# 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

Sparse fusion is a compile-time loop transformation and runtime scheduling implemented as a domain-specific code generator. Sparse fusion generates efficient parallel code for the combination of two sparse matrix kernels, where at least one of the kernels has loop-carried dependencies. Available implementations optimize individual sparse kernels. When optimized separately, the irregular dependence patterns of sparse kernels create synchronization overheads and load imbalance, and their irregular memory access patterns result in inefficient cache usage, which reduces parallel efficiency. Sparse fusion uses a novel inspection strategy with code transformations to generate parallel fused code for sparse kernel combinations that is optimized for data locality and load balance. Code generated by Sparse fusion outperforms the existing implementations ParSy and MKL on average 1.4× and 4.3× respectively and outperforms the LBC and DAGP coarsening strategies applied to a fused data dependence graph on average 5.1× and 8.6× respectively for various kernel combinations.

**ACM Reference Format:**
Kazem Cheshmi, Michelle Mills Strout, and Maryam Mehri Dehnavi. 2023. Runtime Composition of Iterations for Fusing Loop-carried Sparse Dependence . In *Proceedings of SC (SC'23).* ACM, New York, NY, USA, 14 pages.

## 1 INTRODUCTION

Numerical algorithms [39] and optimization methods [4, 8, 44] are typically composed of numerous consecutive sparse matrix computations. For example, in iterative solvers [39] such as Krylov methods [9, 40], 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 [15, 52]. The amount of parallelism varies per wavefront and often tapers off towards 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.

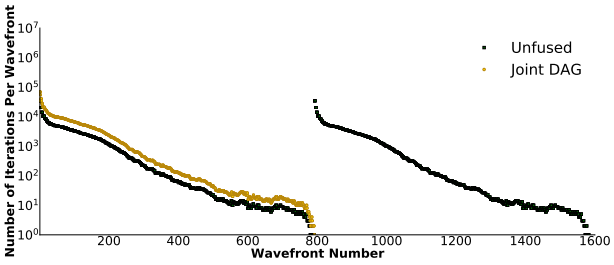
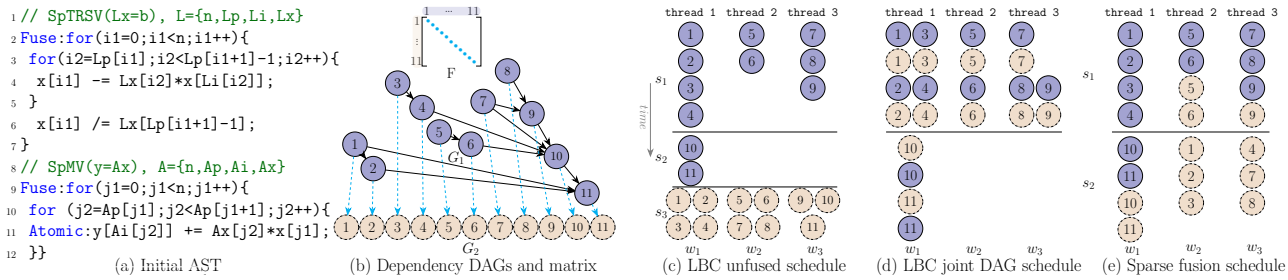



**Figure 1: The nonuniform parallelism in the DAGs of sparse incomplete Cholesky and triangular solver (annotated with unfused) and for the joint DAG of the two kernels results in load imbalance. Higher value in the y-axis shows high parallelism in a given wavefront. Wavefront numbers in the x-axis are numbered based on their order of execution.**

Instead of scheduling iterations of sparse kernels 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 dependencies between iterations of a kernel [6, 18, 48]. A joint DAG includes all of the dependencies between iterations within and across kernels. The joint DAG of sparse kernels with partial parallelism with 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 provides more parallelism per wavefront and significantly reduces the number of wavefronts (synchronizations).

*The work is primarily done while the author was at the University of Toronto.




(a) Initial AST (b) Dependency DAGs and matrix (c) LBC unfused schedule (d) LBC joint DAG schedule (e) Sparse fusion schedule

**Figure 2: Solid purple ($G_1$) and dash-dotted yellow ($G_2$) vertices in Figure 2b in order represent iterations of SpTRSV and SpMV kernels shown in Figure 2a, and edges show the dependencies between iterations. Figures 2c-2e show three different schedules for running SpTRSV followed by SpMV as shown in Figure 2b where the number of processors ($r$) is three. Dashed edges in Figure 2b show dependencies between two kernels and correspond to the nonzero elements of matrix $F$. The unfused implementation schedules each DAG separately as shown in Figure 2c. Two different fused implementations in Figure 2d and 2e use both DAGs and dependencies between kernels to build a fused schedule.**

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. DAG partitioners such as Load-Balanced Level Coarsening (LBC) [7] and DAGP [19] apply aggregation, however, when applied to the joint DAG because they aggregate iterations from consecutive wavefronts, load imbalance might still occur. Also, by aggregating iterations from wavefronts in the joint DAG, DAG partitioning methods potentially improve the temporal locality between the two kernels, but this can disturb 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 profitable. Finally, even when applied to the DAG of an individual kernel, DAGP and LBC are slow for large DAGs because of the overheads of coarsening [19]. This problem exacerbates when applied to the joint because the joint DAG is typically 2-4× larger than an individual kernel's DAG.

We present sparse fusion that creates an efficient schedule and fused code for when a sparse kernel with loop-carried dependencies is combined with another sparse kernel. Sparse fusion uses an inspector to apply a novel Multi-Sparse DAG Partitioning (MSP) runtime scheduling algorithm on the DAGs of the two input sparse kernels. MSP uses a vertex dispersion strategy to balance workloads in the fused schedule, uses two novel iteration packing heuristics to improve the data locality due to spatial and temporal locality of the merged computations, and uses vertex pairing strategies to aggregate iterations without 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 kernels (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 are the DAG of sparse triangular solve (SpTRSV) and the dash-dotted yellow 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 run sequentially, each s-partition is composed of some independent w-partitions. In the LBC unfused schedule in Figure 2c, LBC partitions the SpTRSV DAG and creates two s-partitions, *i.e.* $s_1$ and $s_2$. The vertices of SpMV are scheduled to run in parallel in a separate wavefront $s_3$. This implementation is not load balanced since the number of partitions that can run in parallel differs for each s-partition. In the LBC joint DAG schedule, the DAGs are first joined using the dependency information between the two kernels shown with blue dotted arrows and then LBC is applied to create the two s-partitions in Figure 2d. These s-partitions are also not load balanced, for example $s_2$ only has one partition. Throughout the paper load balance for s-paritions means each s-partition needs to have load balance between its own computations. Sparse fusion uses MSP 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 2e have three closely balanced partitions.

SpTRSV solves $Lx = b$ to find $x$ and SpMV performs $y = A * x$ 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 two kernels to reuse x.

However, this can disturb 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 language in C++ that takes the specifications of the sparse kernels as input, inspects the code of the two kernels, and transforms code to generate 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. Sparse fusion is tested on seven 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. The generated code is evaluated against MKL and ParSy with average speedups of 4.3× and 1.4× respectively. Sparse fusion compared to fused implementations of LBC, DAGP, and wavefront techniques applied to the joint DAG provides on average 5.1×, 8.6× and 2.4× speedup respectively. We also use sparse fusion to fuse more than two loops in the Gauss-Seidel kernel resulting in on 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 a code generator with an inspector-executor technique that can be used as a library. It takes the input specification shown in Figure 3a and generates the inspector and the executor in Figure 3b. The inspector includes the MSP 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 Code Generation

Sparse fusion is implemented as an embedded domain-specific language. It takes as input the specification shown in Figure 3a and generates the driver code in Figure 3b. At compile-time, the data types and kernels in Figure 3a are converted to an initial Abstract Syntax Tree (AST) using `TM.gen_c()` in line 14. Lines 11–12 in Figure 3a demonstrate how the user specifies the two kernels for the running example in Figure 2 as inputs to Sparse fusion. The corresponding AST for the example is shown in Figure 2a. These kernels are defined internally using an intermediate representation (IR), e.g. `For(..,For(..,MulAdd(...)))` for SpMV. Sparse fusion uses the the IR of Sympiler [6]. To add a new kernel to sparse fusion, the kernel should be defined as a new AST node using the available IR nodes.

```
1  #include "def.h"
2  void main(){
3   Int n;
4   Int r(MAX_THREADS);
5   CSR L(n,n,"./L.mtx");
6   CSC A(n,n,"./A.mtx");
7   Vec x(n), y(n);
8   Vec b(n,"./b.mtx");
9   ...
10  Fuse TM(
11     SpTRSV(L,b,x),
12     SpMV(A,x,y)
13    );
14  TM.gen_c("TrsvMv.h"
15     , "Driver.cpp",r);}
```

(a) Input specification

```
1  #include "TrsvMv.h"
2  #include "MSP.h"
3  void main(){
4   L.load();A.load();b.load();
5  /// -------- Inspector -------- ///
6   G1 = SpTRSV.intra_DAG(L);//Sec 2.2
7   G2 = SpMV.intra_DAG(A);
8   F = inter_DAG(A,L,b,x,y);//Sec 2.2
9   reuse_ratio = compute_reuse(
10    A,L,b,x,y); //Sec 2.2
11  FusedSchedule = MSP(G1,G2,F,
12    r,reuse_ratio); //Sec 3
13 /// -------- Executor -------- ///
14  fused_code(L,b,A,x,y,FusedSchedule,
15    reuse_ratio); /*Sec 2.3*/}
```

(b) Driver code (`driver.cpp`)

**Figure 3: Sparse fusion's input and the driver code.**

At runtime by running the driver code in Figure 3b, the inspector creates a fused schedule, and the executor runs it. The inspector first builds inputs to MSP using functions `intra_DAG`, `inter_DAG`, and `compute_reuse` in lines 6–10 in Figure 3b and then calls `MSP` in line 11 to generate `FusedSchedule` for `r` threads. Then the executor code, `fused_code` in line 14 in Figure 3b, runs in parallel using the fused schedule. For every kernel pair, sparse fusion generates and compiles the code once. Sparse fusion's generated code can be used as a library and does not need to change per input matrix.

### 2.2 The Inspector in Sparse Fusion

The inputs of the MSP 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 3b use an internal domain-specific library to generate the dependency DAG of each kernel. General approaches such as work by Mohammadi et al. [31] can also be used to generate the DAGs, however, that will 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 $j$.

*Dependency Matrix F:* MSP 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 AST. Each nonzero $F_{i,j}$ 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[i1]` in line 6 and `x[j1]` in line 11. As a result, sparse fusion generates the function shown

in Listing 1. The resulting $F$ matrix, generated at runtime, is shown in Figure 2b.

```
Matrix inter_DAG(CSR A, CSC L, double *b, double *x, double
     *y){
 for(i1=0; i1<A.n; i1++){
  j1 = i1;
  if(A.p[j1] < A.p[j1+1] )
   F[j1].append(i1);
  }
  return F;}
```

**Listing 1: `inter_DAG` function for the example in Figure 2a.**

*Reuse Ratio:* MSP uses a reuse ratio based on the memory access patterns of the kernels to decide whether to improve locality within each kernel or between the kernels. The inspector in line 9 in Figure 3b computes the reuse ratio metric. The metric represents the ratio of common to total memory accesses of the two kernels, i.e. $\frac{2*\text{common memory access}}{max(\text{kernel1 accesses, kernel2 accesses})}$. For a reuse ratio larger than one, the number of common accesses between the two kernels is larger than the accesses inside a kernel. Sparse fusion estimates memory accesses using the ratio of the size of common variables over the maximum of the total size of variables amongst the kernels. For the running example, the code generated for `compute_reuse` is `2*x.n / max(A.size+x.n+y.n,L.size+x.n+b.n)`. Since `x` is smaller than $L$ or $A$, the reuse ratio is less than one.

## 2.3 Fused Code

To generate the fused code, a fused transformation is applied to the initial AST at compile-time and two variants of the fused code are generated, shown in Figure 4. The transformation variants are *separated* and *interleaved*. The fused code uses the reuse ratio at runtime to select the correct variant for the specific input. The variable `fusion` in line 1 of Figure 4b and 4c is set to `False` if MSP decides fusion is not profitable. Figure 4a shows the sequential loops in the AST, which are annotated with `Fuse`, and are transformed to the separated and interleaved code variants as shown in Figures 4b and 4c, respectively. The separated variant is selected when the reuse ratio is smaller than one. In this variant, iterations of one of the loops run consecutively without checking the loop type. The interleaved variant is chosen when the reuse ratio is larger than one. In this variant, iterations of both loops should run interleaved, and the variant checks the loop type per iteration as shown in lines 6 and 10 in Figure 4c.

# 3 MULTI-SPARSE DAG PARTITIONING

Sparse fusion uses the multi-sparse DAG partitioning (MSP) algorithm to create an efficient fused partitioning that will be used to schedule iterations of the fused code. MSP partitions vertices of the DAGs of the two input kernels to create parallel load-balanced workloads for all cores while improving locality within each thread. This section describes the inputs, output, and three steps of the MSP algorithm (Algorithm 1) using the running **example** in Figures 2 and 5.

## 3.1 Inputs and Output to MSP

The inputs to MSP (shown in Algorithm 1) are two DAGs $G_1$ and $G_2$ from lexicographically first and second input kernels, respectively, and the inter-DAG dependency matrix $F$ that stores the dependencies between kernels. A DAG shown with $G_j(V_j, E_j, c)$ has a vertex set $V_j$ and an edge set $E_j$ and a non-negative integer weight $c(v_i)$ for each vertex $v_i \in V_j$. The vertex $v_i$ of $G_j$ represents iteration $i$ of a kernel and each edge shows a dependency between two iterations of a kernel. $c(v_i)$ is the computational load of a vertex and is defined as the total number of nonzeros touched to complete its computation. Because sparse matrix computations are generally memory bandwidth-bound, $c(v_i)$ is a good metric to evaluate load balance in the algorithm [7]. $F$ is stored in the compressed sparse row (CSR) format and $F_i$ is used to extract the set of vertices in $G_1$ that $v_i \in V_2$ depends on. 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 MSP is a *fused partitioning* $\mathcal{V}$ that has $b \geq 1$ s-partitions, each s-partition contains up to $k > 1$ w-partitions, where $k \leq r$. MSP creates $b$ disjoint s-partitions from vertices of both DAGs, shown with $\mathcal{V}_{s_i}$ where $\cup_{i=0}^{b} \mathcal{V}_{s_i} = V_1 \cup V_2$. Each s-partition includes vertices from a lower bound and upper bound of wavefront numbers shown with $s_i = [lb_i..ub_i)$ as well as some *slack vertices*. For each s-partition $\mathcal{V}_{s_i}$, MSP creates $m_i \leq k$ independent *w-partitions* $\mathcal{V}_{s_i,w_j}$ where $\mathcal{V}_{s_i,w_1} \cup ... \cup \mathcal{V}_{s_i,w_{m_i}} = \mathcal{V}_{s_i}$. Since w-partitions are independent, they can run in parallel.

***Example.*** In Figure 2b, the SpTRSV DAG $G_1$, the SpMV DAG $G_2$, the inter-DAG dependency matrix $F$ are inputs to MSP. Other inputs to MSP are $r$=3 and the *reuse_ratio*. The fused partitioning shown in Figure 2e has two s-partitions ($b$=2). The first s-partition has three w-partitions ($m_1$=3) shown with $\mathcal{V}_{s_1} = \{[\underline{1}, \underline{2}, \underline{3}, \underline{4}]; [\underline{5}, \underline{6}, 5, 6]; [\underline{7}, \underline{8}, \underline{9}, 9]\}$, the underscored vertices belong to $G_1$.

## 3.2 The MSP Algorithm

Algorithm 1 shows the MSP algorithm. It takes the inputs and goes through three steps of *(1)* vertex partitioning and partition pairing with the objective to aggregate iterations without joining the DAGs of the inputs kernels; *(2)* merging and slack vertex assignment to reduce synchronization and to balance workloads; and *(3)* packing to improve locality.

```
Fuse:for(I1){//loop 1
 ...
  for(In)
   x[h(I1,...,In)] = a*y[g(I1,...,In)];
 }
Fuse:for(J1){//loop 2
...
  for(Jm)
   z[h'(J1,...,Jm)] = a*x[g'(J1,...,Jm)];
 }
```

(a) Before

```
if(FusedSchedule.fusion && reuse_ratio < 1){
 for (every s-partition s){
#pragma omp parallel for
  for (every w-partition w){
   for(v ∈ FusedSchedule[s][w].L1){//loop 1
    ...
    for(In)
     x[h(v,...,In)] = a*y[g(v,...,In)];
   }
   for(v ∈ FusedSchedule[s][w].L2){//loop 2
    ...
    for(Jm)
     z[h'(v,...,Jm)] = a*x[g'(v,...,Jm)];
   }}}}
```

(b) After - separated variant

```
if(FusedSchedule.fusion && reuse_ratio >= 1){
 for (every s-partition s){
#pragma omp parallel for
  for (every w-partition w){
   for(v ∈ FusedSchedule[s][w]){
    if(v.type == L1){//loop 1
     for(In)
      x[h(v.id,...,In)] = a*y[g(v,...,In)];

    } else {//loop 2
     for(Jm)
      z[h'(v.id,...,Jm)] = a*x[g'(v,...,Jm)];
    }
   }}}}
```

(c) After - interleaved variant

**Figure 4: The general form of the sparse fusion code transformation with its two variants, interleaved and separated. `I1...In` and `J1...Jm` represent two loop nests. `h'` and `g'` are data access functions. `FusedSchedule` contains the schedule for iterations of loops `I1`, shown with `L1` and `J1`, shown with `L2`.**

*3.2.1 Vertex Partitioning and Partition Pairing.* The first step of MSP 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 $\mathcal{V}$. Partitioning the joint DAG is complex and might not be efficient because of the significantly larger number of edges and vertices added compared to the individual DAG of each kernel. Instead, MSP ignores the dependencies across kernels 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. A *head DAG choice strategy* is used to select the head DAG.

*Vertex partitioning.* MSP uses the LBC DAG partitioner [7] to construct a partitioning of the head DAG in lines 2 and 9 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 number of s-partitions in LBC is set to two for simplicity of notation and illustration. The created partitions are stored in a two-dimensional list $H$ using `list`.

*Partition pairing.* The algorithm then partitions the tail DAG with *forward pairing*, if $G_1$ is the head DAG, or with *backward pairing*, if $G_2$ is the head DAG. With the pairing strategy, some of the partitions of the tail DAG are paired with the head DAG partitions. Pair-partitions are *self-contained* so that they execute in parallel if assigned to the same s-partition. The created partitions are put in the fused partitioning $\mathcal{V}$ to be used in step two. The following first describes the condition for partitions to be self-contained and then explains the forward and backward pairing strategies.

Pair partitions $H_{ij}$ and $T_{ij}$ are called self-contained if all reachable vertices from a breadth first search (BFS) on $\forall v \in H_{ij} \cup T_{ij}$ through vertices of $G_1$ and $G_2$ are in $H_{ij} \cup T_{ij}$.

**Algorithm 1:** The MSP algorithm.

**Input** : $G_1(V_1, E_1, c_1)$, $G_2(V_2, E_2, c_2)$, $F$, $r$, *reuse_ratio*
**Output**: $\mathcal{V}$

```
/* (i) Vertex partitioning and partition pairing    */
1  if |E2| > 0 then
2      [H, k] = LBC(G2, r).list(), T = ∅, V = ∅
3      for (every partition H_{i,j}) do        // Backward pairing
4          T_{i,j} = BFS(H_{i,j}, F, G1)
5          V.add(T_{i,j}, H_{i,j})
6      end
7      if |V| > 2 × (|V1| + |V2|) then  V.fusion = False, exit()
8  else
9      [H, k] = LBC(G1, r).list(), T = ∅, V = ∅
10     for (every partition H_{i,j}) do        // Forward pairing
11         T_{i,j} = BFS(H_{i,j}, F^T, G2)
12         U_{i,j} = T_{i,j}.remove_uncontained(F)
13         V.add(H_{i,j}, T_{i,j}, U_{i,j})
14     end
15 end
   /* (ii) Merging and slacked vertex assignment       */
16 S = slack_info(V)
17 for (every w-partition pair (w, w') ∈ V.pairs) do
18     if (SN(w) = 0) ∧ (SN(w') = 0) then  V.merge(w,w')
19 end
20 V = V − S, ϵ = |V| × 0.001
21 for (every s-partition V_{s_i} ∈ V) do
22     S = V_{s_i}.balance_with_slack(S)
23     if S ≠ ∅ then  S = V_{s_i}.assign_even(S)
24 end
   /* (iii) Packing                                    */
25 if reuse_ratio ≥ 1 then V.interleaved_pack(F)
26 else  V.separated_pack()
```

Self-contained pair partition $(H_{ip}, T_{ip})$ and pair partition $(H_{iq}, T_{iq})$ can execute in parallel without synchronization if

in the same wavefront $i$. Partitions that do not satisfy this condition create synchronization in the final schedule.

The backward pairing strategy starts by partitioning the head DAG. It then uses BFS and the dependence graph $F$ to increase the size of these partitions using the reachable vertices of the tail DAG. Lines 3–8 in Algorithm 1 show this pairing process. If a vertex in the head DAG depends on more than one vertex in the tial DAG, some vertices will get replicated to ensure s-partitions are self-contained. MSP performs fusion only if profitable, hence fusion is disabled (by setting `fusion` to `False` in Line 7 of the algorithm) if the number of redundant computations exceeds a threshold. This threshold is shown in line 7 and is defined as the sum of vertices of both DAGs. In forward pairing (Lines 10–14), partitions of the tail DAG, $G_1$, are used as a seed and then vertices of the head DAG, $G_2$, are added to these partitions with a BFS on the dependence matrix. Forward pairing uses the transpose of the dependence matrix because columns of $F$ need to be accessed. The columns show dependence relations between vertices of $G_1$ to $G_2$. Finally in line 12, forward pairing removes vertices that their dependence is not satisfied by calling the `remove_uncontained` function. This step ensures the self-contained condition. The final pairing is stored in $\mathcal{V}$ in line 13.

*The head DAG choice.* MSP chooses the DAG with edges as the head DAG to improve locality. Locality is improved because the head DAG is partitioned with LBC, which provides good locality. Selecting $G_2$ as the head DAG reduces inspector overhead. If both $G_1$ and $G_2$ are DAGs of kernels with dependencies, then $G_2$ is chosen as the head DAG to reduce inspector overhead. When $G_2$ is partitioned first, MSP chooses backward pairing which is more efficient compared to forward pairing. Forward pairing traverses $F$ and its transpose $F^T$ and thus performs $2 * nnz_F + 2 * n$ operations where $nnz_F$ is the number of nonzeros in $F$. However, backward pairing only traverses $F$ and performs $nnz_F + n$ operations.

***Example.*** Figures 5b shows the output of MSP after the first step for the inputs in Figure 2b. MSP chooses $G_1$ as the head DAG because it has edges ($|E_1| > 1$), $G_2$ 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 5a and are stored in $H$. The first s-partition $\mathcal{V}_{s_1}$ is stored in $H_1$. Similarly, $\mathcal{V}_{s_2}$ is stored $H_2$. Figure 5b shows the output of partition pairing.

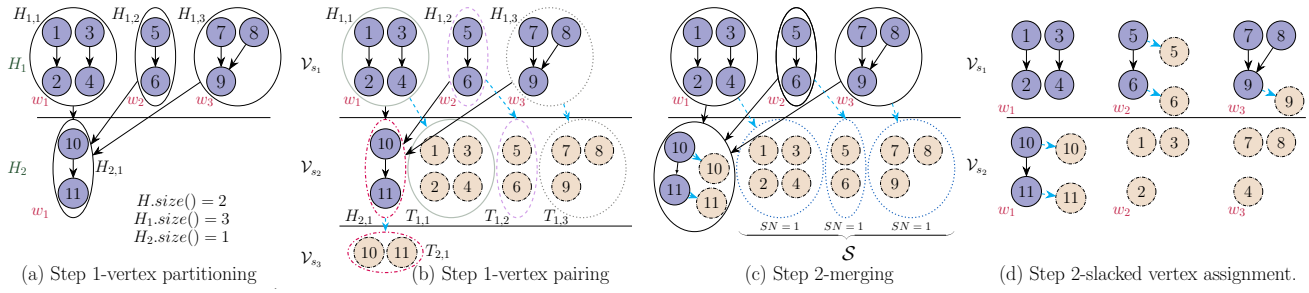
*3.2.2 Merging and Slack Vertex Assignment.* The second step of MSP reduces the number of synchronizations by merging some of the pair partitions in a *merging* phase. It also improves load balance by dispersing vertices across partitions using *slacked vertex assignment.*

*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 having to move its dependent vertices to later wavefronts. $SN(v)$ is the slack number of $v$ and is defined as $SN(v) = P_G - l(v) - height(v)$ where $height(v)$ is the maximum path from a vertex $v$ to a sink vertex (a sink vertex is a vertex without any 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, MSP iterates over partitions 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 16 of Algorithm 1, all partition slack numbers of $\mathcal{V}$ are computed via `slack_info` and are stored in $\mathcal{S}$. For example, because vertices in $\mathcal{V}_{s_2,w_3}$ can be postponed one wavefront, from s-partition 2 to 3, their slack number is 1. Vertices in w-partitions $\mathcal{V}_{s_2,w_1}$ and $\mathcal{V}_{s_3,w_1}$ can not be moved because their slack numbers are zero.

*Merging.* MSP finds pair partitions with partition slack number of zero and then merges them as shown in lines 17-19. 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 $\mathcal{V}.pairs$ and merges them using the `merge` function in line 18 if their slack numbers are zero, i.e. $SN(w) = 0$ and $SN(w') = 0$. The resulting merged partition is stored in $\mathcal{V}$ in place of the w-partition with the smaller s-partition number.

*Slacked vertex assignment.* The algorithm then uses slacked vertex assignment to approximately load balance the w-partitions of an s-partition using a cost model. The cost of w-partition $w \in \mathcal{V}_{s_i}$ is defined as $cost(w) = \sum_{v \in w} c(v)$. 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.

MSP first removes all slacked vertices $\mathcal{S}$ from the fused partitioning $\mathcal{V}$ in line 20. It then goes over every s-partition $i$ and balances $\mathcal{V}_{s_i}$ by assigning a slacked vertex to its imbalanced w-partition. Function `balance_with_slack` in line 22 balances each partition using two different strategies. The first strategy uses slack vertices of the pair partition. The second strategy uses a slack vertex $v_l \in \mathcal{S}$ from any other partition that satisfies the following condition $l(v_l) < i < (l(v_l) + SN(v_l))$. Slack vertices in $\mathcal{S}$ that depend on each other are dispersed as a group to the same w-partition for correctness. In line 23, slacked vertices in $\mathcal{S}$ that are not postponed to later s-partitions are evenly divided between the w-partitions of the current s-partition ($\mathcal{V}_{s_i}$) using the `assign_even` function.




(a) Step 1-vertex partitioning (b) Step 1-vertex pairing (c) Step 2-merging (d) Step 2-slacked vertex assignment.

**Figure 5: Stages of MSP for DAGs $G_1$ and $G_2$ and matrix $F$ in the running example shown in Figure 2b where the reuse ratio (*reuse_ratio*) is smaller than one and number of processors ($r$) is three. The first step of the algorithm selects $G_1$ and creates $H$ partitioning for three processors using the LBC algorithm as shown in Figure 5a. Then it pairs each $H_{i,j}$ through dependencies in matrix $F$ to create partitioning $T$ of $G_2$ as shown in Figure 5b. The partitions with the same line pattern/color are pair partitions. In the second step, MSP merges pair partitions that cannot be dispersed such as first w-partitions of s-partitions 2 and 3 ($\mathcal{V}_{s_3,w_1}$ and $\mathcal{V}_{s_2,w_1}$) in Figure 5b, these are merged into $\mathcal{V}_{s_2,w_1}$ in Figure 5c. Slack vertices, which are denoted as $\mathcal{S}$ are shown with blue dotted circles in Figure 5c. Slack vertices are assigned into imbalanced w-partitions as shown in Figure 5d. Since the reuse ratio is smaller than one, vertices inside each partition are packed separately as shown in Figure 2e.**

***Example.*** Figure 5d shows the output of the second step of MSP from the partitioning in Figure 5b. First pair partitions ($\mathcal{V}_{s_2,w_1}, \mathcal{V}_{s_3,w_1}$), shown with red dash-dotted circles in Figure 5b, are merged because their slack numbers are zero. The resulting merged partition is placed in $\mathcal{V}_{s_2,w_1}$ to reduce synchronization as shown in Figure 5c. Then slacked vertex assignment balances the w-partitions in Figure 5c. The balanced partitions are shown in Figure 5d. The slacked vertices, $S$, are shown with dotted blue circles in Figure 5c. The w-partitions in $\mathcal{V}_{s_1}$ are balanced using vertices of their pair partitions, e.g. the yellow dash-dotted vertices 5 and 6 are moved to $w_2$ in $\mathcal{V}_{s_1}$ as shown in Figure 5d. The second strategy in `balance_with_slack` is used to balance partitions in $\mathcal{V}_{s_2}$. This is because the vertices in $S$ do not belong to the pair partitions of the w-partitions in $\mathcal{V}_{s_2}$. However, since the slack vertices in $S$ can execute in either s-partition two or three because they are from s-partition one and have a slack number of one, they are used to balance the w-partitions in $\mathcal{V}_{s_2}$.

*3.2.3 Packing.* The third step of MSP reorders the vertices inside a w-partition to improve data locality for a thread within each kernel or between the two kernels. The previous steps of the algorithm create w-partitions that are composed of vertices of one or both kernels however the order of execution is not defined. Using the reuse ratio, the order at which the nodes in a w-partition should be executed is determined with a packing strategy. MSP has two packing strategies: *(i)* in interleaved packing, the vertices of the two DAGs in a w-partition are interleaved for execution and *(ii)* in separated packing the vertices of each kernel are executed separately. Interleaved packing improves temporal locality between kernels while separated packing enhances spatial and temporal locality within kernels. When the reuse ratio is greater than one, in line 25 of Algorithm 1 function `interleaved_pack` is called to interleave iterations of the two kernels based on F. Otherwise, `separated_pack` is called (line 26) to pack iterations of each kernel separately.

***Example.*** Figure 2e shows the output of MSP's third step from the partitioning in Figure 5d. Since the reuse ratio is smaller than one separated packing is chosen thus $\mathcal{V}_{s_2,w_1}$ is stored as $\mathcal{V}_{s_2,w_1} = \{[\underline{10}, \underline{11}, 10, 11]\}$. Vertices are ordered to keep dependent iterations of SpTRSV and consecutive iterations SpMV next to each other.

## 3.3 Fusing More than Two Loops

The MSP algorithm processes one DAG 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, MSP processes their DAGs in the order they appear in the code. The first two DAGs are fused as described in Section 3.2. When a loop with carried dependence is fused with another loop, it will always result in a loop-carried dependence. Thus, for any additional DAG after the second DAG, MSP uses forward pairing to partition its DAG. Forward pairing uses the final partitioned fused schedule of the previous DAG as the new head and the additional DAG as a tail. In the second step, MSP finds slacked partitions and applies merging and slack

**Table 1: The list of sparse matrices.**

| ID | Name | Rank ($10^3$) | Nonzeros ($10^6$) | Nonzero per row |
|---|---|---|---|---|
| 1 | Flan_1565 | 1,564 | 117.4 | 119 |
| 2 | bone010 | 986 | 71.7 | 48 |
| 3 | Hook_1498 | 1498 | 59.4 | 39 |
| 4 | af_shell10 | 1508.1 | 52.3 | 34 |
| 5 | Emilia_923 | 923 | 40.4 | 43 |
| 6 | StocF-1465 | 1498 | 21.1 | 14 |
| 7 | af_0_k101 | 503 | 17.5 | 34 |
| 8 | ted_B_unscal | 10 | 0.14 | 13 |

vertex assignment as described in Section 3.2.2. And finally, vertices inside each w-partition are sorted based on MSP packing strategies. It is important to note that not all loops are profitable for merging. For two loops, when fusion is not profitable because of redundant computation, MSP disables fusion. However, for more than two loops, the decision of which loops to fuse and in what order is more complex as it depends on the kernel types, inputs, and the application. When the profitable loops for fusion are identified (e.g. via profiling, a search, practitioner input), MSP can be used to deliver noticeable speedups via fusion. We will discuss the efficiency of MSP 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 [54] and ParSy [7], 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 create a schedule for execution using the created partitioning, the methods will be referred to as fused LBC, fused DAGP, and fused wavefront in order.

### 4.1 Setup.

The set of symmetric positive definite matrices listed in Table 1 are used for experimental results. The matrices are from [11] and with real values in double precision. The testbed architecture is a multicore processor with 24 cores of a Xeon Platinum8160 processor with 33MB L3 cache. All generated codes, implementations of different approaches, and library drivers are compiled with GCC v.7.2.0 compiler and with the `-O3` flag. Each thread is pinned to physical code and a close thread binding is selected. Matrices are first reordered with METIS [22] to improve 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 [10] or chordal DAGs. Thus, we make DAGs chordal before using LBC. *II. MKL* uses Intel MKL [54] routines with MKL 2021.1.0 and calls them separately for each kernel. We use inspector executor version of MKL through calling `mkl_sparse_set_sv_hint` and `mkl_sparse_set_mv_hint` for inspection. For executor of SpTRSV, SpMV, and SpILU0 we use `mkl_sparse_d_trsv`, `mkl_sparse_d_mv`, and `dcsrilu0`, respectively.

Sparse fusion is also compared to three fused approaches all of which take as input the *joint DAG*; the joint DAG is created from 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 DAGP 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 three fused approaches use sparse fusion's packing.

The list of sparse kernel combinations investigated is in Table 2. 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 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 [3, 60], GMRES [8], Gauss-Seidel [39]. Preconditioned Krylov methods [16] and Newton solvers [43] frequently use kernel combinations 3, 5, 6, 7. The s-step Krylov solvers [5] and s-step optimization methods used in machine learning [43] provide even more opportunities to interleave iterations. Thus, they use these kernel combinations significantly more than their classic formulations.

**Table 2: 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.**

| ID | Kernel combination | Operations | Dependency DAGs | Reuse Ratio |
|---|---|---|---|---|
| 1 | SpTRSV CSR - SpTRSV CSR | $x = L^{-1}y, z = L^{-1}x$ | CD - CD | $\frac{2n+2size_L}{max(2n+size_L,size_L+2n)} \geq 1$ |
| 2 | SpMV CSR - SpTRSV CSR | $y = Ax, z = L^{-1}y$ | Parallel - CD | $\frac{2n}{max(2n+size_L,size_A+2n)} < 1$ |
| 3 | DSCAL CSR - SpILU0 CSR | $LU \approx DAD^T$ | Parallel - CD | $\frac{2size_A}{max(size_A,size_A+2n)} \geq 1$ |
| 4 | SpTRSV CSR - SpMV CSC | $y = L^{-1}x, z = Ay$ | CD - Parallel | $\frac{2n}{max(2n+size_L,size_A+2n)} < 1$ |
| 5 | SpIC0 CSC - SpTRSV CSC | $LL^T \approx A, y = L^{-1}x$ | CD - CD | $\frac{2size_L}{max(size_L,size_L+2n)} \geq 1$ |
| 6 | SpILU0 CSR - SpTRSV CSR | $LU \approx A, y = L^{-1}x$ | CD - CD | $\frac{2size_A}{max(size_A,size_L+2n)} \geq 1$ |
| 7 | DSCAL CSC - SpIC0 CSC | $LL^T \approx DAD^T$ | Parallel - CD | $\frac{2size_L}{max(size_L,size_L+2n)} \geq 1$ |

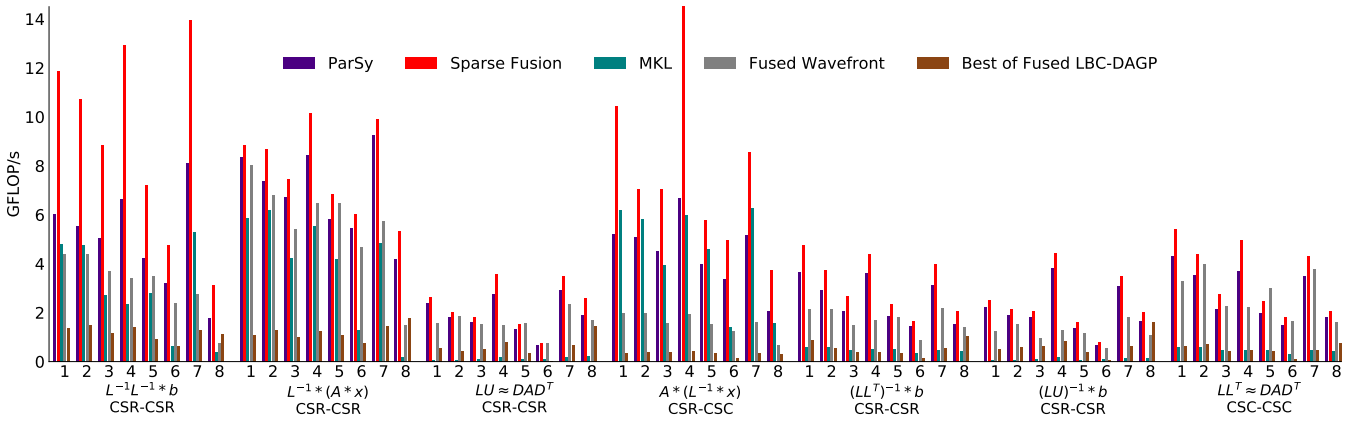



**Figure 6: Performance of different implementations shown in GFLOPs per second.**

## 4.2 Sparse Fusion's Performance

Figure 6 shows the performance of the fused code from sparse fusion, the unfused implementation from ParSy and MKL, and the fused wavefront, fused LBC, and fused DAGP implementations. Performance of implementations are shown in floating point operations per second (GFLOP/s). The sparse fusion's fused code is on average 1.4× faster than ParSy's executor code and 4.3× faster than MKL across all kernel combinations. 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. The fused code from sparse fusion is on average 2.4×, 5.1×, and 8.6× faster than in order fused wavefront, fused LBC, and fused DAGP. Obtained speedups of sparse fusion over ParSy (the fastest unfused implementation) for SpILU0-SpTRSV and SpIC0-SpTRSV is lower than other kernel combinations. Because SpIC0 and SpILU0 have a high execution time, when combined with others sparse kernels with a noticeably lower execution time, the realized speedup from fusion will not be significant.

## 4.3 Locality in Sparse Fusion

Figure 7 shows the efficiency of the two packing strategies to improve locality. The effect of the packing strategy is shown for kernel combinations with a reuse ratio smaller and larger than one as shown in Table 2. Kernel combinations 1, 3, 5, 6, and 7 share the sparse matrix $L$ and thus have a reuse ratio larger than one while combination 2 and 4 only share vector $y$ leading to a reuse ratio lower than one. Figure 7 shows the range of speedup over all matrices for the selected packing strategy versus the other packing method for each combination. As shown, the selected packing strategy in sparse fusion improves the performance in 88% of kernel combinations and matrices and provides 1-3.9× improvement in both categories.

Figure 8 shows the average memory access latency [17] 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 example, other matrices exhibit similar behavior). The average memory access latency is used as a proxy for locality

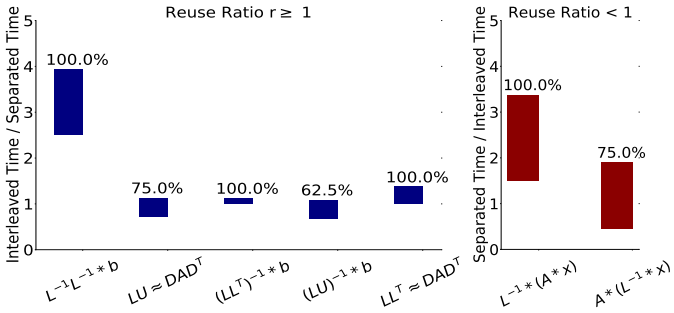



**Figure 7: The range of speedup for all matrices achieved as a result of using interleaved packing vs. separated packing. The labels on bars show how often the choice of packing strategy made by sparse fusion leads to performance improvement.**

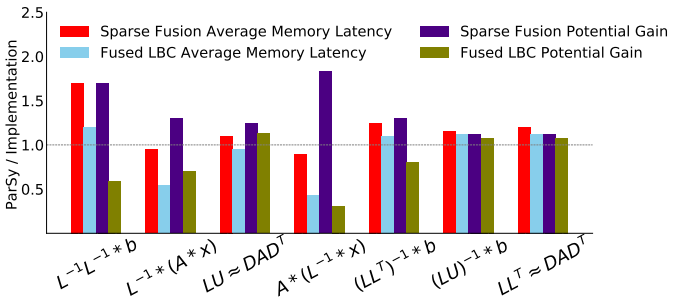



**Figure 8: Average memory access time and the OpenMP potential gain for matrix *bone010*. The legends show the implementation, values are normalized over ParSy.**

and is computed using the number of accesses to L1, LLC, and TLB measured with PAPI performance counters [50]. We use the measured numbers to calculate cache miss ratio in L1, L2, and L3 caches. Then, we multiply the calculated miss ratios with total number of accesses to each level and the access cost of each level of cache to measure average memory access latency.

For kernels 1, 3, 5, 6, and 7 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 kernels 2 and 4 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 sparse fusion and ParSy. Sparse fusion and ParSy both improve data locality within each kernel for these kernel combinations.

## 4.4 Load Balance and Synchronization in Sparse Fusion

Figure 8 shows the OpenMP potential gain [42] of sparse fusion, ParSy, and Fused LBC for all kernel combinations normalized over ParSy's potential gain (shown for matrix *bone010* as example, but all other matrices in Table 1 follow similar behavior.) The OpenMP potential gain is a metric in Vtune [63] 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 combinations 2 and 4 have slack vertices that provide opportunities to balance workloads. For example, for matrices shown in Table 1, between 35-76% vertices can be slacked thus the potential gain balance of ParSy is 1.6× larger than sparse fusion and 2.4× lower than fused LBC. The effect of slack variables is also visible by comparing kernel combinations 2 and 4 where the number of slack vertices in kernel combination 4 is on average 1.8× larger than kernel combination 2 that leads to a noticeable performance difference shown in Figure 6. This is also due to redundant SpMV iterations created in kernel combination 2, which is between 12%-44% (2k-312k) of number of vertices in Table 1. ParSy can only improve load balance using the workloads of an individual kernel. As shown in Figure 1, for the kernel combination 5, the joint DAG has a small number of parallel iterations in final wavefronts that makes the final s-partitions of the LBC fused implementation imbalanced (a similar trend exists for kernel combination 6). 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, 4, and 7 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 50% compared to that of ParSy.

## 4.5 Inspector Time

Figure 9 shows the number of times that the executor should run to amortize the cost of inspection for implementations that have an inspector. For space only combinations 1, 3, 4, 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 $\frac{Inspector\ Time}{Baseline\ Time - Executor\ Time}$. The *baseline* time is obtained by running each kernel individually with a sequential implementation, the inspector and executor times belong to the

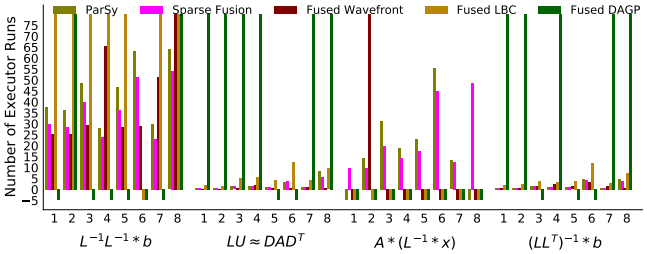



**Figure 9: The number of executor runs to amortize inspector cost. Values are clipped between -5 and 80. (lower is better)**

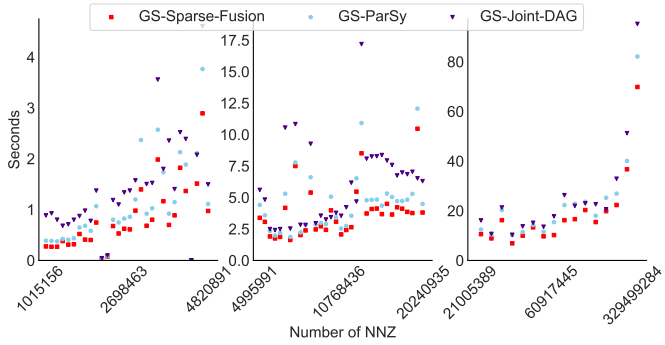



**Figure 10: Performance of Guass-Seidel (GS) using unfused (GS-ParSy) and fused implementations, i.e. sparse fusion (GS-Sparse-Fusion) and best of joint DAG methods (GS-Joint-DAG) for matrices of different nonzeros. (lower is better)**

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 and fused wavefront have the lowest NER amongst 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 SpILU0-TRSV only need one iteration to amortize the inspection time and SpTRSV-SpMV, SpTRSV-SpTRSV, 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 ten of thousands of iterations [3, 23, 34], hence amortizing the overhead of inspection.

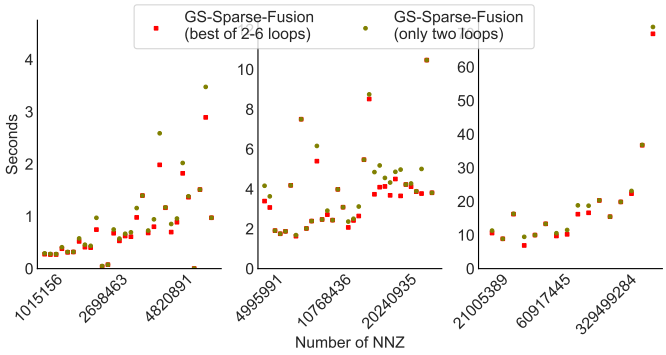



**Figure 11: Performance of Guass-Seidel (GS) using sparse fusion (GS-Sparse-Fusion) for two loops and for the best among fusing 2–6 loops. (lower is better)**

## 4.6 Gauss-Seidel, a case study for more than two loops.

To demonstrate the efficiency of Sparse Fusion in merging more than two loops, we use Gauss-Seidel (GS) [39] 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 [39] 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 in order diagonal, lower triangular, and upper triangular matrices with decomposition of $A = D - F - E$. 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 [5].

To guarantee convergence in GS, we select all 79 SPD matrices with larger than one million nonzero elements from SuiteSparse matrix repository [11]. 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 fusion of 2–6 loops and select the lowest timing. Figure 10 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.

Figure 11 compares the performance of Gauss-Seidel using sparse fusion when the number of fused loops is two to when 2–6 loops are fused. In Figure 11, in 68% of matrices, sparse fusion provides a better performance when more than two loops are fused. Sparse fusion when more than two loops are fused is up to 1.4× faster than sparse fusion with only fusing two loops. The average speedup of fusing more than two loops is 1.1×.

## 4.7 Limitation of joint-DAG

Figure 12 compares the performance of two DAG partitioners, DAGP and LBC for different sizes of sparse DAGs. In the one-DAG configuration, the DAG paritioner 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. Both configurations run on all SPD matrices larger than 1 million nonzero from the SuiteSparse collection [11]. 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 edges of the SpTRSV DAG. As shown in the Figure 12, the DAGP in both one DAG and joint DAG configurations are 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).

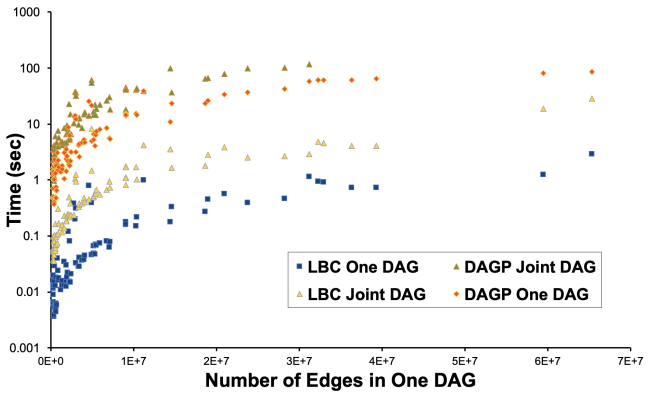



**Figure 12: Performance of DAGP and LBC DAG partitioners for DAGs with different number of edges in an individual and joint DAG.**

# 5 RELATED WORK

Parallel implementations of individual sparse matrix kernels exist in both highly-optimized libraries [18, 27] and inspector-executor approaches [6, 31, 48]. Some libraries such as MKL [54], and code generators such as ReACT [59] and SparseLNR [13] provide a domain specific compiler that can fuse parallel loops in tensor expression, while others provide optimizations for a specific sparse kernel. For example, the sparse triangular solve has been optimized in [26, 33, 35, 36, 41, 51, 53, 55, 57], optimizations of sparse matrix-vector multiply are available in [21, 25, 28, 30, 56], and LU and Cholesky factorization have been optimized in SuperLU [27] and Pastix [18].

Inspector-executor frameworks commonly use wavefront parallelism [15, 33, 37, 46, 52, 62] to parallelize sparse matrix computations with loop-carried dependencies. Recently, task coarsening approaches such as LBC [7], HDagg [58], and DAGP [19] 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 inspector-executor frameworks provide parallel implementations of fused sparse kernels with no loop-carried dependencies such as, two or more SpMV kernels [2, 20, 29, 32, 38] or SpMV and dot products [1, 2, 12, 14, 38, 61]. The formulation of *s*-step Krylov solvers [5] has enabled iterations of iterative solvers to be interleaved and hence multiple SpMV kernels are optimized simultaneously via replicating computations to minimize communication costs [20, 29, 32, 43]. Sparse tiling [24, 45–47, 49] is an inspector executor approach that uses manually written inspectors [45, 47] to group iteration of different loops of a specific kernel such as Gauss-Seidel [47] and Moldyn [45] and is generalized for parallel loops without loop-carried dependencies [24, 49]. 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 MSP algorithm to balance the workload between wavefronts and determine whether to optimize data locality for within or between the kernels. Sparse fusion's generated code outperforms state-of-the-art implementations for sparse matrix optimizations.

# REFERENCES

[1] Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov. 2009. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. In *Journal of Physics: Conference Series*, Vol. 180. IOP Publishing, 012037.

[2] José I Aliaga, Joaquín Pérez, and Enrique S Quintana-Ortí. 2015. Systematic fusion of CUDA kernels for iterative sparse linear system solvers. In *European Conference on Parallel Processing*. Springer, 675–686.

[3] Michele Benzi, Jane K Cullum, and Miroslav Tuma. 2000. Robust approximate inverse preconditioning for the conjugate gradient method. *SIAM Journal on Scientific Computing* 22, 4 (2000), 1318–1332.

[4] Stephen Boyd, Stephen P Boyd, and Lieven Vandenberghe. 2004. *Convex optimization*. Cambridge university press.

[5] Erin Claire Carson. 2015. *Communication-avoiding Krylov subspace methods in theory and practice*. Ph.D. Dissertation. UC Berkeley.

[6] Kazem Cheshmi, Shoaib Kamil, Michelle Mills Strout, and Maryam Mehri Dehnavi. 2017. Sympiler: transforming sparse matrix codes by decoupling symbolic analysis. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13.

[7] Kazem Cheshmi, Shoaib Kamil, Michelle Mills Strout, and Maryam Mehri Dehnavi. 2018. ParSy: inspection and transformation of sparse matrix computations for parallelism. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 779–793.

[8] Kazem Cheshmi, Danny M Kaufman, Shoaib Kamil, and Maryam Mehri Dehnavi. 2020. NASOQ: numerically accurate sparsity-oriented QP solver. *ACM Transactions on Graphics (TOG)* 39, 4 (2020), 96–1.

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

[10] Timothy A Davis. 2006. *Direct methods for sparse linear systems*. SIAM.

[11] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1.

[12] Maryam Mehri Dehnavi, David M Fernández, and Dennis Giannacopoulos. 2011. Enhancing the performance of conjugate gradient solvers on graphic processing units. *IEEE Transactions on Magnetics* 47, 5 (2011), 1162–1165.

[13] Adhitha Dias, Kirshanthan Sundararajah, Charitha Saumya, and Milind Kulkarni. 2022. SparseLNR: accelerating sparse tensor computations using loop nest restructuring. In *Proceedings of the 36th ACM International Conference on Supercomputing*. 1–14.

[14] Pieter Ghysels and Wim Vanroose. 2014. Hiding global synchronization latency in the preconditioned conjugate gradient algorithm. *Parallel Comput.* 40, 7 (2014), 224–238.

[15] R Govindarajan and Jayvant Anantpur. 2013. 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)*. IEEE Computer Society, 1–10.

[16] Laura Grigori and Sophie Moufawad. 2015. Communication avoiding ILU0 preconditioner. *SIAM Journal on Scientific Computing* 37, 2 (2015), C217–C246.

[17] John L Hennessy and David A Patterson. 2017. *Computer architecture: a quantitative approach*. Elsevier.

[18] Pascal Hénon, Pierre Ramet, and Jean Roman. 2002. PASTIX: a high-performance parallel direct solver for sparse symmetric positive definite systems. *Parallel Comput.* 28, 2 (2002), 301–321.

[19] Julien Herrmann, M Yusuf Ozkaya, Bora Uçar, Kamer Kaya, and Ümit VV Çatalyürek. 2019. Multilevel algorithms for acyclic partitioning of directed acyclic graphs. *SIAM Journal on Scientific Computing* 41, 4 (2019), A2117–A2145.

[20] Mark Frederick Hoemmen et al. 2010. Communication-avoiding Krylov subspace methods. (2010).

[21] Sam Kamin, María Jesús Garzarán, Barış Aktemur, Danqing Xu, Buse Yılmaz, and Zhongbo Chen. 2014. Optimization by runtime specialization for sparse matrix-vector multiplication. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 93–102.

[22] George Karypis and Vipin Kumar. 1998. A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. *University of Minnesota, Department of Computer Science and Engineering, Army HPC Research Center, Minneapolis, MN* (1998).

[23] David S Kershaw. 1978. The incomplete Cholesky-conjugate gradient method for the iterative solution of systems of linear equations. *Journal of computational physics* 26, 1 (1978), 43–65.

[24] Christopher D Krieger, Michelle Mills Strout, Catherine Olschanowsky, Andrew Stone, Stephen Guzik, Xinfeng Gao, Carlo Bertolli, Paul HJ Kelly, Gihan Mudalige, Brian Van Straalen, et al. 2013. Loop chaining: A programming abstraction for balancing locality and parallelism. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*. IEEE, 375–384.

[25] Jiajia Li, Guangming Tan, Mingyu Chen, and Ninghui Sun. 2013. SMAT: an input adaptive auto-tuner for sparse matrix-vector multiplication. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. 117–126.

[26] Ruipeng Li and Yousef Saad. 2013. GPU-accelerated preconditioned iterative linear solvers. *The Journal of Supercomputing* 63, 2 (2013), 443–466.

[27] Xiaoye S Li. 2005. An overview of SuperLU: Algorithms, implementation, and user interface. *ACM Transactions on Mathematical Software (TOMS)* 31, 3 (2005), 302–325.

[28] Changxi Liu, Biwei Xie, Xin Liu, Wei Xue, Hailong Yang, and Xu Liu. 2018. Towards efficient SpMV on sunway manycore architectures. In *Proceedings of the 2018 International Conference on Supercomputing*. 363–373.

[29] M. MehriDehnavi, Y. El-Kurdi, J. Demmel, and D. Giannacopoulos. 2013. Communication-Avoiding Krylov Techniques on Graphic Processing Units. *IEEE Transactions on Magnetics* 49, 5 (2013), 1749–1752. https://doi.org/10.1109/TMAG.2013.2244861

[30] Duane Merrill and Michael Garland. 2016. Merge-based parallel sparse matrix-vector multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 58.

[31] Mahdi Soltan Mohammadi, Tomofumi Yuki, Kazem Cheshmi, Eddie C Davis, Mary Hall, Maryam Mehri Dehnavi, Payal Nandy, Catherine Olschanowsky, Anand Venkat, and Michelle Mills Strout. 2019. Sparse computation data dependence simplification for efficient compiler-generated inspectors. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 594–609.

[32] M. Mohiyuddin, M. Hoemmen, J. Demmel, and K. Yelick. 2009. Minimizing communication in sparse matrix solvers. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. 1–12. https://doi.org/10.1145/1654059.1654096

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

[34] M Papadrakakis and N Bitoulas. 1993. Accuracy and effectiveness of preconditioned conjugate gradient algorithms for large and ill-conditioned problems. *Computer methods in applied mechanics and engineering* 109, 3-4 (1993), 219–232.

[35] Jongsoo Park, Mikhail Smelyanskiy, Narayanan Sundaram, and Pradeep Dubey. 2014. Sparsifying Synchronization for High-Performance Shared-Memory Sparse Triangular Solver. In *Proceedings of the 29th International Conference on Supercomputing - Volume 8488 (ISC 2014)*. Springer-Verlag New York, Inc., New York, NY, USA, 124–140.

[36] A. Picciau, G. E. Inggs, J. Wickerson, E. C. Kerrigan, and G. A. Constantinides. 2016. Balancing Locality and Concurrency: Solving Sparse Triangular Systems on GPUs. In *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*. 183–192.

[37] Lawrence Rauchwerger, Nancy M Amato, and David A Padua. 1995. Run-time methods for parallelizing partially parallel loops. In *Proceedings of the 9th international conference on Supercomputing*. 137–146.

[38] Karl Rupp, Philippe Tillet, Florian Rudolf, Josef Weinbub, Andreas Morhammer, Tibor Grasser, Ansgar Jungel, and Siegfried Selberherr. 2016. ViennaCL—linear algebra library for multi-and many-core architectures. *SIAM Journal on Scientific Computing* 38, 5 (2016), S412–S439.

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

[40] Yousef Saad and Andrei V Malevsky. 1995. P-Sparslib: a portable library of distributed memory sparse iterative solvers. In *Proceedings of Parallel Computing Technologies (PaCT-95), 3-rd international conference, St. Petersburg*. Citeseer.

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

[42] Intel Software. 2018. *OpenMP potential gain definition in intel VTune*. https://software.intel.com/content/www/us/en/develop/documentation/vtune-help/top/reference/cpu-metrics-reference/openmp-potential-gain.html

[43] Saeed Soori, Aditya Devarakonda, Zachary Blanco, James Demmel, Mert Gurbuzbalaban, and Maryam Mehri Dehnavi. 2018. Reducing communication in proximal Newton methods for sparse least squares problems. In *Proceedings of the 47th International Conference on Parallel Processing*. 1–10.

[44] Bartolomeo Stellato, Goran Banjac, Paul Goulart, Alberto Bemporad, and Stephen Boyd. 2020. OSQP: An operator splitting solver for quadratic programs. *Mathematical Programming Computation* (2020), 1–36.

[45] Michelle Mills Strout, Larry Carter, and Jeanne Ferrante. 2003. Compile-time composition of run-time data and iteration reorderings. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*. 91–102.

[46] Michelle Mills Strout, Larry Carter, Jeanne Ferrante, Jonathan Freeman, and Barbara Kreaseck. 2002. Combining performance aspects of irregular gauss-seidel via sparse tiling. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 90–110.

[47] Michelle Mills Strout, Larry Carter, Jeanne Ferrante, and Barbara Kreaseck. 2004. Sparse tiling for stationary iterative methods. *The International Journal of High Performance Computing Applications* 18, 1 (2004), 95–113.

[48] Michelle Mills Strout, Mary Hall, and Catherine Olschanowsky. 2018. The sparse polyhedral framework: Composing compiler-generated inspector-executor code. *Proc. IEEE* 106, 11 (2018), 1921–1934.

[49] Michelle Mills Strout, Fabio Luporini, Christopher D Krieger, Carlo Bertolli, Gheorghe-Teodor Bercea, Catherine Olschanowsky, J Ramanujam, and Paul HJ Kelly. 2014. Generalizing run-time tiling with the loop chain abstraction. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 1136–1145.

[50] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. 2010. Collecting performance data with PAPI-C. In *Tools for High Performance Computing 2009*. Springer, 157–173.

[51] Ehsan Totoni, Michael T Heath, and Laxmikant V Kale. 2014. Structure-adaptive parallel solution of sparse triangular linear systems. *Parallel Comput.* 40, 9 (2014), 454–470.

[52] Anand Venkat, Mahdi Soltan Mohammadi, Jongsoo Park, Hongbo Rong, Rajkishore Barik, Michelle Mills Strout, and Mary Hall. 2016. Automating wavefront parallelization for sparse matrix computations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 41.

[53] Richard Vuduc, Shoaib Kamil, Jen Hsu, Rajesh Nishtala, James W Demmel, and Katherine A Yelick. 2002. Automatic performance tuning and analysis of sparse triangular solve. ICS.

[54] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. 2014. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi™*. Springer, 167–188.

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

[56] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. 2009. Optimization of sparse matrix–vector multiplication on emerging multicore platforms. *Parallel Comput.* 35, 3 (2009), 178–194.

[57] Buse Yılmaz, Buğrra Sipahioğrlu, Najeeb Ahmad, and Didem Unat. 2020. 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*. 188–198.

[58] Behrooz Zarebavani, Kazem Cheshmi, Bangtian Liu, Michelle Mills Strout, and Maryam Mehri Dehnavi. 2022. HDagg: hybrid aggregation of loop-carried dependence iterations in sparse matrix computations. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 1217–1227.

[59] Tong Zhou, Ruiqin Tian, Rizwan A Ashraf, Roberto Gioiosa, Gokcen Kestor, and Vivek Sarkar. 2022. ReACT: Redundancy-Aware Code Generation for Tensor Expressions. (2022).

[60] Sicong Zhuang and Marc Casas. 2017. Iteration-fusing conjugate gradient. In *Proceedings of the International Conference on Supercomputing*. 1–10.

[61] Sicong Zhuang and Marc Casas. 2017. Iteration-Fusing Conjugate Gradient. In *Proceedings of the International Conference on Supercomputing* (Chicago, Illinois) *(ICS '17)*. Association for Computing Machinery, New York, NY, USA, Article 21, 10 pages. https://doi.org/10.1145/3079079.3079091

[62] Xiaotong Zhuang, Alexandre E Eichenberger, Yangchun Luo, Kevin O'Brien, and Kathryn O'Brien. 2009. Exploiting parallelism with dependence-aware scheduling. In *Parallel Architectures and Compilation Techniques, 2009. PACT'09. 18th International Conference on*. IEEE, 193–202.

[63] Intel Developer Zone. [n.d.]. Intel VTune Amplifier, 2017. *Documentation at the URL: https://software.intel.com/en-us/intel-vtune-amplifier-xe-support/documentation* ([n. d.]).