Runtime Composition of Iterations for Fusing Loop-carried Sparse Dependence

Kazem Cheshmi∗
McMaster University
Hamilton, Canada
cheshmi@mcmaster.ca

Michelle Mills Strout
University of Arizona and HPE
Tucson, USA
mstrout@cs.arizona.edu

Maryam Mehri Dehnavi
University of Toronto
Toronto, Canada
mmehride@cs.toronto.edu

ABSTRACT
Dependence between iterations in sparse computations causes inefficient use of memory and computation resources. This paper proposes sparse fusion, a technique that generates efficient parallel code for the combination of two sparse matrix kernels, where at least one of the kernels has loop-carried dependencies. Existing implementations optimize individual sparse kernels separately. However, this approach leads to synchronization overheads and load imbalance due to the irregular dependence patterns of sparse kernels, as well as inefficient cache usage due to their irregular memory access patterns. Sparse fusion uses a novel inspection strategy and code transformation to generate parallel fused code optimized for data locality and load balance. Sparse fusion outperforms the best of unfused implementations using ParSy and MKL by an average of 4.2× and is faster than the best of fused implementations using existing scheduling algorithms, such as LBC, DAGP, and wavefront by an average of 4× for various kernel combinations.

CCS CONCEPTS
• Software and its engineering → Runtime environments.

1 INTRODUCTION
Numerical algorithms [32] and optimization methods [5, 11, 36] often involve numerous consecutive sparse matrix computations. For example, in iterative solvers [32] such as Krylov methods [13, 33], sparse kernels that apply a preconditioner are repeatedly executed inside and between iterations of the solver. Sparse kernels with loop-carried dependencies, i.e., kernels with partial parallelism, are frequently used in numerical algorithms, and the performance of scientific simulations relies heavily on efficient parallel implementations of these computations. Sparse kernels that exhibit partial parallelism often have multiple wavefronts of parallel computation where a synchronization is required for each wavefront, i.e., wavefront parallelism [19, 44]. The amount of parallelism varies per wavefront and often tapers off toward the end of the computation, which results in load imbalance. Figure 1 shows with dark lines the nonuniform parallelism for the sparse incomplete Cholesky (SpIC0) and the sparse triangular solve (SpTRSV) kernels when SpTRSV executes after SpIC0 completes. Separately optimizing such kernels exacerbates this problem by adding even more synchronization. Also, opportunities for data reuse between two sparse computations might not be realized when sparse kernels are optimized separately.

Figure 1: The nonuniform parallelism in the DAGs of sparse incomplete Cholesky and triangular solver (annotated with unfused) and for the joint DAG of the two kernels results in load imbalance. Higher value in the y-axis shows high parallelism in a given wavefront. Wavefront numbers in the x-axis are numbered based on their order of execution.
Instead of iterations of sparse kernels being scheduled separately, they can be scheduled jointly. Wavefront parallelism can be applied to the joint DAG of two sparse computations. A data flow directed acyclic graph (DAG) describes the dependencies between iterations of a kernel [9, 22, 40]. A joint DAG includes all of the dependencies between iterations within and across kernels. The joint DAG of sparse kernels with partial parallelism and the DAG of another sparse kernel provides slightly more parallelism per wavefront without increasing the number of wavefronts. The yellow line in Figure 1 shows how scheduling the joint DAG of SpIC0 and SpTRSV increases the parallelism per wavefront and significantly reduces the number of wavefronts (synchronizations). However, the load balance issues remain, and there are still several synchronizations.

Wavefronts of the joint DAG can be aggregated to reduce the number of synchronizations. However, existing DAG partitioners such as Load-Balanced Level Coarsening (LBC) [10] and DAGP [23] may not achieve good load balance when applied to the joint DAG, because they aggregate iterations from consecutive wavefronts, which may have different amounts of parallelism. Moreover, by aggregating iterations from wavefronts in the joint DAG, DAG partitioning methods may improve the temporal locality between the two kernels, but this can compromise the spatial locality within each kernel. For example, for two sparse kernels that only share a small array and operate on different sparse matrices, optimizing temporal locality between kernels will not be beneficial. Finally, due to the dependence between kernels, the number of dependence edges per iteration increases, e.g., between 0.2–40% for matrices in this study, making it challenging for existing DAG partitioners to create balanced workloads for all cores.

We present sparse fusion, a technique that creates an efficient schedule and fused code for combining a sparse kernel with loop-carried dependencies and another sparse kernel. Sparse fusion uses an inspector to apply a novel iteration composition and ordering (ICO) runtime scheduling algorithm on the DAGs of the two input sparse kernels. ICO uses a vertex dispersion strategy to balance the workloads in the fused schedule, uses two novel iteration packing heuristics to improve the data locality by exploiting the spatial and temporal locality of the merged computations, and uses vertex pairing strategies to aggregate iterations without explicitly joining the DAGs.

**Motivation Example.** Figure 2 compares the schedule created by sparse fusion (sparse fusion schedule) with the schedules created by applying LBC to the individual DAGs of each sparse kernel (LBC unfused schedule) and LBC applied to the joint DAG (LBC joint DAG schedule). All approaches take the input DAGs in Figure 2b. Solid purple vertices represent the DAG of sparse triangular solve (SpTRSV), and the dash-dotted yellow vertices correspond to sparse matrix-vector multiplication (SpMV). LBC is a DAG partitioner that partitions a DAG into a set of aggregated wavefronts called s-partitions that can run in parallel in a separate wavefront. S-partitioning methods may improve the temporal locality between the two kernels, but this can compromise the spatial locality within each kernel. For example, for two sparse kernels that only share a small array and operate on different sparse matrices, optimizing temporal locality between kernels will not be beneficial. Finally, due to the dependence between kernels, the number of dependence edges per iteration increases, e.g., between 0.2–40% for matrices in this study, making it challenging for existing DAG partitioners to create balanced workloads for all cores.
are also not load balanced, for example, $s_2$ only has one partition. Throughout the paper, load balance for $s$-partitions means that each $s$-partition needs to have the specifications of different workloads among its own computations. Sparse fusion uses ICO to first partition the SpTRSV DAG and then disperses the SpMV iterations to create load-balanced $s$-partitions, e.g., the two $s$-partitions in Figure 2a have three closely balanced partitions.

SpTRSV solves $Lx = b$ to find $x$, and SpMV performs $y = A \ast x$ where $L$ is a sparse lower triangular matrix, $A$ is a sparse matrix, and $x$, $b$, and $y$ are vectors. The LBC joint DAG schedule interleaves iterations of the two kernels to reuse $x$. However, this can compromise spatial locality within each kernel because the shared data between the two kernels, $x$, is smaller than the amount of data used within each kernel, $A$ and $L$. With the help of a reuse metric, sparse fusion realizes the larger data accesses inside each kernel and hence packs iterations to improve spatial locality within each kernel.

We implement sparse fusion as an embedded domain-specific library in C++ that takes the specifications of the sparse kernels as input and generates an efficient and correct parallel fused code. The primary focus of sparse fusion is to fuse two sparse kernels where at least one of the kernels has a loop-carried dependence. We test sparse fusion on six of the most commonly used sparse kernel combinations in scientific codes, which include kernels such as SpTRSV, SpMV, incomplete Cholesky, incomplete LU, and diagonal scaling. We evaluate sparse fusion against unfused and fused implementations across all symmetric positive definite matrices larger than 100K nonzero elements from SuiteSparse [15]. Sparse fusion is faster than the best of unfused implementations using MKL or ParSy by an average factor of 4.2× and is faster than the best-fused implementations using LBC, DAGP, and wavefront techniques applied to the joint DAG by an average factor of 4×. We also use sparse fusion to fuse more than two loops in the Gauss-Seidel kernel, resulting in an average speedup of 1.3× and 1.8× compared to ParSy and the best of Joint-DAG respectively.  

2 SPARSE FUSION

Sparse fusion is implemented as an inspector-executor technique that can be used as a library. The inspector includes the ICO algorithm and functions that generate its inputs, i.e. dependency DAGs, reuse ratio, and the dependency matrix. The executor is the fused code that is created by the fused transformation.

2.1 Overview

For every kernel pair, sparse fusion generates an inspector and an executor, such as Listing 1, for the kernels in Figure 2a. The inspector first builds the inputs to ICO using the functions intra_DAG, inter_DAG, and compute_reuse in lines 6–8 in Listing 1 and then calls ICO in line 10 to generate FusedSchedule for $r$ threads. Then the executor code, fused_code in line 12 in Listing 1, runs in parallel using the fused schedule. The fused schedule can be reused as long as the sparsity patterns of $A$ and $L$ do not change.

2.2 The Inspector in Sparse Fusion

The inputs of the ICO algorithm are the dependency matrix between kernels, the DAG of each kernel, and a reuse ratio. Sparse fusion analyzes the kernel code to generate inspector components that create these inputs.

Dependency DAGs: Lines 6–7 in Figure 1 use an internal domain-specific library to generate the dependency DAG of each kernel. General approaches such as the work by Mohammadi et al. [31] can also be used to generate the DAGs, however, that would lead to higher inspection times compared to a domain-specific approach. For example, with domain knowledge, sparse fusion will use the $L$ matrix as the SpTRSV DAG $G_1$ in Figure 2b. Each nonzero $L_{ij}$ represents a dependency from iteration $i$ to $j$.

```
Matrix inter_DAG(CSR A, CSC L, double *b, double *x, double *y){
    for(i=0; i<n; i++){
        j1 = i;
            F[j1].append(i1);
    return F;
}
```

Listing 2: inter_DAG function for the example in Figure 2a.

Dependency Matrix $F$: ICO uses the dependency information between kernels to create a correct fused schedule. By running the inter_DAG function, sparse fusion creates this information and stores it in matrix $F$. To generate inter_DAG, sparse fusion finds dependencies between statements of the two kernels by analyzing the body of the outermost loop of kernels. Each nonzero $F_{ij}$ represents a dependency from
iteration $j$ of the first loop, i.e., column $j$ of $F$, to iteration $i$ of the second loop, i.e., row $i$ of $F$. In Figure 2a, there exists a read after write (flow) dependency between statements $x[11]$ in line 6 and $x[1j]$ in line 11. As a result, sparse fusion generates the function shown in Listing 2. The resulting $F$ matrix, generated at runtime, is shown in Figure 2b.

Reuse Ratio: ICO uses a reuse ratio based on the memory access patterns of the kernels to decide whether to improve locality within kernels or between them. The inspector in line 9 in Listing 1 computes the reuse ratio metric. The metric represents the ratio of common to total memory accesses. If the reuse ratio is greater than or equal to one, the vertex $v$ is classified as slack, otherwise it is classified as non-slack. The vertex $v$ has a vertex set $V_i$, $i \in \{1, 2\}$, and an edge set $E_i$ which stores the dependencies between loops. A DAG shown in Figure 2a represents two loop nests. The vertex $v$ is non-slack if there exists an edge between two loop vertices of a loop and is defined as the total number of nonzeros touched by the instruction, and the reuse ratio discussed in section 2.2.

Therefore, we use vertex and iteration interchangeably and similarly DAG and loop. $c(v_i)$ is the computational load of a loop and is defined as the total number of nonzeros touched to complete its computation. Other inputs to the algorithm are the number of requested partitions $r$, which is set to the number of cores, and the reuse ratio discussed in section 2.2.

The output of ICO is a fused partitioning $\mathcal{V}$ that has $b \geq 1$ s-partitions, each s-partition contains up to $k > 1$ w-partitions, where $k \leq r$. ICO creates $b$ disjoint s-partitions from vertices of both DAGs, shown with $\mathcal{V}_{S_i}$, where $\cup j = \emptyset$, $\forall j$. Each s-partition includes vertices from a lower bound and upper bound of wavefront numbers shown with $s_j = \{l_j, u_j\}$ as well as some slack vertices. For each s-partition $\mathcal{V}_{S_i}$, ICO creates $m_i \leq k$ independent w-partitions $\mathcal{V}_{S_i, w_j}$.
where $V_{s_1}, w_1 \cup ... \cup V_{s_t}, w_m = V_h$. Since w-partitions are independent, they can run in parallel.

**Example.** In Figure 2b, the SpTRSv DAG $G_1$, the SpMV DAG $G_2$, and the inter-DAG dependency matrix $F$ are inputs to ICO. Other inputs to ICO are $r=3$ and the $\text{reuse, ratio}$. The fused partitioning shown in Figure 2e has two s-partitions ($b=2$). The first s-partition has three w-partitions ($m_t=3$) shown with $V_{s_1} = \{1, 2, 3, 4\}; \{5, 6, 7, 8, 9\}$, where the underscored vertices belong to $G_1$.

### 3.2 The ICO Algorithm

Algorithm 1 shows the ICO algorithm. It takes the inputs and goes through three steps of (1) vertex partitioning and partition pairing with the objective to compose iterations of loops that can run independently; (2) merging and slack vertex assignment to reduce synchronization and to balance workloads; and (3) packing to improve thread locality.

#### 3.2.1 Vertex Partitioning and Partition Pairing

The first step of ICO partitions one of the input DAGs $G_1$ or $G_2$, and then uses that partitioning to partition the other DAG. The created partitions are stored in $V$. Due to inter-DAG dependence, the number of dependent edges between iterations increases after fusion, posing challenges to load balancing. For example, across the studies in this paper and for all symmetric positive definite (SPD) matrices with nonzeros larger than 100K in SuiteSparse [15], the average number of edges per vertex increases between 0.2–40% after fusion.

To find independent workloads for threads, ICO ignores the dependencies across loops and first creates a partitioning from one of the DAGs with the help of vertex partitioning. Then the other DAG is partitioned using a partition pairing strategy. The DAG that is partitioned first is the head DAG and the other is the tail DAG. The joint-DAG does not need to be explicitly created in this two-step process, enabling scalability.

**Vertex partitioning.** ICO first selects the DAG with edges as the head DAG in line 1 in Algorithm 1. Then it uses the LBC DAG partitioner [10] to construct a partitioning of the head DAG, $G_h$, in line 2 of Algorithm 1 by calling the function LBC. The resulting partitioning has a set of disjoint s-partitions. Each s-partition contains $k$ disjoint w-partitions which are balanced using vertex weights. Disjoint w-partitions ensure all w-partitions within s-partitions are independent. The created partitions are stored in a two-dimensional list $H$ using list.

**Partition pairing.** The algorithm then partitions the tail DAG using partition pairing. Pair-partitions are self-contained so that they can run in parallel if assigned to the same s-partition. The created partitions are put in the fused partitioning $V$ to be used in step two. Pair partitions $H_{ij}$ and $T_{ij}$ are called self-contained if all reachable vertices from a breadth-first search (BFS) on $V_0 \in H_{ij} \cup T_{ij}$ through vertices of $G_1$ and $G_2$ are in $H_{ij} \cup T_{ij}$. Self-contained pair partitions ($P_{i_p}, T_{i_p}$) and ($H_{iq}, T_{iq}$) can execute in parallel without synchronization if they are in the same wavefront $i$. Partitions that do not satisfy this condition create synchronization in the final schedule.

Lines 3–7 show the partitioning of the tail DAG by performing a BFS on the dependence matrix, starting from the head DAG partitioning $H$. Then in line 5, partition pairing removes vertices whose dependence is not satisfied by calling the $\text{remove_uncontained}$ function. This step ensures the self-contained condition. Finally, in line 6, the created partitions are added to the partitioned graph $P_G$ where each vertex of $P_G$ is a partition and edges represent dependence.

**Example.** Figure 4b shows the output of ICO after the first step for the inputs in Figure 2b. ICO chooses $G_1$ as the head DAG because it has edges ($|E_1| > 0$), while $G_2$ has no edges. In vertex partitioning, $G_1$ is partitioned with LBC to create up to three w-partitions (because $r = 3$) per s-partition. The created partitions are shown in Figure 4a and are stored in $H$. The first s-partition $V_{s_1}$ is stored in $H_1$. The three w-partitions of $V_{s_1}$ are indexed with $H_{1,1}$, $H_{1,2}$, and $H_{1,3}$. Similarly, $V_{s_2}$ is stored in $H_2$. Figure 4b shows the partitioned graph $P_G$ after partition pairing.

#### 3.2.2 Merging and Slack Vertex Assignment

The second step of ICO reduces the number of synchronizations by merging...
some of the pair partitions in a merging phase. It also improves load balance by dispersing vertices across partitions using slack vertex assignment.

**Slack definitions.** A vertex \( v \) can always run in its wavefront number \( l(v) \). However, the execution of vertex \( v \) can sometimes be postponed up to \( SN(v) \) wavefronts without the need to move its dependent vertices to later wavefronts. \( SN(v) \) is the slack number of \( v \) and is defined as \( SN(v) = P_G - l(v) - \text{height}(v) \) where \( \text{height}(v) \) is the maximum path from a vertex \( v \) to a sink vertex (a sink vertex is a vertex with no outgoing edge), \( P_G \) is the critical path of \( G \), and \( l(v) \) is the wavefront number of \( v \). A vertex with a positive slack number is a slack vertex. To compute vertex slack numbers efficiently, instead of visiting all vertices, ICO iterates over partitions \( (G_P) \) and computes the slack number of each partition in the partitioned DAG, i.e., partition slack number. The computed slack number for a partition is assigned to all vertices of the partition. As shown in line 8 of Algorithm 1, all partition slack numbers of \( G_P \) are computed via \texttt{s\_slack\_info} and are stored in \( S \). For example, since vertices in \( V_{S_2}, w_3 \) can be postponed one wavefront, from s-partition 2 to 3, their slack number is 1. Vertices in w-partitions \( V_{S_2}, w_1 \) and \( V_{S_3}, w_1 \) cannot be moved because their slack numbers are zero.

**Merging.** ICO finds pair partitions with partition slack number of zero and then merges them as shown in lines 9-11. Since pair partitions are self-contained, merging them does not affect the correctness of the schedule. Algorithm 1 visits all pair partitions \( (w, w') \) in \( G_p, \text{pairs} \) and merges them using the \texttt{merge} function in line 10 if their slack numbers are zero, i.e., \( SN(w) = 0 \) and \( SN(w') = 0 \).

**Slacked vertex assignment.** The algorithm then reorders slacked vertices to approximately load balance the w-partitions of an s-partition using a cost model. The cost of w-partition \( w \in V_{S_i} \) is defined as \( \text{cost}(w) = \sum v \in wc(v) \). A w-partition is balanced if its maximal difference is smaller than a threshold \( \epsilon \). The maximal difference for a w-partition inside an s-partition is computed by subtracting its cost from the cost of the w-partition (from the same s-partition) with the maximum cost. ICO first removes all slacked vertices \( S \) from the \( G_p \) and stores it as fused partitioning \( V \) in line 12. It then goes over every s-partition \( i \) and balances \( V_{S_i} \) by assigning a slack vertex to its imbalanced w-partition. The function \texttt{balance\_with\_slack} in line 14 balances each partition using either a slack vertex of the pair partition or a slack vertex \( q_i \in S \) from any other partition that satisfies \( l(q_i) < i < (l(q_i) + SN(q_i)) \). In line 15, slacked vertices in \( S \) that are not postponed to later s-partitions are evenly divided between the w-partitions of the current s-partition \( (V_{S_i}) \) using the \texttt{assign\_even} function.

**Example.** Figure 4d shows the output of the second step of ICO from the partitioning in Figure 4b. First, pair partitions \( (V_{S_1}, w_1, V_{S_2}, w_1, V_{S_3}, w_1) \), shown with red dash-dotted circles in Figure 4b, are merged because their slack numbers are zero. The resulting merged partition is shown in Figure 4e. Then slacked vertex assignment balances the w-partitions in Figure 4c. The balanced partitions are shown in Figure 4d. The slacked vertices, \( S \), are shown with dotted blue circles in Figure 4c. The w-partitions in \( V_{S_1} \) are balanced using vertices of their pair partitions, e.g., the yellow dash-dotted vertices 5 and 6 are moved to \( w_3 \) in \( V_{S_1} \) as shown in Figure 4d. The second strategy in \texttt{balance\_with\_slack} is used to balance partitions in \( V_{S_2} \). This is because the slack vertices in \( S \) can execute in either s-partition two or three, since they are from s-partition one and have a slack number of one, and they are used to balance the w-partitions in \( V_{S_2} \).
3.2.3 Packing. The third step of ICO reorders the vertices inside a w-partition to improve data locality for a thread. The previous steps of the algorithm create w-partitions that are composed of vertices of one or both kernels, but the order of execution is not defined. Using the reuse ratio, the order in which the nodes in a w-partition should be executed is determined with a packing strategy. ICO has two packing strategies: (i) in interleaved packing, the iterations of the two loops are interleaved to improve temporal locality between loops and (ii) in separated packing, the vertices of each kernel are executed separately to benefit from spatial locality within iterations of a loop. When the reuse ratio is greater than one, in line 17 of Algorithm 1, the function interleaved_pack is called to interleave iterations of the two kernels based on F. Otherwise, separated_pack is called in line 18.

Example. Figure 2e shows the output of ICO’s third step from the partitioning in Figure 4d. Since the reuse ratio is smaller than one, separated packing is chosen and $V_{s2}$, $w_1$ is stored as $V_{s2}$, $w_1 = [10, 11, 10, 11]$. Vertices are ordered to keep dependent iterations of SpTRSV and consecutive iterations of SpMV next to each other.

3.3 Fusing More than Two Loops
The ICO algorithm processes one loop at a time and thus efficiently supports the fusion of any number of loops without the explicit creation of the joint DAG, which can be infeasible. For more than two loops, ICO processes their DAGs in the order they appear in the code. The first two loops are fused as described in Section 3.2. Partition pairing uses the final partitioned fused schedule of the previous loop as the new head and the additional DAG as a tail. In the second step, ICO finds slacked partitions and applies merging and slack vertex assignment as described in Section 3.2.2. And finally, vertices inside each w-partition are sorted based on ICO packing strategies. It is important to note that sparse fusion always applies fusion. When fusing loops is not profitable, sparse fusion performance becomes almost identical to the unfused performance. We will discuss the efficiency of ICO for more than one loop using a case study in the experimental results section.

4 EXPERIMENTAL RESULTS
We compare the performance of sparse fusion to MKL [46] and ParSy [10], two state-of-the-art tools that accelerate individual sparse kernels, which we call unfused implementations. Sparse fusion is also compared to the three fused implementations that we create. To our knowledge, sparse fusion is the first work that provides a fused implementation of sparse kernels where at least one kernel has loop-carried dependencies. For comparison, we also create three fused implementations of sparse kernels by applying LBC, DAGP, and a wavefront technique to the joint DAG of the two input sparse kernels and creating a schedule for execution using the created partitioning. The methods will be referred to as fused LBC, fused DAGP, and fused wavefront, respectively.

4.1 Setup.
All symmetric positive definite matrices larger than 100K nonzeros from [15] are used for experimental results. The matrix values are real and stored in double precision. The test-bed architecture is a multicore processor with 20 Intel CascadeLake cores at 2.5 GHz with 33 MB L3 Cache. All generated codes, implementations of different approaches, and library drivers are compiled with GCC v.11.3.0 compiler and with the -O3 flag. Each thread is pinned to a physical core and a close thread binding is selected. Matrices are first reordered with METIS [25] to improve thread parallelism.

We compare sparse fusion with two unfused implementations where each kernel is optimized separately: I. ParSy applies LBC to DAGs that have edges. For parallel loops, the method runs all iterations in parallel. LBC is developed for L-factors [14] or chordal DAGs. Thus, we make DAGs chordal before using LBC. II. MKL uses Intel MKL [46] routines with MKL 2021.1.0 and calls them separately for each kernel. We use the inspector executor version of MKL by calling mkl_sparse_set_mv_hint, mkl_sparse_set_mv_hint, and mkl_sparse_optimize for inspection. For the executor of SpTRSV, SpMV, and SpILU0 we use mkl_sparse_d_trsv, mkl_sparse_d_mv, and dcsrcilu0, respectively.

Sparse fusion is also compared to three fused approaches, all of which take as input the joint DAG; the joint DAG is created by combining the DAGs of the input kernels using the inter-DAG dependency matrix $F$. We then implement three approaches to build the fused schedule from the joint DAG: I. Fused wavefront traverses the joint DAG in topological order and builds a list of wavefronts that show vertices of both DAGs that can run in parallel. II. Fused LBC applies the LBC algorithm to the joint DAG and creates a set of s-partitions, each composed of independent w-partitions. LBC is taken from ParSy and its parameters are tuned for best performance. We use 4 for initial_cut and 400 for coarsening_factor. III. Fused DAGP applies the DAG partitioning algorithm to the joint DAG and then executes all independent partitions that are in the same wavefront in parallel. DAGP is used with METIS for its initial partitioning, with one run ($runs=1$), and the remaining parameters are set to default. All fused approaches use sparse fusion packing.

The list of sparse kernel combinations investigated is in Table 1. To demonstrate sparse fusion’s capabilities, the sparse kernels are selected with different combinations of storage formats, i.e., CSR and compressed sparse column (CSC) storage, different combinations of parallel loops and loops with
Table 1: The list of kernel combinations. CD: loops with carried dependencies, SpIC0: Sparse Incomplete Cholesky with zero fill-in, SpILU0: Sparse Incomplete LU with zero fill-in, DSCAL: scaling rows and columns of a sparse matrix.

<table>
<thead>
<tr>
<th>ID</th>
<th>Kernel combination</th>
<th>Operations</th>
<th>Dependency DAGs</th>
<th>Reuse Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SpTRSV CSR - SpTRSV CSR</td>
<td>$x = L^{-1}y, z = L^{-1}x$</td>
<td>CD - CD</td>
<td>$\frac{2n^2+10n}{2n^2+10n} \geq 1$</td>
</tr>
<tr>
<td>2</td>
<td>DSCAL CSR - SpILU0 CSR</td>
<td>$LU \approx DAD^T$</td>
<td>Parallel - CD</td>
<td>$\frac{max(2n+size_A, size_A+2n)}{2n^2} \geq 1$</td>
</tr>
<tr>
<td>3</td>
<td>SpTRSV CSR - SpMV CSC</td>
<td>$y = L^{-1}x, z = Ay$</td>
<td>CD - Parallel</td>
<td>$\frac{max(2n+size_A, size_A+2n)}{2n^2} &lt; 1$</td>
</tr>
<tr>
<td>4</td>
<td>SpIC0 CSC - SpTRSV CSC</td>
<td>$LL^T \approx A, y = L^{-1}x$</td>
<td>CD - CD</td>
<td>$\frac{max(size_A+4n, size_A)}{2n^2} \geq 1$</td>
</tr>
<tr>
<td>5</td>
<td>SpILU0 CSR - SpTRSV CSR</td>
<td>$LU \approx A, y = L^{-1}x$</td>
<td>CD - CD</td>
<td>$\frac{max(size_A+4n, size_A)}{2n^2} \geq 1$</td>
</tr>
<tr>
<td>6</td>
<td>DSCAL CSC - SpIC0 CSC</td>
<td>$LL^T \approx DAD^T$</td>
<td>Parallel - CD</td>
<td>$\frac{max(size_A+4n, size_A)}{2n^2} \geq 1$</td>
</tr>
</tbody>
</table>

Figure 5: Performance of different implementations shown in GFLOPs per second.

carried dependencies, and a variety of memory access pattern behaviour. For example, combinations of SpTRSV, $Lx = b$ and SpMV are main bottlenecks in conjugate gradient methods [4, 52]. GMRES [11], Gauss-Seidel [32]. Preconditioned Krylov methods [20] and Newton solvers [35] frequently use kernel combinations 3, 5, 6, 7. The s-step Krylov solvers [6] and s-step optimization methods used in machine learning [35] provide even more opportunities to interleave iterations. Thus, they use these kernel combinations significantly more than their classic formulations.

4.2 Sparse Fusion Performance.

Figure 5 shows the performance of the fused code from sparse fusion, the best-unfused implementation from ParSy and MKL, and the best-fused wavefront, fused LBC, and fused DAGP implementations. We report the minimum execution time of ParSy and MKL per matrix for each unfused data point in Figure 5. For each fused data point in Figure 5, we report the minimum execution time of fused wavefront, fused LBC, and fused DAGP per each matrix. The performance of implementations is shown in floating point operations per second (GFLOP/s). The theoretical floating point operations are computed per kernel combination and matrix and used for all implementations. Sparse fusion is on average 4.2× and 4× faster than the best unfused and fused implementations, respectively. For all kernel combinations and matrices shown in Figure 5, sparse fusion provides the fastest execution time in 76% of instances compared to the best fused and unfused implementations. Even though sparse fusion is on average 11.5× faster than MKL for ILU0-TRSV, since ILU0 only has a sequential implementation in MKL, the speedup of this kernel combination is excluded from the average speedups.

Locality in Sparse Fusion. The efficiency of the reuse ratio differs per kernel properties. Kernel combinations 1, 2, 4, 5, and 6 share the sparse matrix $L$ and thus have a reuse ratio larger than one, while combination 3 only shares vector $y$. 

leading to a reuse ratio lower than one. The selected packing strategy in sparse fusion improves the performance in 88% of kernel combinations and matrices and provides a 1-3.9× improvement in both categories. The highest improvement belongs to kernel combination 1, where accessing L values dominates the execution time, and reusing them after interleaved packing always leads to improvement. In kernel combinations including SpILU0 and SpIC0, the effect is lower since the accesses are more irregular than the rest.

Figure 6 top shows the average memory access latency [21] of sparse fusion, the fastest unfused implementation (ParSy), and the fastest fused partitioning-based implementation (Fused LBC) for all kernel combinations normalized over the ParSy average memory access latency (shown for matrix bone010 as an example, other matrices exhibit similar behavior). The average memory access latency is used as a proxy for locality and is computed using the number of accesses to L1, LLC, and TLB measured with PAPI performance counters [42].

For kernels 1, 2, 4, 5, and 6 where the reuse ratio is larger than one, the memory access latency of ParSy is on average 1.3× larger than that of sparse fusion. Because of their high reuse ratio, these kernels benefit from optimizing locality between kernels made possible via interleaved packing. ParSy optimizes locality in each kernel individually. When applied to the joint DAG, LBC can potentially improve the temporal locality between kernels and thus there is only a small gap between the memory access latency of sparse fusion and that of fused LBC. For kernel combination 3 where the reuse ratio is smaller than one, the gap between the memory access latency of sparse fusion and fused LBC is larger than the gap between the memory access latency of ParSy and ParSy. Sparse fusion and ParSy both improve data locality within each kernel for these kernel combinations.

Load Balance and Synchronization in Sparse Fusion. Figure 6 bottom shows the OpenMP potential gain [34] of sparse fusion, ParSy, and Fused LBC for all kernel combinations normalized over ParSy’s potential gain (shown for matrix bone010 as an example). The OpenMP potential gain is a metric in Vtune [53] that shows the total parallelism overhead, e.g., wait-time due to load imbalance and synchronization overhead, divided by the number of threads. This metric is used to measure the load imbalance and synchronization overhead in ParSy, fused LBC, and sparse fusion.

Kernel combination 3 has slack vertices that provide opportunities to balance workloads. For example, for the studied matrices, between 39-82% of vertices can be slacked, thus the potential gain balance of ParSy is 1.6× larger than that of sparse fusion and 2.4× lower than that of fused LBC. ParSy can only improve load balance using the workloads of an individual kernel. As shown in Figure 1, for kernel combination 4, the joint DAG has a small number of parallel iterations in the final wavefronts that makes the final s-partitions of the LBC fused implementation imbalanced (a similar trend exists for kernel combination 5). For these kernel combinations, the code from sparse fusion has on average 33% fewer synchronization barriers compared to ParSy due to merging. For kernel combinations 1, 2, 3, and 6, the potential gain in sparse fusion is 1.3× less than that of ParSy. Merging in sparse fusion reduces the number of synchronizations in the fused code on average by 50% compared to that of ParSy. The sparsity pattern of matrices has a direct influence on thread parallelism. To ensure sufficient parallel iterations, all matrices are reordered with METIS, which also allows different schedulers to enhance the load balance. The exploration of techniques that do not rely on METIS is a topic for future work.

Inspector Time. Figure 7 shows the number of times that the executor should run to amortize the cost of inspection for implementations that have an inspector. For space reasons, only combinations 3 and 5 are shown, others follow the same trend. The number of executor runs (NER) that amortize the cost of inspector for an implementation is calculated using

Inspector Time

Baseline Time—Executor Time

The baseline time is obtained by running each kernel individually with a sequential implementation, and the inspector and executor times belong to the specific implementation. When NER is negative, it means the inspector is not amortized in that tool. The fused LBC implementation has a NER of 3.1-745. The high inspection time is because of the high cost of converting the joint DAG into a chordal DAG, typically consuming 64% of its inspection time. The NER of the fused DAGP implementation is either negative or higher than 80. The fused wavefront implementation sometimes has a negative NER because the executor time
is slower than the baseline time. As shown, sparse fusion, MKL, and ParSy have the lowest NER among all implementations. Sparse fusion’s low inspection time is due to pairing strategies that enable partitioning one DAG at a time. Kernel combinations such as SpIC0-TRSV and SpLLU0-TRSV only need one iteration to amortize the inspection time and SpTRSV-SpMV, SpTRSV-SpTRS, and SpMV-SpTRSV need between 11-50 iterations. Sparse kernel combinations are routinely used in iterative solvers in scientific applications. Even with preconditioning, these solvers typically converge to an accurate solution after tens of thousands of iterations [4, 11, 27], hence amortizing the overhead of inspection.

Figure 8 compares the performance of two DAG partitioners, DAGP and LBC, for different sizes of sparse DAGs to demonstrate the expensive inspection time of fused joint-DAG implementations in Figure 7. In the one-DAG configuration, the DAG partitioner partitions the DAG of sparse triangular solve (SpTRSV) CSR. In the joint DAG configuration, the DAG partitioner partitions the joint DAG of the sparse matrix-vector multiplication (SpMV) CSR and SpTRSV CSR. To compare the joint DAG configuration with the one-DAG configuration, the x-axis shows the number of edges in one of the DAGs, i.e., SpTRSV DAG. The number of edges in the joint DAG is three times the number of edges in the SpTRSV DAG. As shown in Figure 8, DAGP in both one-DAG and joint-DAG configurations is slower than LBC for both small and large-size DAGs. Also, DAGP on the joint DAG runs out of memory for the last seven large DAGs (hence not shown in the figure).

### 4.3 Sparse Fusion Extension

We discuss the potential for sparse fusion when fusing more than two loops and parallel loops.

*Gauss-Seidel, a case study for fusing more than two loops.* To demonstrate the efficiency of sparse fusion in merging more than two loops, we use Gauss-Seidel (GS) [32] as an end-to-end case study. GS iteratively solves for the unknown vector $x$ in $Ax = b$ where $A$ is a sparse symmetric matrix stored in a CSR format, and $b$ is a vector. We specifically use backward GS [32] that in its $i^{th}$ iteration updates the solution by computing $(D - F)x_{i+1} = Ex_i + b$ where $D$, $F$, and $E$ are diagonal, lower triangular, and upper triangular matrices with a decomposition of $A = D - F - E$, respectively. Each iteration of GS computes an SpMV followed by SpTRSV. By unrolling the outermost loop of GS, Sparse Fusion has the opportunity to fuse more than two loops, e.g., unrolling one iteration exposes four kernels/loops for fusion; unrolling loops of iterative solvers for performance is a commonly used approach, e.g., $s$-step solvers [6].

The choice of SPD matrices guarantees convergence in GS. The linear system corresponding to each matrix is solved for either the accuracy threshold of $10^{-6}$ or the most accurate solution after 1000 iterations. To detect profitable loops for sparse fusion and joint-DAG approaches, we exhaustively
search for a fusion of 2–6 loops and select the fastest code. Figure 9 compares the performance of GS using sparse fusion (GS Sparse Fusion), ParSy (GS ParSy), and the best of joint-DAG implementations (GS Joint-DAG) for all the selected matrices. In more than 96% of the matrices, GS-Sparse-Fusion is faster than GS ParSy and GS joint-DAG. GS Sparse Fusion also provides an average speedup of 1.3× and 1.8× over GS ParSy and GS Joint-DAG, respectively, demonstrating the efficiency of sparse fusion when fusing more than two loops to accelerate iterative solvers. We also found that fusing more than 6 loops does not lead to improvement. In the reported execution times for sparse fusion in Figure 9, 37%, 8%, and 55% of data points are obtained from fusing two, four, and six loops respectively. This indicates the value of fusing more than two loops in iterative solvers.

**Fusing parallel loops.** While sparse fusion and ICO are designed for loops with carried dependence, they can also be used to fuse parallel loops, such as the sparse matrix-vector product (SpMV) with SpMV. Figure 10 compares the performance of fused SpMV-SpMV with the unfused MKL implementation. As shown, sparse fusion provides an average speedup of 1.18× over MKL. The sparse fusion implementation does not benefit from vector instructions, while MKL is a highly-optimized code. Sparse fusion focuses on thread parallelism and improves zero-stride (temporal locality) and unit-stride (spatial locality), potentially improving vectorization when combined with existing vectorization techniques [8, 47] for sparse matrix kernels.

**5 RELATED WORK**

Parallel implementations of individual sparse matrix kernels exist in both highly-optimized libraries [22, 29] and compiler frameworks [2, 9, 31, 40, 48]. High-performance sparse libraries optimize a sparse kernel such as [43, 45, 49] optimizes SpTRSV, [3, 28] optimizes SpMV, and MKL [46] contains optimized versions of several sparse kernels individually. Compiler frameworks such as Sympiler [7, 9, 12] and sparse polyhedral framework [40] also provide techniques to optimize a sparse kernel with loop-carried dependence. These libraries and frameworks provide an efficient implementation for their supported sparse kernels.

Inspector-executor frameworks commonly use wavefront parallelism [19, 44] to parallelize sparse matrix computations with loop-carried dependencies. Recently, task coarsening approaches such as LBC [10], HDagg [50] and DAGP [23] coarsen wavefronts and thus generate code that is optimized for parallelism, load balance, and locality. While available approaches can provide efficient optimizations for sparse kernels with or without loop-carried dependencies, they can only optimize sparse kernels individually.

A number of libraries and compiler frameworks provide parallel implementations of fused sparse kernels with no loop-carried dependencies, such as tensor multiplication kernels in ReACT [51] and SparseLNR [17], two or more SpMV kernels [24, 30] or SpMV and dot products [1, 16, 18]. The formulation of s-step Krylov solvers [6] enables iterations of iterative solvers to be interleaved and hence multiple SpMV kernels are optimized simultaneously by replicating computations to minimize communication costs [24, 30]. Sparse tiling [37–39, 41] is an inspector-executor approach that uses manually written inspectors [37, 39] to group iterations of different loops of a specific kernel such as Gauss-Seidel [39] and Moldyn [37] and is generalized for parallel loops without loop-carried dependencies [41]. Sparse fusion optimizes combinations of sparse kernels where at least one of the kernels has loop-carried dependencies.

**6 CONCLUSION**

We present sparse fusion and demonstrate how it improves parallelism, load balance, and data locality in sparse matrix combinations compared to when sparse kernels are optimized separately. Sparse fusion inspects the DAGs of the input sparse kernels and uses the ICO algorithm to balance the workload between wavefronts and determine whether to optimize data locality within or between the kernels. Sparse fusion’s generated code outperforms state-of-the-art implementations for sparse matrix optimizations. In future work, we plan to investigate strategies that enable sparse fusion for arbitrary sparse operations.

**ACKNOWLEDGMENTS**

We thank the reviewers for their constructive feedback. This work is supported by NSERC Discovery Grants (RGPIN-06516, DGECR00303, RGPIN-2023-04897, DGECR-2023-00133), National Science Foundation (NSF CCF-2106621), the Canada Research Chairs program, the Ontario Early Researcher Award, Chameleon Project [26] and the Digital Research Alliance of Canada (www.alliancecan.ca).
REFERENCES


