs. While the baseline struggles with the resulting synchronization latency, LCD effectively mitigates this bottleneck. By employing DAG-coarsening, the generated kernels successfully mask the dependence latency and maintain high throughput across structurally challenging scenarios.

3) Gauss-Seidel Method: We also evaluate the Gauss-Seidel method, an iterative solver where parallelism is constrained by sparse structural dependences. LCD achieves a median throughput of 0.184 GFLOP/s, yielding a 3.09× gmean speedup over CuPy (0.059 GFLOP/s) and maintaining a consistent advantage across most matrices (Fig. 7).

Comparing basic parallelization strategies reveals a distinct behavior for this operation. The standard wavefront method performs poorly (0.003 GFLOP/s), proving that global synchronization barriers are disastrous for repeated, fine-grained iterative processes. Conversely, the basic barrier-free implementation achieves a reasonable 2.27× speedup over CuPy. However, LCD's 3.09× speedup demonstrates that even when simple sync-free methods are viable, integrating hierarchical task mapping provides a gain by further localizing memory accesses and optimizing thread-level contention.

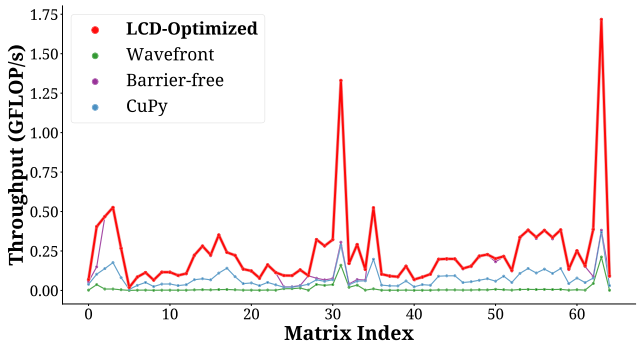


Fig. 7: Performance distribution for the Gauss-Seidel method. The plot is restricted to the 0–2 GFLOP/s range to preserve detail visibility.

C. Ablation Study

1) Scheduler Efficacy: To evaluate the effectiveness of the proposed scheduler, we compare the performance of kernels using the scheduler-selected parameters against an oracle selection, defined as choosing the configuration that achieves the best observed performance among all evaluated candidates for each matrix. This scheduler analysis is reported on the common 139-matrix dataset and uses cuSPARSE as the baseline. As illustrated in Figure 8, the performance profile of the LCD-Optimized kernels overlays the theoretical upper bound across the majority of the dataset. For SpTRSV forward pass in CSC, the scheduler is effective at navigating the trade-off between kernel launch overhead and serialization by achieving a gmean speedup of 2.36× over cuSPARSE, closely capturing the theoretical upper bound at 2.72× and matching 69.2% of the oracle choices. Similarly, SpTRSV in CSR, the scheduler delivers a 1.87× speedup, close to the theoretical maximum of 2.11×, matching 66.9% of the oracle choices. The same trend has been observed for other operations.

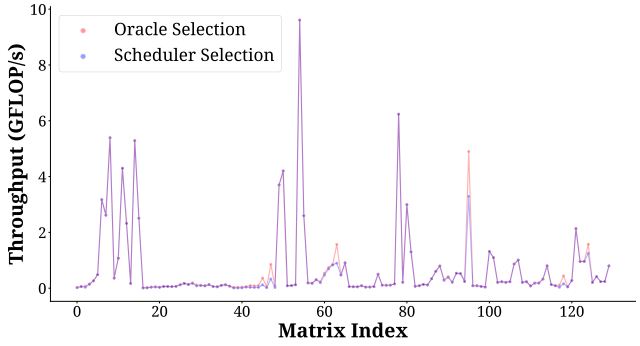


Fig. 8: Performance comparison for CSC SpTRSV between the scheduler-selected LCD-optimized kernels and the oracle-selected configuration.

To further validate the scheduler's analytical model, we correlated the modeled and actual performance rankings of 59 Gauss-Seidel kernel variants, which serves as a stress test for operations with complex loop-carried dependences. By

comparing percentiles derived from scheduler-predicted execution times against experimental throughput measurements, we observed a moderate positive correlation with Pearson $R = 0.676$ and $p < 10^{-8}$. This confirms that the analytical model successfully preserves the relative ordering of execution strategies, empowering the scheduler to reliably isolate high-performing configurations from suboptimal ones.

2) CUDA Graphs: As discussed in Section IV and Algorithm 3, we implemented SpTRSV Supernodal CSC using CUDA Graphs, where the Task Handler assigns a dense lower triangular solver followed by a dense matrix-vector multiplication. To evaluate its performance, we compared it against the cuSPARSE SpTRSV CSR transposed implementation. As shown in Figure 9, the LCD-optimized kernels achieve a gmean speedup of 1.70× over the baseline on matrices with supernodal structures. Specifically, the median throughput improves from 1.93 GFLOP/s with cuSPARSE to 4.94 GFLOP/s with the LCD-optimized implementation, demonstrating the efficacy of LCD's stream manager for supernodal solvers.

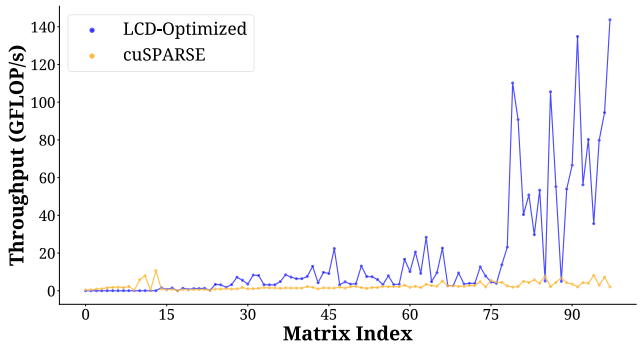


Fig. 9: Performance of supernodal CSC SpTRSV-fwd in LCD-optimized implementation against cuSPARSE on Cholesky factor of SPD matrices.

3) JIT Runtime Overhead: The LCD framework introduces runtime preprocessing overhead through AST lowering, DAG creation, and optional coarsening. AST lowering overhead is constant for each operation, whereas DAG creation and coarsening depend on the input matrix. In this experiment, we quantify the host-side preprocessing overhead and determine the break-even point, defined as the number of SpTRSV solves required for the LCD-optimized code to outperform its corresponding sequential baseline when including preprocessing costs:

$$n_{\text{break-even}} = \frac{T_{\text{overhead}}}{T_{\text{baseline}} - T_{\text{optimized}}} \tag{4}$$

where n is the number of solves, T_{overhead} is the one-time preprocessing time, and T_{baseline} and $T_{\text{optimized}}$ are the per-solve times for the baseline and optimized kernels, respectively.

Based on the full dataset, the preprocessing overhead is amortized very quickly for both sparse formats. The CSC format requires a median of 1.22 solves to break even, with a gmean of 1.29, while the CSR format requires a median of 1.15 solves, with a gmean of 1.17. Nearly half of the matrices break even within a single solve, specifically 45.3% for CSC and 45.7% for CSR, and more than three quarters break even within four solves, namely 75.3% for CSC and 78.0% for CSR. These results indicate that rapid amortization is the common case across the benchmark set, and that the difference between CSC and CSR preprocessing overhead is small in practice.

Consequently, for applications requiring repeated triangular solves, such as iterative linear solvers, time-stepping methods, or nonlinear optimization, the host-side preprocessing overhead becomes effectively negligible relative to the total execution time. These results confirm that the LCD preprocessing stage is rapidly amortized in realistic workloads.

4) Cross-Architecture Portability: To assess the portability of the LCD framework across varying hardware capabilities, we extended our evaluation to the NVIDIA RTX 4060Ti (Ada Lovelace). Unlike the data-center-grade H100, this consumer-grade architecture possesses different memory bandwidth and atomic operation characteristics. Table II summarizes the CSC SpTRSV performance across both architectures, evaluated on a mutually feasible subset of matrices.

The LCD-optimized kernels demonstrate consistent performance, outperforming the vendor library on both platforms. On H100, the framework achieves gmean speedups of 2.36× and 1.87× for CSC and CSR respectively. On RTX 4060Ti, it maintains this advantage with speedups of 2.94× and 1.76×, confirming that the LCD framework effectively retargets optimization configurations to match the specific resource constraints of the underlying hardware.

As detailed for block coarsening in Section IV, LCD can be extended to multi-GPU configurations by enhancing the stream manager algorithm. We performed a preliminary evaluation on a 4-GPU A100 node and observed promising scalability on a subset of matrices. The gmean speedup over the single-GPU case reaches 1.62× on 2 GPUs and 2.46× on 4 GPUs, suggesting that LCD has the potential to benefit from multi-GPU execution, with a matrix-dependent degree of scaling.

TABLE II: Cross-architecture performance comparison of SpTRSV on NVIDIA H100 (Hopper) and RTX 4060Ti (Ada) using the CSC Format over the subset of SuiteSparse matrices.

	H100 (Hopper)		RTX 4060Ti (Ada)	
Method	**GFLOP/s**	**Speedup**	**GFLOP/s**	**Speedup**
cuSPARSE	0.063	1.00×	0.088	1.00×
Wavefront	0.016	0.26×	0.025	0.26×
Barrier-free	0.080	1.08×	0.074	0.71×
LCD-Optimized	**0.176**	**2.36×**	**0.321**	**2.94×**

D. Preconditioned Iterative Solver Case Study

To evaluate the practical impact of the LCD framework on iterative solvers, we integrated the generated kernels for the Gauss-Seidel forward and backward passes into a Preconditioned Conjugate Gradient (PCG) solver, employing Symmetric Gauss-Seidel (SymGS) as the preconditioner. SymGS imposes significant challenges on parallel architectures due to the strict row-wise loop-carried dependences inherent in

its forward and backward sweeps. We compared the end-to-end execution time of the PCG solver utilizing these LCD-optimized kernels against it using baseline kernels based on CuPy, backed by cuSPARSE, on the NVIDIA H100.

The LCD-optimized solver shows that the micro-benchmark advantages translate to full application acceleration. As demonstrated in Figure 10, across 75 converging matrices, the PCG solver invoking LCD-optimized kernels achieves a gmean speedup of 1.73× over the CuPy baseline, outperforming it in 81.3% of the test cases. Additionally, the dataset analysis reveals a mean of 180 iterations for convergence, with a median of 57. Notably, 30.7% of the matrices require between 200 and 1000 iterations. In these computationally intensive regimes, the one-time compiler preprocessing overhead becomes negligible, rapidly amortized over hundreds of kernel invocations, confirming the overhead analysis in Section V-C3.

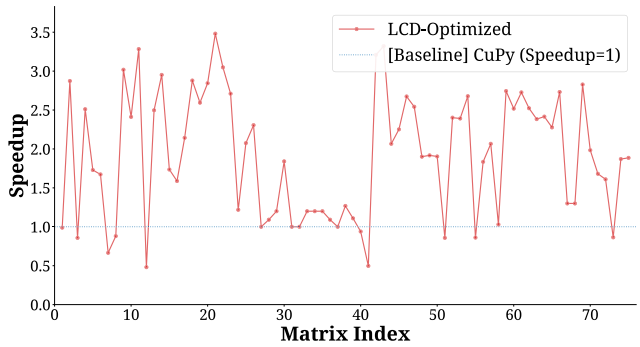


Fig. 10: End-to-end SymGS PCG solver speedup of LCD-optimized kernels over CuPy across converging matrices.

VI. Limitations

LCD assumes that loop-carried dependences expose partial parallelism, as seen in sparse solvers. In applications like Infinite Impulse Response [21], dense solvers [22], [23], and heat distribution simulations [24], iterations depend densely on previous ones, offering limited parallelism. In these cases, while LCD remains applicable, the block manager provides no performance gains. However, thread manager optimizations may still be beneficial and warrant further investigation. Furthermore, regarding the scope of dependence analysis, LCD supports the control-flow and data-access patterns present in the evaluated kernels. This includes nested conditionals, indirect writes, multiple right-hand side (RHS) vectors, and multiple sparse arrays. However, the current transformation framework does not support loops containing conditional increments or array aliasing.

VII. Related work

Libraries. Sparse computation libraries for SpTRSV, SpILU0, and other numerical methods rely on two task interaction techniques: wavefront parallelism [3], [4], [25], [26] and barrier-free methods [27]. Furthermore, recent approaches for multi-core CPUs have introduced stale-synchronous-parallel execution models to actively overlap synchronization with computation [28]. While these are efficient on CPUs, GPU implementations [5]–[7], [9], [29], [30] favor barrier-free methods to reduce kernel launches. Task decomposition also varies; CPU methods [25], [31] use coarsening for locality, whereas GPUs require managing a complex thread hierarchy. GPU libraries either employ fine-grained decomposition, assigning rows to thread-blocks [7], [29], or aggregate rows and subtrees [32]. Supernodal solvers [33] further utilize coarsening by batching BLAS kernels within wavefronts. Other operations like ILU0 [34], [35] or Gauss-Seidel [36] use similar library-based techniques. However, these libraries are often specialized for specific matrices or methods, making a universal, maintainable solution difficult to achieve.

Compilers for CPU. RECUMA [37] and REPTILE [10] tensor compilers handle recurrences and tiling but primarily target single-threaded CPUs. For parallelism, inspector-executor methods [13], [15] utilize wavefront strategies, often leveraging sparse polyhedral frameworks and matrix properties like monotonicity [17] to optimize indirect memory accesses. Sparsity-driven techniques like Sympiler [11], ParSy [15], and Partially Strided Codelets [38] exploit memory access traces to build DAGs for vectorization, coarsening, and fusion [39]–[41]. While effective for CPU thread-level parallelism, these approaches often underutilize GPUs due to host-to-device overhead or rigid parallelism models. LCD addresses these limitations by providing wavefront, barrier-free, and diverse coarsening strategies tailored for the GPU thread hierarchy.

Compilers for GPU. There are several compilers for parallel loops, such as those used in tensor multiplication [20], [42]–[44]. However, for loop-carried dependences, these techniques are not applicable. Prior GPU methods have designed compilers to predict dependences [45], [46] for statically non-deterministic data. In contrast, dependence relations in loops within sparse solvers are often statically predictable since they stem from the sparsity pattern of the matrices. LCD is the first compiler that addresses parallelism for loop-carried dependences in solvers.

VIII. Conclusion

Leveraging partial parallelism in loop-carried dependence is critical for sparse matrix operations on GPUs. Existing libraries employ strategies, such as wavefront and barrier-free methods with diverse task decomposition, but lack adaptability, as performance varies significantly with matrix sparsity. We propose LCD, a JIT compiler that generates adaptive GPU code by mapping loop iterations to thread-blocks and threads while ensuring correctness through targeted synchronization. LCD adapts to input sparsity at runtime, delivering a gmean speedup of 1.60×-4.56× over cuSPARSE on H100 GPUs.

Acknowledgments

This work is supported by NSERC discovery grants (RGPIN-2023-04897, DGECR-2023-00133), NSERC alliance grant (ALLRP 586319-23), National Science Foundation Award (CCF-2107556), and the Digital Research Alliance of Canada (alliancecan.ca).

REFERENCES

[1] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed. Society for Industrial and Applied Mathematics, 2003. [Online]. Available: https://doi.org/10.1137/1.9780898718003

[2] R. Allen and K. Kennedy, *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001. [Online]. Available: https://dl.acm.org/doi/book/10.5555/502981

[3] J. Anantpur and R. Govindarajan, "Runtime dependence computation and execution of loops on heterogeneous systems," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Feb 2013, pp. 1–10. [Online]. Available: https://doi.org/10.1109/CGO.2013.6494992

[4] E. Anderson and Y. Saad, "Solving sparse triangular linear systems on parallel computers," *Int. J. High Speed Comput.*, vol. 1, no. 1, p. 73–95, Apr. 1989. [Online]. Available: https://doi.org/10.1142/S0129053389000056

[5] W. Liu, A. Li, J. Hogg, I. S. Duff, and B. Vinter, "A synchronization-free algorithm for parallel sparse triangular solves," in *Euro-Par 2016: Parallel Processing*, P.-F. Dutot and D. Trystram, Eds. Cham: Springer International Publishing, 2016, pp. 617–630. [Online]. Available: https://doi.org/10.1007/978-3-319-43659-3_45

[6] F. Zhang, J. Su, W. Liu, B. He, R. Wu, X. Du, and R. Wang, "Yuenyeungsptrsv: A thread-level and warp-level fusion synchronization-free sparse triangular solve," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 9, pp. 2321–2337, 2021. [Online]. Available: https://doi.org/10.1109/TPDS.2021.3066635

[7] J. Su, F. Zhang, W. Liu, B. He, R. Wu, X. Du, and R. Wang, "Capellinisptrsv: A thread-level synchronization-free sparse triangular solve on gpus," in *Proceedings of the 49th International Conference on Parallel Processing*, ser. ICPP '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3404397.3404400

[8] NVIDIA Corporation, *cuSPARSE Library*, 2026, accessed: 2026-04-07. [Online]. Available: https://docs.nvidia.com/cuda/cusparse/

[9] K. Rupp, P. Tillet, F. Rudolf, J. Weinbub, A. Morhammer, T. Grasser, A. Jüngel, and S. Selberherr, "Viennacl—linear algebra library for multi- and many-core architectures," *SIAM J. Sci. Comput.*, vol. 38, no. 5, p. S412–S439, Jan. 2016. [Online]. Available: https://doi.org/10.1137/15M1026419

[10] M. U. Tariq, S. Sundram, and F. Kjolstad, "Reptile: Performant tiling of recurrences," *Proc. ACM Program. Lang.*, vol. 9, no. OOPSLA2, Oct. 2025. [Online]. Available: https://doi.org/10.1145/3763074

[11] K. Cheshmi, S. Kamil, M. M. Strout, and M. M. Dehnavi, "Sympiler: transforming sparse matrix codes by decoupling symbolic analysis," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3126908.3126936

[12] K. Cheshmi, "Transforming sparse matrix computations," Ph.D. dissertation, University of Toronto (Canada), 2022.

[13] M. M. Strout, M. Hall, and C. Olschanowsky, "The sparse polyhedral framework: Composing compiler-generated inspector-executor code," *Proceedings of the IEEE*, vol. 106, no. 11, pp. 1921–1934, 2018. [Online]. Available: https://doi.org/10.1109/JPROC.2018.2857721

[14] A. Venkat, M. S. Mohammadi, J. Park, H. Rong, R. Barik, M. M. Strout, and M. Hall, "Automating wavefront parallelization for sparse matrix computations," in *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016, pp. 480–491. [Online]. Available: https://doi.org/10.1109/SC.2016.40

[15] K. Cheshmi, S. Kamil, M. M. Strout, and M. M. Dehnavi, "Parsy: inspection and transformation of sparse matrix computations for parallelism," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '18. IEEE Press, 2019. [Online]. Available: https://doi.org/10.1109/SC.2018.00065

[16] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, Dec. 2011. [Online]. Available: https://doi.org/10.1145/2049662.2049663

[17] M. S. Mohammadi, T. Yuki, K. Cheshmi, E. C. Davis, M. Hall, M. M. Dehnavi, P. Nandy, C. Olschanowsky, A. Venkat, and M. M. Strout, "Sparse computation data dependence simplification for efficient compiler-generated inspectors," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 594–609. [Online]. Available: https://doi.org/10.1145/3314221.3314646

[18] S. K. Lam, A. Pitrou, and S. Seibert, "Numba: a llvm-based python jit compiler," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, ser. LLVM '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2833157.2833162

[19] R. Okuta, Y. Unno, D. Nishino, S. Hido, and C. Loomis, "Cupy: A numpy-compatible library for nvidia gpu calculations," in *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*, 2017. [Online]. Available: http://learningsys.org/nips17/assets/papers/paper_16.pdf

[20] P. Tillet, H. T. Kung, and D. Cox, "Triton: an intermediate language and compiler for tiled neural network computations," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, ser. MAPL 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 10–19. [Online]. Available: https://doi.org/10.1145/3315508.3329973

[21] G. Chaurasia, J. Ragan-Kelley, S. Paris, G. Drettakis, and F. Durand, "Compiling high performance recursive filters," in *Proceedings of the 7th Conference on High-Performance Graphics*, ser. HPG '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 85–94. [Online]. Available: https://doi.org/10.1145/2790060.2790063

[22] J. Dongarra, M. Abalenkovs, A. Abdelfattah, M. Gates, A. Haidar, J. Kurzak, P. Luszczek, S. Tomov, I. Yamazaki, and A. YarKhan, "Parallel programming models for dense linear algebra on heterogeneous systems," *Supercomputing Frontiers and Innovations*, vol. 2, no. 4, p. 67–86, Mar. 2016. [Online]. Available: https://superfri.org/index.php/superfri/article/view/90

[23] K. A. Gallivan, R. J. Plemmons, and A. H. Sameh, "Parallel algorithms for dense linear algebra computations," *SIAM Review*, vol. 32, no. 1, pp. 54–135, 1990. [Online]. Available: https://doi.org/10.1137/1032002

[24] Y. Jaluria, *Computational Heat Transfer*. Routledge, 2003. [Online]. Available: https://doi.org/10.1201/9781315140018

[25] B. Zarebavani, K. Cheshmi, B. Liu, M. M. Strout, and M. M. Dehnavi, "Hdagg: Hybrid aggregation of loop-carried dependence iterations in sparse matrix computations," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2022, pp. 1217–1227. [Online]. Available: https://doi.org/10.1109/IPDPS53621.2022.00121

[26] D. Ma, K. Ahmad, K. Cheshmi, H. Sundar, and M. Hall, "Sparsified preconditioned conjugate gradient solver on gpus," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 602–616. [Online]. Available: https://doi.org/10.1145/3712285.3759796

[27] J. Park, M. Smelyanskiy, N. Sundaram, and P. Dubey, "Sparsifying synchronization for high-performance shared-memory sparse triangular solver," in *Proceedings of the 29th International Conference on Supercomputing - Volume 8488*, ser. ISC 2014. Berlin, Heidelberg: Springer-Verlag, 2014, p. 124–140. [Online]. Available: https://doi.org/10.1007/978-3-319-07518-1_8

[28] R. S. Steiner, C. K. Matzoros, P. A. Papp, T. Böhnlein, and A. N. Yzelman, "Elasticity in parallel sparse triangular solve," 2026. [Online]. Available: https://arxiv.org/abs/2607.02324

[29] M. Naumov, "Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the gpu," NVIDIA Corporation, Tech. Rep. NVR-2011-001, 2011. [Online]. Available: https://research.nvidia.com/sites/default/files/pubs/2011-06_Parallel-Solution-of/nvr-2011-001.pdf

[30] K. Tuteja, G. Olenik, R. Mishchuk, Y.-H. Tsai, M. Götz, A. Streit, H. Anzt, and C. Debus, "pyginkgo: A sparse linear algebra operator framework for python," in *Proceedings of the 54th International Conference on Parallel Processing*, ser. ICPP '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 753–763. [Online]. Available: https://doi.org/10.1145/3754598.3754648

[31] B. Yılmaz, B. Sipahioğrlu, N. Ahmad, and D. Unat, "Adaptive level binning: A new algorithm for solving sparse triangular systems," in *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, ser. HPCAsia '20. New York,

NY, USA: Association for Computing Machinery, 2020, p. 188–198. [Online]. Available: https://doi.org/10.1145/3368474.3368486

[32] Z. Hu, J. Sun, Z. Li, and G. Sun, "Ag-sptrsv: An automatic framework to optimize sparse triangular solve on gpus," *ACM Trans. Archit. Code Optim.*, vol. 21, no. 4, Nov. 2024. [Online]. Available: https://doi.org/10.1145/3674911

[33] I. Yamazaki, S. Rajamanickam, and N. Ellingwood, "Performance portable supernode-based sparse triangular solver for manycore architectures," in *Proceedings of the 49th International Conference on Parallel Processing*, ser. ICPP '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3404397.3404428

[34] H. Anzt, T. Ribizel, G. Flegar, E. Chow, and J. Dongarra, "Parilut - a parallel threshold ilu for gpus," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019, pp. 231–241. [Online]. Available: https://doi.org/10.1109/IPDPS.2019.00033

[35] Y. Chen, X. Tian, H. Liu, Z. Chen, B. Yang, W. Liao, P. Zhang, R. He, and M. Yang, "Parallel ilu preconditioners in gpu computation," *Soft Comput.*, vol. 22, no. 24, p. 8187–8205, Dec. 2018. [Online]. Available: https://doi.org/10.1007/s00500-017-2764-7

[36] A. Ahmadi, F. Manganiello, A. Khademi, and M. C. Smith, "A parallel jacobi-embedded gauss-seidel method," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 6, pp. 1452–1464, 2021. [Online]. Available: https://doi.org/10.1109/TPDS.2021.3052091

[37] S. Sundram, M. U. Tariq, and F. Kjolstad, "Compiling recurrences over dense and sparse arrays," *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA1, Apr. 2024. [Online]. Available: https://doi.org/10.1145/3649820

[38] K. Cheshmi, Z. Cetinic, and M. M. Dehnavi, "Vectorizing sparse matrix computations with partially-strided codelets," in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2022, pp. 1–15. [Online]. Available: https://doi.org/10.1109/SC41404.2022.00037

[39] M. M. Salehi and K. Cheshmi, "Loop fusion in matrix multiplications with sparse dependence," in *Proceedings of the 39th ACM International Conference on Supercomputing*, ser. ICS '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 625–639. [Online]. Available: https://doi.org/10.1145/3721145.3730427

[40] K. Cheshmi, M. Strout, and M. Mehri Dehnavi, "Runtime composition of iterations for fusing loop-carried sparse dependence," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3581784.3607097

[41] K. Cheshmi, M. M. Strout, and M. M. Dehnavi, "Optimizing sparse computations jointly," in *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 459–460. [Online]. Available: https://doi.org/10.1145/3503221.3508439

[42] A. Venkat, M. Hall, and M. Strout, "Loop and data transformations for sparse matrix code," *SIGPLAN Not.*, vol. 50, no. 6, p. 521–532, Jun. 2015. [Online]. Available: https://doi.org/10.1145/2813885.2738003

[43] C. Li, Y. Xu, S. M. Saravani, and P. Sadayappan, "Accelerated auto-tuning of gpu kernels for tensor computations," in *Proceedings of the 38th ACM International Conference on Supercomputing*, ser. ICS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 549–561. [Online]. Available: https://doi.org/10.1145/3650200.3656626

[44] P. Stpiczyński, "Semiautomatic acceleration of sparse matrix-vector product using openacc," in *Parallel Processing and Applied Mathematics*, R. Wyrzykowski, E. Deelman, J. Dongarra, K. Karczewski, J. Kitowski, and K. Wiatr, Eds. Cham: Springer International Publishing, 2016, pp. 143–152. [Online]. Available: https://doi.org/10.1007/978-3-319-32152-3_14

[45] H. Liu, K. T. Lam, H. Lin, C.-L. Wang, and J. Ma, "Lightweight dependency checking for parallelizing loops with non-deterministic dependency on gpu," in *2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS)*, 2016, pp. 884–893. [Online]. Available: https://doi.org/10.1109/ICPADS.2016.0119

[46] M. Samadi, A. Hormati, J. Lee, and S. Mahlke, "Leveraging gpus using cooperative loop speculation," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 1, Feb. 2014. [Online]. Available: https://doi.org/10.1145/2579617