Loop Fusion in Matrix Multiplications with Sparse Dependence

Mohammad Mahdi Salehi
McMaster University
Hamilton, Canada
salehm32@mcmaster.ca

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

Abstract

Fusing parallel loops in consecutive matrix multiplications presents an opportunity for data locality optimization. However, irregular dependencies between iterations across the loops hinder existing compilers from performing this fusion. It also poses challenges for runtime methods, leading to excessive synchronization overhead or limited data reuse. This paper introduces tile fusion, a compiler approach that fuses tiles from the two parallel loops of matrix multiplications with sparse dependence between them. By enhancing data locality and providing balanced workloads, tile fusion accelerates graph neural network training and the solution of sparse linear systems, achieving geometric mean speedups of 2.33× over PyG and 1.32× over MKL, respectively.

CCS Concepts

• **Software and its engineering** → *Domain specific languages.*

Keywords

Loop Transformation, Loop Fusion, Runtime Schedulers, Graph Neural Network

ACM Reference Format:
Mohammad Mahdi Salehi and Kazem Cheshmi. 2025. Loop Fusion in Matrix Multiplications with Sparse Dependence. In *2025 International Conference on Supercomputing (ICS '25), June 08–11, 2025, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3721145.3730427

ACM ISBN 979-8-4007-1537-2/25/06
https://doi.org/10.1145/3721145.3730427

1 Introduction

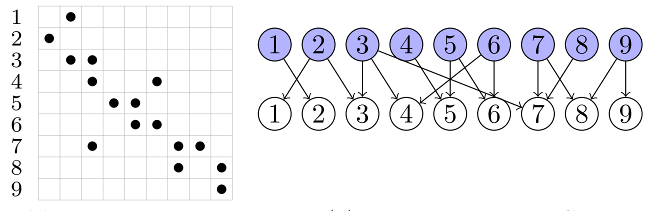
Matrix multiplications involving dense and sparse data structures can be represented as parallel loops, where each iteration executes independently without compromising correctness. These operations are fundamental to numerous scientific and machine learning applications, including iterative linear solvers with multiple right-hand sides [31], Jacobi iterations [50], power methods [44], Graph Neural Networks (GNNs) [19, 51, 56, 58], and sparse neural networks [47]. These operations often share data through matrix exchanges. Loop fusion is an optimization technique that exploits data locality by combining operations that share data. However, loop fusion is not always feasible when sparse data structures are involved. Consider, for example, a dense matrix-matrix multiplication, $D_1 = BC$, followed by a sparse matrix-dense matrix multiplication (SpMM), $D = AD_1$, where A is a sparse $n \times n$ matrix, B is a dense $n \times bCol$ matrix, and C is a dense $bCol \times cCol$ matrix. Reusing D_1 is challenging because the compact indexing used in sparse matrix A introduces non-affine dependencies between the two computations.

The mainstream approach in existing numerical libraries such as MKL [57] or machine learning frameworks such as PyTorch Geometric (PyG) [19] and Deep Graph Library (DGL) [58] is to optimize each operation separately. For example, $D_1 = BC$ and $D = AD_1$ map to an efficient General Matrix Multiplication (GeMM)-SpMM sequence. While these routines benefit from several locality and load balancing opportunities [25, 57] within each loop separately, potential locality opportunities across loops, e.g., reusing D_1, are ignored. In an efficient fusion strategy, it is essential to consider both within- and across-loop data reuse opportunities.

Fusing operations or loops is commonly used to eliminate intermediate matrices between two operations. Tensor compilers [18, 30, 41] generate fused code when A is sparse and B and C are dense. The generated code iterates over A and performs a general matrix-vector multiplication (GeMV) for each nonzero element of A. Although this approach eliminates the need to store intermediate results, i.e., D_1, it introduces redundant computation. Moreover, this method is inapplicable when both A and B are sparse due to irregular loop bounds at compile time.

Listing 1: GeMM-SpMM code, $D1 = B \times C$ and $D = A \times D1$

```
1  @njit(pipeline_class = TileFusion)
2  def gemm_spmm(A, B, C, n, bCol, cCol):
3   D1 = np.zeros((n, cCol))
4   D = np.zeros((n, cCol))
5   for i1 in fprange(n): # GeMM
6    for i2 in range(bCol):
7     for i3 in range(cCol):
8      D1[i1,i3] += B[i1,i2]*C[i2,i3]
9   for j1 in fprange(n): # SpMM
10   for j2 in range(A.p[j1], A.p[j1+1]):
11    for j3 in range(cCol):
12     D[j1,j3] += A.x[j2]*D1[A.i[j2],j3]
13  return D
```



(a) Sparse matrix A (b) Dependence graph G

Figure 1: Sparse matrix A in Figure 1a creates sparse dependence between iterations of GeMM (i1) and SpMM (j1) shown as purple and white vertices in Figure 1b.

Prior compiler approaches [32] and libraries [16] have leveraged sparsity information at runtime to fuse sparse matrix-vector multiplications (SpMV) and enable data reuse between operations. These methods model SpMV operations as iteration-directed acyclic graphs (DAGs), where vertices represent outermost loop iterations and edges denote dependencies between iterations. A scheduler then groups DAG vertices into tiles at runtime. Sparse tiling [32] employs barriers and atomic operations to maintain dependencies between tiles during parallel execution. Given the parallel nature of outermost loops in GeMM, SpMM, and SpMV, these techniques could potentially be adapted for fusing GeMM-SpMM and SpMM-SpMM. However, the synchronization or redundant computation cost of each fused iteration in both computations is proportional to *bCol* and *cCol*, exacerbating race conditions and wasting computing resources. Also, the data reuse opportunities across loops are higher in these computations, and an effective compromise is needed.

We propose sparsity-oriented tile fusion, or *tile fusion* for short, to fuse parallel loops with sparse dependencies. Tile fusion includes a transformation that fuses tiles of two parallel loops to enable locality within and across loops with sparse dependencies. To ensure tiles fit within fast memory and processors are balanced, a tile fusion scheduler builds variable

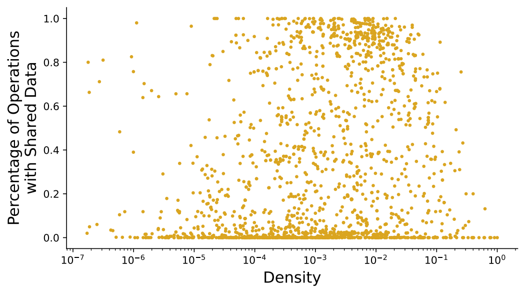


Figure 2: Potential computation and data reuse percentage for GEMM-SpMM for SuiteSparse matrices

tile sizes based on the sparsity pattern and enables partial safe fusion between tiles. Tile fusion has been thoroughly explored on different multicore CPUs and GPUs, as both architectures are highly relevant in numerical simulations and machine learning. Tile fusion demonstrates superior performance, achieving a geometric mean (GMean) speedup of 2.33× and 1.32× compared to unfused implementations in Graph Neural Networks (GNNs) and linear solvers, respectively. Furthermore, it outperforms the best unfused and fused baselines by GMean speedups of 1.55× and 3.52×, respectively, on SuiteSparse matrices.

2 Motivation and Overview

We use the code in Listing 1 to discuss challenges in fusing matrix multiplications and motivate tile fusion's compile-time and runtime strategies. Lines 5–8 execute GeMM, $D_1 = BC$, and lines 9–12 perform SpMM, $D = AD_1$. Iterations of loops `i1` and `j1` are independent, allowing for parallel execution. The two loops both operate on a shared matrix D_1, which, if reused, will improve locality. To enable this reuse and improve locality, loops `i1` and `j1` should be fused. However, fusing loops `i1` and `j1` is problematic because each iteration in `j1` depends on a variable number of `i1` iterations, and this number is not known until runtime. Due to this, each loop is often optimized separately and mapped to efficient GeMM and SpMM implementations.

If the accesses to `D1[A.i[j2],j3]` in line 12 in Listing 1 are known, then multiple iterations of the first loop can be fused with iterations of the second loop. Since `A.i[j2]` depends on the matrix sparsity pattern of A, we use an example sparse matrix, shown with its dense representation in Figure 1a. For example, if iterations 1–4 of loop `i1` execute, then iterations 1–3 of the loop `j1` can be executed, enabling reuse of a row of D_1. In the DAG in Figure 1b, this dependence is illustrated by four shared edges connecting purple vertices 1–4 and white vertices 1–3. This, however, requires the dependence between the two loops to be sparse to enable interleaving. Fortunately, sparse matrices exhibit this property. Figure 2 shows the percentage of operations with shared data that can be reused if the tile size is large enough. For this figure, a tile size of 2048 enables, on average, 34% of the computations of both loops to execute in one fused tile. However, defining fused tile size is crucial for balancing parallel workload and ensuring locality within and across loops.

Listing 2: Tile fusion overall view

```
1 // Matrices A, B, and C are constructed
2 G = DependenceDAGGeMMSpMM(A);//Sec 3.2
3 T = Scheduler(G, bCol, cCol, Arch, I, J);//
     Sec 4
4 // other operations ...
5 D = TileFusedGeMMSpMM(A,B,C,T);//Sec 3.1
```

To enable the fusion of iterations when sparse dependencies exist between loops, tile fusion employs a compile-time and runtime strategy. Listing 2 shows an overview of tile fusion. Tile fusion comprises a compile-time transformation and a runtime scheduler. Functions in lines 2 and 5 in Listing 2 show the DAG generation and fused code produced by the tile fusion transformation from the input code provided in Listing 1. The tile fusion runtime scheduler is a library call that takes a DAG of loop iterations and constructs a fused tiling schedule based on sparsity, architecture information, and the loops' iteration spaces (`I` and `J`). This schedule is then passed to the fused code to guide the order of iterations within the fused code. The generated schedule depends on the sparsity pattern of A and remains valid as long as this pattern is static.

3 Tile Fusion Transformation

Tile fusion is a compiler transformation that fuses two parallel loops even when there are non-affine accesses with sparse dependencies between iterations. This section discusses the tile fusion code lowering, its compiler implementation, and how the dependencies between fused iterations are satisfied.

Listing 3: Generic form of tile fusion code lowering.

```
1 for(w in 𝒯){ // for each tile group
2  parallel_for(t in 𝒯[w]){ // for each tile
3   for(i1 in t.first)
4    \\ first loop body
5   for(j1 in t.second)
6    \\ second loop body
7  }}
```

3.1 Code Lowering

The tile fusion code lowering fuses the two annotated loops with `fprange`. The fused code is generated by merging the outermost loops of the two computations, as depicted in Listing 3. As shown, the outermost loops are replaced with a doubly nested loop structure that iterates over a partial order stored in $\mathcal{T}$. The order of statements in the bodies of the loops remains unchanged. The partial order of fused iterations, $\mathcal{T}$, is computed using a runtime scheduler. Each loop iteration in Line 1 of Listing 3 corresponds to a barrier in CPUs and a kernel launch in GPUs. Each iteration of loop t (Line 2), internally annotated with `parallel_for`, is mapped to a thread in CPUs or a thread block in GPUs. Fused iterations in $\mathcal{T}$ are stored as `List<List<Pair<Set<int>, Set<int>>>>` where `first` and `second` are two sets of iterations mapped to the corresponding loop body at runtime. This enables data reuse between iterations of the two loops.

Listing 4: `TileFusedGeMMSpMM` in Listing 2.

```
1 for (w in T){ // for each wavefront
2  parallel_for(t in T[w]){ // for each tile
3   for(i1 in t.first)
4    for(int i2=0; i2<bCol; i2++)
5     for(int i3=0; i3<cCol; i3++)
6      D1[i1][i3] += B[i1][i2]*C[i2][i3];
7   for(j1 in t.second)
8    for(int j2=A.p[j1]; j2<A.p[j1+1]; j2++)
9     for(int j3=0; j3<cCol; j3++)
10     D[j1][j3] += A.x[j2]*D1[A.i[j2]][j3];
11 }}
```

Listing 4 presents an example of the transformation for the GeMM-SpMM loop pair shown in Listing 1. As illustrated, the fused code incorporates code segments for both operations. Lines 3–6 and 7–10 in Listing 4 represent the innermost loops for GeMM and SpMM, respectively. Loop bounds in line 3 of Listing 4 are dynamically determined based on the schedule $\mathcal{T}$, enabling data reuse between loops. The fused code preserves fine-grained parallelism, such as vectorization, inherent to the unfused code. For instance, lines 3–6 in Listing 4 can be mapped to a highly optimized GEMM BLAS on CPUs to leverage vector processors. While using BLAS calls prevents fusion benefits on GPUs, the transformation relies on warp-level parallelism from the original code.

3.2 Dependence Analysis

To ensure the safety of the tile fusion transformation, a dependence analysis is essential to guarantee the correct order of iterations after fusion. The dependence analysis fuses loops if the loop headers are compatible and all three kinds of dependencies, i.e., flow, anti-, and output dependencies, in the original program are preserved. Compatibility of two loops means both loops have the same number of iterations but not necessarily the same loop index expressions. Since the loop fusion lowering discussed in Section 3.1 does not change the order of statements within each loop body and loops are assumed to be parallel, the focus of dependency analysis for correctness is on dependencies between iterations of the two annotated loops with `fprange`. Because, when the two loops are fused, their accesses will be in the same scope, causing dependence violations if the order of iterations is not determined properly. This section discusses how dependencies between the two loops are satisfied.

l_1 and l_2 represent the two loop nests labeled with `fprange` to be fused. We assume there are no basic blocks between l_1 and l_2, or that basic blocks can be safely moved after loop l_2. All statements are internally converted to static single assignment (SSA) form. $l_1 = \{i_1, ..., i_n\}$ and $l_2 = \{j_1, ..., j_n\}$ are loop iterators in the two loops. $\mathcal{D}$ is the set of k data spaces/arrays, $D_1, D_2, ..., D_k$, where the numbering does not indicate a sequence. $R_{l \to D_d}(I)$ and $W_{l \to D_d}(I)$ are access functions where R and W represent, respectively, a read from and a write to data space D_d in one iteration of l. We can derive any kind of dependency across the two loops; for example, the derivation of flow dependency is shown in Equation 1:

$$\begin{aligned} \{J \longrightarrow I | I \in l_1 \wedge J \in l_2 \wedge D_d \in \mathcal{D} \wedge \\ R_{l_2 \to D_d}(J) \cap W_{l_1 \to D_d}(I) \neq \emptyset\} \end{aligned} \tag{1}$$

output dependence can be defined as expected. The output dependency within each loop or between loops are resolved by using `atomic` operations when happens.

Since some data access functions in the flow dependence condition are non-affine at compile time, the tile fusion transformation ensures them by generating code that enforces a partial order at runtime, which is represented by $\mathcal{T}$ in Listing 3. The fusion transformation generates a dependency DAG G to compute the flow dependencies between iterations of the fused loops(I and J), as shown in Equation 1. When $G_{ij} = 1$, it means iteration $j \in J$ should be executed after iteration $i \in I$. To generate the DAG, the flow dependence relations between statements are calculated and then used to generate code that constructs G. Listing 5 shows the general code that is generated for computing dependencies across two loop bodies based on Equation 1. The code can detect dependencies between iterations for any type of affine or non-affine access functions. The generated code iterates over all write accesses of l_1 to check for overlaps with read accesses of l_2 to detect an existing read-after-write dependency (line 7 of Listing 5). Each loop iterator in l_1 and l_2 is defined using its lower bound, upper bound, and indent.

Listing 5: General form of dependency DAG for partial order when loop iterators `i1` and `j1` are fused based on Equation 1.

```
1 for(j1=j1_lb; j1<j1_ub; j1+=j1_ind)
2  ...
3  for(jn=jn_lb; jn<jn_ub; jn+=jn_ind)
4   for(i1=i1_lb; i1<i1_ub; i1+=i1_ind)
5    ...
6     for(in=in_lb; in<in_ub; in+=in_ind)
7      if(R_{l2→Dd}(j1,...,jn) == W_{l1→Dd}(i1,...,in))
8       G[j1].append(i1);
```

For the GeMM-SpMM example shown in Listing 4, the dependence relations for computing the dependence DAG are shown in Equation 2. 2D array indexing is shown linearly.

$$\begin{aligned} \{[j1] \longrightarrow [i1] : 0 \leq i1, j1 < n \\ \wedge Ap[j1] < j2 < Ap[j1+1] \wedge 0 \leq i3, j3 < cCol \\ Ai[j2] * cCol + j3 = i1 * cCol + i3\} \end{aligned} \tag{2}$$

Generating code for the set of relations in Equation 2 requires six nested loop to iterate over $i1$, $j1$, $i2$, $j2$, $i3$, $j3$, which would make the DAG creation $O(n \times bCol \times cCol^2 \times nnz)$. We apply a set of symbolic calculations to simplify the relations. We treat access functions symbolically and as mathematical functions to find where accesses are equal at compile time, when possible. For example, for the GeMM-SpMM DAG code, the constraint $Ai[j2] * cCol + j3 = i1 * cCol + i3$ can be simplified to $Ai[j2] = i1$ because $i3$ and $j3$ have the same iteration space. An efficient `dependenceDAGGeMMSpMM` after symbolic simplification is shown in Listing 6. This function can be further simplified by removing the if condition, since the condition is always true with the assignment in place, making the function perform in $O(nnz)$.

Listing 6: Dependency function in Listing 2 for the input code in Listing 1.

```
Graph DependenceDAGGeMMSpMM(A){
 Graph G(A.n); // Graph is a 2D vector
  for(j1=0; j1<A.n; j1++)
   for(j2=Ap[j1]; j2<Ap[j1+1]; j2++)
     if (Ai[j2] == j1)
       G[j1].append(Ai[j2]);
  return G;
}
```

3.3 Compiler Implementation

We implement the tile fusion transformation as a `Python`-to-C++ code generator. The input Python code is converted to SSA form using the Numba compiler [35]. Numba is a compiler for Python that converts Python code to an intermediate representation (IR), applies a series of optimizations, and finally converts it to LLVM [36] IR and machine code. We use Numba because it enables the use of features such as SSA form and provides existing structures for identifying loops and their data accesses.

From the SSA form, data access functions, loop iterators, and their bounds are extracted. First, intersections of read and write accesses are extracted to find the three types of dependencies. During this process, it is assumed that the bodies of the two loops annotated with fprange are in the same scope. For each pair of flow dependencies, we list the iteration space of indices as data access functions. Data access functions are converted to SymPy [39] expressions so that symbolic simplification can be applied. SymPy is a package that supports working with symbolic expressions. For dependence analysis, a list of data spaces and their access functions are used. After analysis is complete, the fused code and dependence DAG generator are generated as C++ or CUDA library code.

We directly convert the SSA form into C++ by lowering `parallel_for` to OpenMP directives for CPU parallelism: `#pragma omp parallel for`. For CUDA code, the parallel loop is removed and replaced with a thread ID initialized by the grid size. The fused loop variable is replaced throughout the loop bodies with the thread ID variable. While this paper focuses on non-affine dependencies, affine dependencies, such as those with one-to-one dependencies or affine indexing, are also supported and used where applicable.

4 Tile Fusion Scheduler

The tile fusion scheduler, outlined in Algorithm 1, constructs a fused tile schedule at runtime based on the sparsity pattern. This subsection explains inputs, output, objective, and its two-step process.

Inputs and output. The tile fusion scheduler, as outlined in Algorithm 1, takes as input the dependency graph G, the number of columns in matrices B and C denoted by $bCol$ and $cCol$, architecture-specific parameters including the number of physical cores p and cache size per core $cacheSize$, and a heuristic parameter $ctSize$. The iteration space of the two loops, I and J are provided as inputs. The output is a set of fused tiles $\mathcal{T}$. The graph G represents dependencies between the n iterations of the two fused loops, where $G_{i,j} = 1$ indicates that iteration j of the second loop depends on iteration i of the first loop. Each fused tile in $\mathcal{T}$ is represented as $\mathcal{T}_{w,v}$ where w and v signify the *wavefront* number and tile number, respectively. A wavefront is a set of iterations that can execute independently without violating correctness.

Objective and Constraints. The objective of the tile fusion scheduler is to maximize the fused ratio across all tiles $\mathcal{T}$ while tiles fit into fast memory with two wavefronts or less. The fused ratio is calculated as the total iterations of the second computation within the first wavefront divided by the total number of iterations, as shown in Equation 3:

$$fused\_ratio = \frac{\sum_{v=0}^{|\mathcal{T}_0|} |J_{0v}|}{|I| + |J|} \tag{3}$$

where J_{wv} denotes the list of iterations from the second operation in tile $\mathcal{T}_{wv}$ and I and J shows the list of all iterations (or iteration space) of the first and second operations,

Algorithm 1: Tile Fusion Scheduler

```
Input  : $G, bCol, cCol, p, cacheSize, ctSize, I, J$
Output : $\mathcal{T}$
/* Step 1: Coarse Tile Fusion */
if $\lceil |I|/ctSize \rceil \geq p$ then $t \leftarrow ctSize$ else $t \leftarrow \lceil |I|/p \rceil$
$\mathcal{F} \leftarrow (\{\}, \{\})$
for $i \in I$ do
    $v \leftarrow i/t$
    $\mathcal{F}_{0,v} \leftarrow \mathcal{F}_{0,v} \cup range(i, i+t)$
    for $j \leftarrow i$ to $i+t \wedge j \in J$ do
        if $(i < inEdges(G, j) < i+t)$ then $\mathcal{F}_{0,v} \leftarrow \mathcal{F}_{0,v} \cup j$
        else $\mathcal{F}_{1,v} \leftarrow \mathcal{F}_{1,v} \cup j$
        $j \leftarrow j + 1$
    end
    $i \leftarrow i + t$
end
$\mathcal{F}_{1,v} \leftarrow balance(\mathcal{F}_{1,v}, t)$
/* Step 2: Fused Tile Splitting */
for $w \leftarrow 0$ to $2$ do
    for $v \leftarrow 0$ to $|\mathcal{F}_w|$ do
        if $cost(\mathcal{F}_{w,v}, bCol, cCol) > cacheSize$ then
            $\mathcal{T}_w \leftarrow \mathcal{T}_w \cup split(\mathcal{F}_{w,v}, bCol, cCol, cacheSize)$
        else $\mathcal{T}_w \leftarrow \mathcal{T}_w \cup \mathcal{F}_{w,v})$
        $v \leftarrow v + 1$
    end
    $w \leftarrow w + 1$
end
```

respectively. $\mathcal{T}_0$ is the set of tiles in the first wavefront. Operator $|.|$ indicates the cardinality of a set or the size of a list. The tile fusion scheduler maximizes the fused ratio while adhering to constraints on load balance and locality. The *load balance constraint* enforces a maximum of two wavefronts and a minimum of p tiles per wavefront, expressed as $\forall 0 \leq w < 2; \quad |\mathcal{T}_w| \geq p$. A single wavefront is sufficient when the graph exhibits enough connected components, while two wavefronts is the minimum provide flexibility to accommodate varying fused ratios. The load balance constraint ensures minimum number of barriers, which is two, while ensuring enough workloads for all cores. The *locality constraint* ensures a data movement cost for a tile T_{wv} remains below the cache size per core, formalized as $\forall w, v; \quad cost(\mathcal{T}_{w,v}) < cacheSize$ where $cost(\mathcal{T}_{w,v})$ is data movement cost of T_{wv}.

Figure 1b shows an example DAG G. There is an edge from the first iteration of GeMM to the second iteration of SpMM thus $G_{1,2} = 1$. The output schedule in Figure 3b has three tiles. For example, $\mathcal{T}_{0,1} = (5, 6) \in I \cup 6 \in J$.

Step 1. The first step of the tile fusion scheduler constructs an intermediate fused schedule $\mathcal{F}$ comprising uniform coarse fused tiles to maximize the fused ratio while adhering to the load balance constraint. The scheduler identifies fused iterations from consecutive tiles to enhance spatial locality and reduce scheduling overhead. The algorithm ensures that iterations within different tiles of a wavefront are independent, eliminating the need for synchronization.

Lines 1–13 of Algorithm 1 outline the creation of the intermediate fused tiling $\mathcal{F}$. The scheduler initially computes a uniform tile size t based on the provided coarse tile size $ctSize$ in line 1. As shown, the tile size is set to $ctSize$ if the number of tiles, calculated as $\lceil |I|/ctSize \rceil$, exceeds or is equal to p otherwise, t is set to $|I|/p$. This ensures at least p tiles per wavefront, satisfying the load balance constraint. Each fused tile $\mathcal{F}_{0,k}$ is constructed from t consecutive iterations of I as shown in line 5 and a subset of t consecutive iterations of J as specified in lines 6–13. An iteration from J is included in tile $\mathcal{F}_{0,k}$ only if all its incoming edges are already present in the tile, as indicated in line 7. Iterations that fail this criterion are assigned to tile $\mathcal{F}_{1,k}$ in the second wavefront as shown in line 8. The iterations in the second wavefront, $\mathcal{F}_1$, are evenly distributed across t tiles using the *balance* routine in line 13 to maintain load balance in the second wavefront.

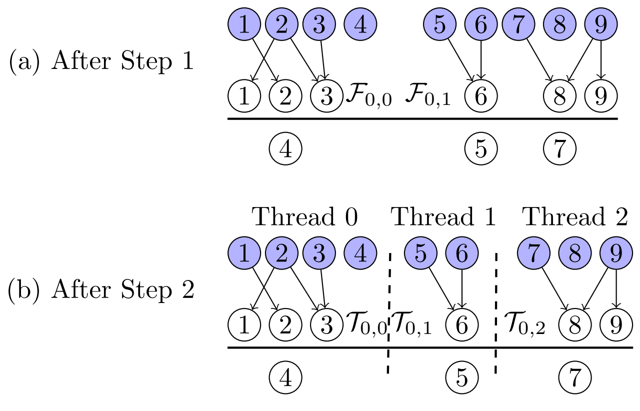


Figure 3: The output of the two steps of the tile fusion scheduler for the sparse matrix in Figure 1 and the fused code in Listing 4. Colored circles represent iterations of loop $i1$ and edges represent dependence.

The coarse tile size parameter, $ctSize$, used to determine t in line 1 of Algorithm 1, is selected empirically to maximize the fused ratio across matrices. To select $ctSize$, we calculate the average fused ratio across all SuiteSparse matrices as a function of tile size. While a larger tile size generally increases the fused ratio, it also reduces parallelism and potentially impacts load balance. Because after a tile size of 2048, the fused ratio increases at a lower rate, we empirically set $ctSize$ to 2048 to achieve a balance between the fused ratio and load balance.

Figure 3a illustrates the output of step 1 for the example presented in Figure 1. Given $ctSize = 4$ and $p = 3$, the tile size t is set to 4. Figure 3a displays two coarse tiles. In tile $\mathcal{F}_{0,0}$, iterations $\{1, 2, 3\} \subset J$ depend solely on iterations $\{1, 2, 3\} \subset I$ which are already present in the tile. Consequently, this tile can be executed independently in parallel.

Step 2. The second step of the tile fusion scheduler refines the coarse tiles generated in the first step, $\mathcal{F}$, to ensure they fit within the fast memory. This process involves splitting tiles as needed to meet the locality constraint. The scheduler iterates over $\mathcal{F}$, evaluating the data movement cost of each tile using a cost model. Tiles exceeding the fast memory capacity are subdivided to accommodate the memory constraint.

The second step of the tile fusion scheduler, outlined in Lines 14–21 of Algorithm 1, refines the coarse tiles. The algorithm iterates over all tiles in both wavefronts, $\mathcal{F}_0$ and $\mathcal{F}_1$, computing the data movement cost for each tile using the *cost* function in line 16. Tiles exceeding the fast memory capacity, *cacheSize*, are recursively split into smaller tiles, each fitting within the fast memory, using the *split* routine in line 16. The resulting tiles are added to the same wavefront in the final schedule $\mathcal{T}$.

Data movement cost. The tile fusion scheduler employs a cost model to estimate data movement for each coarse tile. As indicated in line 16 of Algorithm 1, the data movement cost for tile $\mathcal{T}_{i,j}$ is calculated based on given values of $bCol$ and $cCol$, as defined by Equation 4:

$$cost(\mathcal{T}_{i,j}, bCol, cCol) = \\ (nz(\mathcal{T}_{i,j}) + uc(\mathcal{T}_{i,j}) + t + |J_{i,j}|) * cCol + idx \quad (4)$$

where, $nz(\mathcal{T}_{i,j})$ is the number of unique nonzeros in the tile from A and B. When B is dense, all $n \times bCol$ elements are considered. The term $uc(\mathcal{T}_{i,j})$ denotes the number of nonzeros with unique columns in the tile, $|J_{i,j}|$ is the number of fused iterations from the second operation and idx is the indexing overhead associated with sparse matrices.

Figure 3b presents the final fused schedule $\mathcal{T}$, derived from the coarse fused tile schedule $\mathcal{F}$ in Figure 3a. The schedule $\mathcal{F}$ comprises two coarse fused tiles, each annotated with its corresponding data communication cost. Assuming $bCol = cCol = 1$ and $t = 4$, the cost of $\mathcal{F}_{0,0}$ is calculated using Equation 4. Since this cost is below the *cacheSize* threshold of 30, the tile is directly incorporated into $\mathcal{T}$. Conversely, $\mathcal{F}_{0,1}$ exceeds the *cacheSize* limit and is subsequently divided into two smaller tiles, each satisfying the memory constraint.

Computational Complexity. The first step of the algorithm examines *inEdges* for only t columns of G within each tile of size t. As tiles are disjoint, *inEdges* is accessed once per iteration, resulting in an overall time complexity of $O(nnz)$ for this step. Accessing incoming edges for a given iteration can be performed in linear time. The second step focuses

Platform	IceLake	CascadeLake	EPYC
sockets × cores	2 × 40	2 × 20	2 × 32
L1/L3 Cache Sizes	40K/61M	32K/28M	32K/256M
Compiler	ICC 2024	ICC 2024	GCC v.11
BLAS	MKL'24 [57]	MKL'24	BLIS [55]

Table 1: CPU Platform details

on fused iterations in J and their dependent iterations in I during the splitting process. Since the set of iterations is divided by a factor of two in each split and each split operation can potentially visit up to nnz edges, the time complexity of this step is $O(nnz * log(ctSize))$. The second wavefront processes unfused iterations, requiring $O(|J|)$ time. Consequently, the overall time complexity of the second step is $O(|J| + nnz * log(ctSize))$.

5 Experimental Results

This section evaluates the performance of tile fusion against existing fused and unfused implementations for sparse matrices on three shared-memory processors and a GPU architecture. Overall, tile fusion outperforms both unfused and best-fused code, achieving GMean speedups of 1.98× and 3.52×, respectively. Additionally, tile fusion exhibits scalability across 40, 64, 80 CPU cores and GPU architectures.

5.1 Setup

5.1.1 Environment. All CPU experiments were conducted on the processors listed in Table 1 to assess the cross-platform performance of tile fusion. Unless otherwise specified, experiments were performed on single-socket configurations. For GPU experiments, an A100 GPU is used with CUDA version 12.4. Given the prevalence of single-precision (SP) computations in GNNs [64], all experiments employed SP arithmetic. To evaluate performance across varying matrix sizes, we tested three column dimensions for matrix B: 32, 64, and 128, which align with typical feature dimensions and hidden layer sizes in GNNs [24, 29, 56]. Each reported time represents the median of seven runs. For each matrix, theoretical FLOPs were calculated for the unfused code and used as a baseline for all implementations. The *cacheSize* parameter in Algorithm 1 was set to the sum of L1, L2, and L3 cache sizes per core. Close thread binding was employed, with each thread pinned to a dedicated physical core. We only report the fused code's execution time for matrix-based experiments. The case study includes both scheduler and fused code execution time.

5.1.2 Matrix Dataset. We selected 233 matrices from the SuiteSparse [15] repository to evaluate GeMM-SpMM and SpMM-SpMM computations. To represent scientific computing and machine learning applications, we select (I) all 132 symmetric positive definite (SPD) matrices with more than 10^5 nonzero elements, and (II) all 111 square, non-SPD, graph-related matrices with more than 10^5 nonzero elements and either integer or real data types. A matrix was classified as graph-related if its metadata included the keyword "graph". We reorder matrices with a fused ratio lower than 0.01% using Cuthill-McKee [14] reordering.

5.1.3 Unfused Implementations. To establish an unfused baseline for comparison, we employed the OneAPI MKL [57] library to implement unfused versions of the operations. As MKL does not provide fused GEMM or SpMM routines, we used the `cblas_?gemm` and `mkl_sparse_?_mm` functions separately. The number of threads in MKL is set to the number of physical cores using `mkl_set_num_threads()`. To isolate the impact of tile fusion, we also developed custom unfused parallel implementations for both GeMM-SpMM and SpMM-SpMM, incorporating the same optimizations.

5.1.4 Fused Implementations. Tile fusion, proposed in this work, is implemented in `Python` and `C++`[1] and compared against existing fused implementations and custom-developed implementations of prior fusion techniques.

To compare the fused GeMM-SpMM of tensor compilers, we utilize generated C++ code from the TACO [30] and SparseLNR [18] for the expression `D(i,l) = A(i,j) * B(j,k) * C(k,l)`, where A is sparse and the other matrices are dense. The best performance from both generated codes is reported as *Best of Tensor Compilers*. The generated code by tensor compilers for the fused GeMM-SpMM is shown in Listing 7. To assess the impact of vectorization, we further optimize the generated code in Listing 7 by using MKL GeMV instead of the nested loops in lines 5-11. For SpMM-SpMM, tensor compilers lack support for fused operations and were excluded from the benchmark.

Listing 7: Tensor compiler code for GeMM-SpMM

```
1 #pragma omp parallel for schedule(runtime)
2 for(int i=0; i<n; i++){
3  for(int j1=A.p[i]; j1<A.p[(i+1)]; j1++){
4   int j = A.i[j1];
5   for(int k=0; k<bCol; k++){
6    int iB = j * bCol + k;
7    for (int l = 0; l<cCol; l++) {
8     int iC = k*cCol+l;
9     int iD = i*cCol+l;
10    D[iD] += (A.x[j1]*B[iB]) * C[iC];
11 }}}}
```

Prior runtime schedulers [16, 32] are not publicly available; we implemented their core concepts based on the authors' descriptions. For the communication-avoiding (CA) [16] approach, we evenly partitioned iterations of the first loop and included all dependent iterations in the same partition, resulting in an *overlapped tiling* implementation. For sparse

[1]publicly available from https://github.com/SwiftWare-Lab/tile-fusion

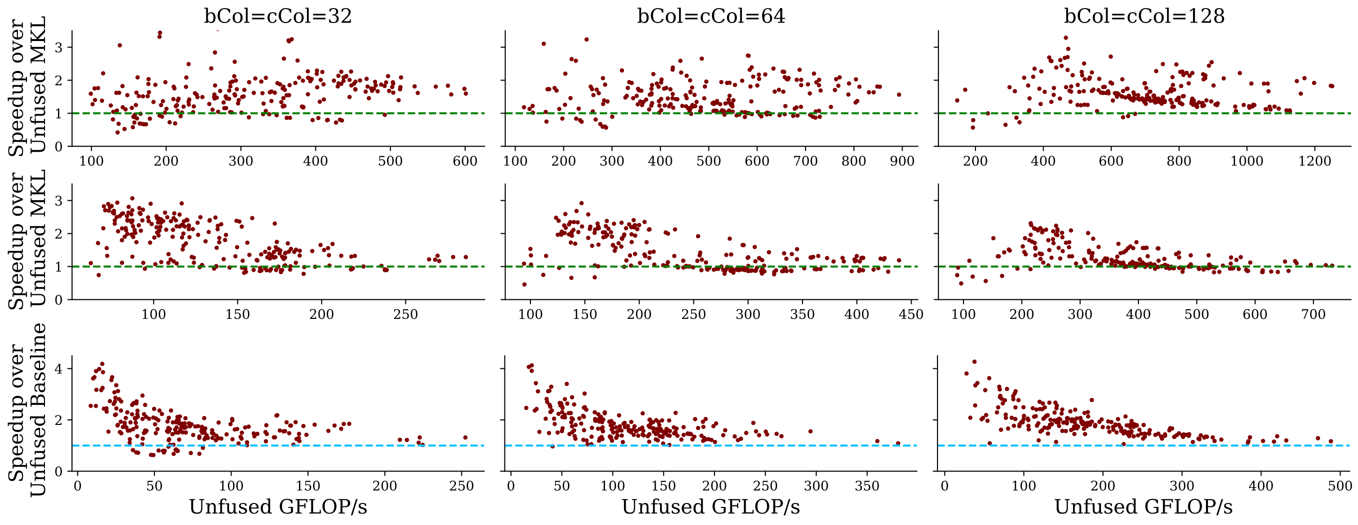


Figure 4: Tile Fusion GeMM-SpMM speedup on IceLake (top), CascadeLake (middle), and EPYC (bottom). Tile fusion outperforms the best performing code in most matrices and bCols.

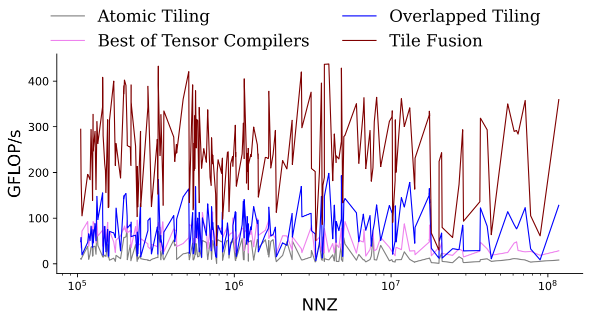


Figure 5: GeMM-SpMM performance of fused methods

Baseline	bCol	Icelake	CascadeLake	EPYC
Unfused MKL	32	1.48	1.63	-
	64	1.42	1.31	-
	128	1.55	1.36	-
Unfused Baseline	32	1.41	1.41	1.64
	64	1.49	1.25	1.67
	128	1.64	1.13	1.73

Table 2: The tile fusion performance for GeMM-SpMM with respect to unfused baselines for three bCol values, 32, 64, and 128 and on three architectures.

tiling [32], we equally divided iterations of the first loop and incorporated dependent iterations from the second loop into each partition, using atomic operations and barriers to manage dependencies, yielding an *atomic tiling* implementation. These two methods are explained in [17] using an example.

5.2 GEMM-SpMM Evaluation

5.2.1 Performance in FLOPs. Figure 4 presents the overall performance of GeMM-SpMM using tile fusion and unfused implementations on the processors listed in Table 1 for three different values of *bCol*. Tile fusion outperforms MKL for 90% of the matrices across all *bCol* values. Table 2 summarizes the speedup achieved by tile fusion for GeMM-SpMM.

Tile fusion performance improves with increasing *bCol* values due to higher arithmetic intensity. The GMean performance rises from 427 GFLOP/s at $bCol = 32$ to 1036 GFLOP/s at $bCol = 128$. In contrast, the MKL implementation exhibits a performance increase from 287 GFLOP/s to 630 GFLOP/s over the same *bCol* range. The enhanced arithmetic intensity within fused tiles provides a greater advantage for tile fusion.

Tile fusion shows a slightly better performance on SPD matrices compared to graph matrices. This difference is attributed to the average fused ratio being twice as high for SPD matrices. Moreover, the larger L3 cache of the EPYC processor amplifies the performance gap between tile fusion and the unfused baseline for larger matrices. Tile fusion also applies to matrix multiplications when transpose of C is provided. Tile fusion provides a GMean of 1.48×, 1.42×, and 1.55× over MKL on IceLake for $bCol = cCol$=32, 64, 128, respectively.

Figure 5 compares the performance of tile fusion with other fused implementations on CascadeLake. Tile fusion surpasses tensor compilers, atomic tiling, and overlapped tiling with GMean speedups of 9.4×, 13.6×, and 3.5×, respectively. Tensor compilers tend to underutilize memory hierarchies due to their heavy reliance on vector operations.

5.2.2 Ablation Study. This section examines the impact of tile fusion on locality and load balance, as well as the contribution of the scheduler's two steps to overall performance.

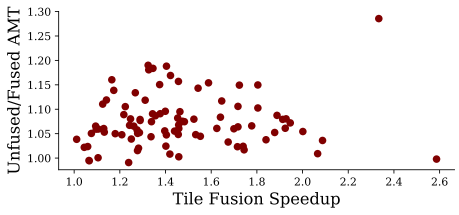


Figure 6: Tile fusion improves the average memory access time (AMT) for GeMM-SpMM.

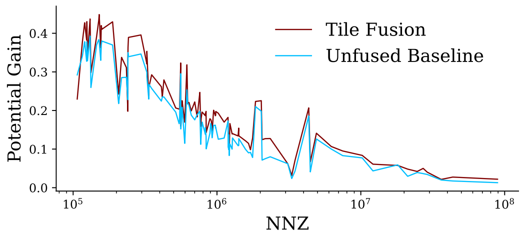


Figure 7: GeMM-SpMM Potential Gain (lower is better). Despite creating coarsened tiles to fit the cache, tile fusion schedules well-balanced workloads that are comparable to fine-grained workloads of the Unfused Baseline.

For this analysis, we selected all 111 graph matrices from the dataset and conducted experiments on CascadeLake.

To assess the impact of tile fusion on locality, we measure average memory access time (AMT) as *AMT = hit time + miss ratio * miss penalty* for all three levels of caches. We use PAPI [54] performance counters, PAPI_L1_TCM, PAPI_L2_TCM, PAPI_L3_TCM to measure L1 accesses, L2 accesses, L3 accesses, and main memory accesses, respectively to compute hit and miss ratios. Figure 6 shows the average memory access time for the graph matrices. Tile fusion demonstrates a 1.1-1.3× reduction in AMT compared to the unfused implementation for 92% of the graph matrices, highlighting the significant role of improved locality in performance gains.

To evaluate the impact of tile fusion on load balance, we measure the potential gain (PG), defined as the maximum potential time savings if all threads were perfectly balanced. We calculate PG as the average difference between the maximum thread execution time and the execution time of other threads, using the PAPI counter PAPI_TOT_CYC to measure thread-level cycles. Figure 7 compares the PG of tile fusion to that of the unfused implementation. While tile fusion demonstrates load balancing comparable to the unfused code, the latter exhibits slightly better load balance due to a larger number of finer-grained tasks.

Figure 8 presents a performance breakdown of the two-step tile fusion process. The first step enhances sequential

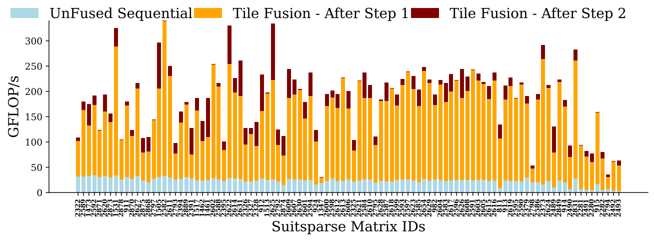


Figure 8: Performance Breakdown of Tile Fusion

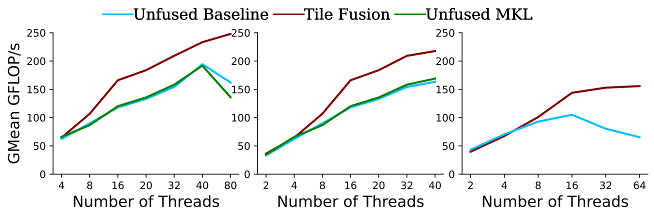


Figure 9: GeMM-SpMM strong scalability on IceLake, CascadeLake, EPYC, from left to right respectively

baseline performance (Figure 1b) by a GMean speedup of 6.7×. The second step contributes to performance improvements for 90% of the matrices analyzed. While the first step introduces threading and improves locality, the second step refines load balance and further optimizes locality using the cost model in Equation 4. For the selected graph matrices, the chosen tile sizes range from 64 to 2048.

5.2.3 Scalability Analysis. This section evaluates the strong scaling of tile fusion across varying thread and sockets. Figure 9 illustrates the geometric mean (GMean) performance across all matrices for different thread counts and architectures. While tile fusion does not incorporate explicit NUMA optimizations, its disjoint coarse-grain partitions effectively reduce memory movement between sockets, enabling scalability on NUMA systems. As the number of threads increases, the performance gap between tile fusion and the unfused implementation widens. This trend is attributed to the growing importance of balancing locality and load distribution in highly parallel environments, where tile fusion excels.

Baseline	bCol	Icelake	CascadeLake	EPYC
Unfused MKL	32	1.43	1.30	-
	64	1.38	1.19	-
	128	1.51	1.30	-
Unfused Baseline	32	1.35	1.19	1.14
	64	1.48	1.23	1.17
	128	1.53	1.26	1.19

Table 3: Tile Fusion GMean speedups for SpMM-SpMM over three bCol values, 32, 64, and 128 on the three CPU architectures.

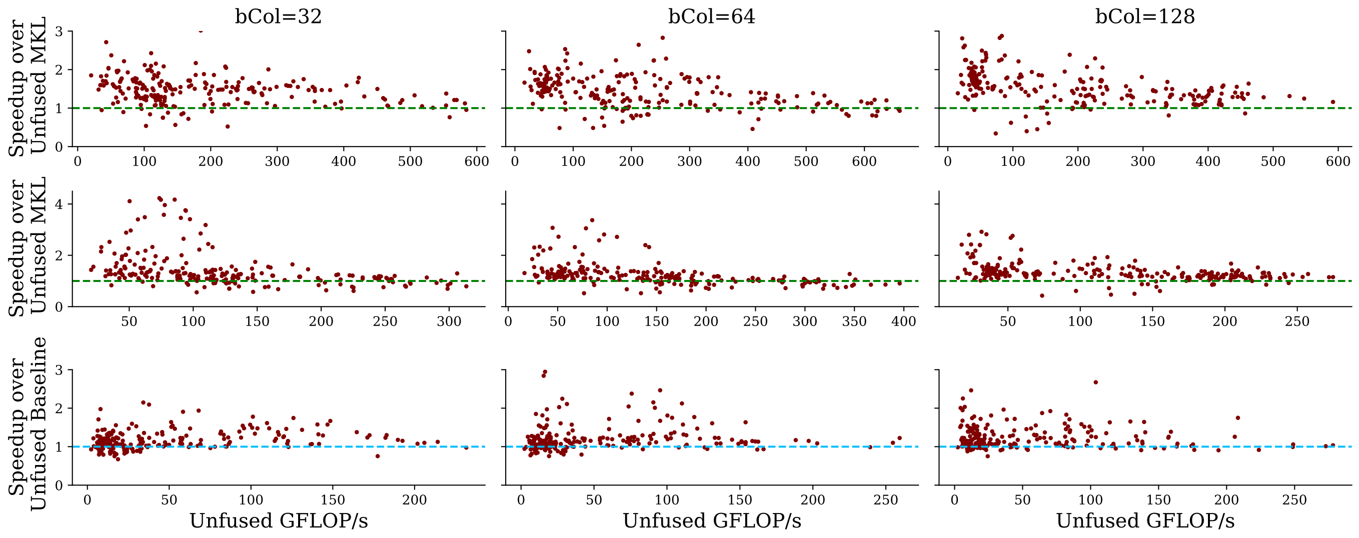


Figure 10: Tile fusion improves the performance of SpMM-SpMM on Ice Lake (top), Cascade Lake (middle), and EPYC (bottom) compared to an unfused baseline.

5.2.4 SpMM-GeMM Performance. Tile fusion is also applied to where SpMM followed by GeMM, i.e. SpMM-GeMM. This variant leads to an affine dependence and thus fusion is done at compile time. But still the tile fusion scheduler enables data reuse where it leads to a speedups of up to 2.17× over Unfused MKL with a GMean speedup of 1.33×. The tile fusion performance ranges from 33 to 333 GLOP/s when $bCol = cCol = 64$.

5.2.5 Scheduler Overhead analysis. To assess the amortized cost of the scheduler, we calculate the number of fused code executions required to offset the scheduler overhead for each matrix in the dataset. Scheduler overhead is amortized after 1–100 fused code executions for the dataset. Given the frequent use of GeMM-SpMM in applications like GNN training, this overhead becomes insignificant.

Id	Name	Vertices	Edges
0	Amazon2k [43]	303,296	586,902
1	Coauthor CS [52]	18,333	163,788
2	Coauthor Physics [52]	34,493	495,924
3	Cora [5]	19,793	63,421
4	DeezerEurope [49]	28,281	185,504
5	Facebook [48]	22,470	342,004
6	Flickr [63]	89,250	899,756
7	Github [48]	37,700	578,006
8	OGBN Arxiv [26]	232,965	114,615,892
9	OGBN products [26]	2,449,029	123,718,152
10	OGBN proteins [26]	132,534	79,122,504
11	PPI [65]	56,944	818,716
12	Reddit [63]	232,965	23,213,838
13	Yelp [63]	716,847	13,954,819

Table 4: Datasets used for evaluating GCN training

5.3 SpMM-SpMM Evaluation

Figure 10 compares the SpMM-SpMM performance of tile fusion with unfused implementations. Tile fusion outperforms both the unfused baseline and MKL implementations in 88% and 91% of tested matrices, respectively, across all *bCol* values. Detailed speedups for each architecture are presented in Table 3. Overall, SpMM-SpMM exhibits lower performance compared to GeMM-SpMM for the same set of matrices due to the inherent memory-bound nature of SpMM.

Tile fusion achieves GMean speedups of 9.3×, 13.2×, and 13.7× over atomic tiling for *bCol* values of 32, 64, and 128, respectively. A similar trend is observed for overlapped tiling, with tile fusion delivering GMean speedups of 5×, 6.5×, and 7.2× for the same *bCol* range. The increased redundant computations in overlapped tiling, as exemplified by matrices G2_circuit and inline_1 with 126,487 and 2,844,351 redundant iterations compared to 150,102 and 503,712 rows, respectively, contribute significantly to the performance gap.

5.4 GPU Evaluation

Figure 11 shows the performance of tile fusion applied to outermost loops of SpMM operations on an A100 GPU. The baseline is the unfused SpMM code with similar warp-level parallelism. Tile fused SpMM-SpMM provide a GMean speedup of 1.18×, 1.16×, and 1.15× for when bCol=cCol is 32, 64, and 128, respectively. Tile fusion speedup range is between 0.2× – 9× and it is faster than the unfused baseline in more than 68% of matrices.

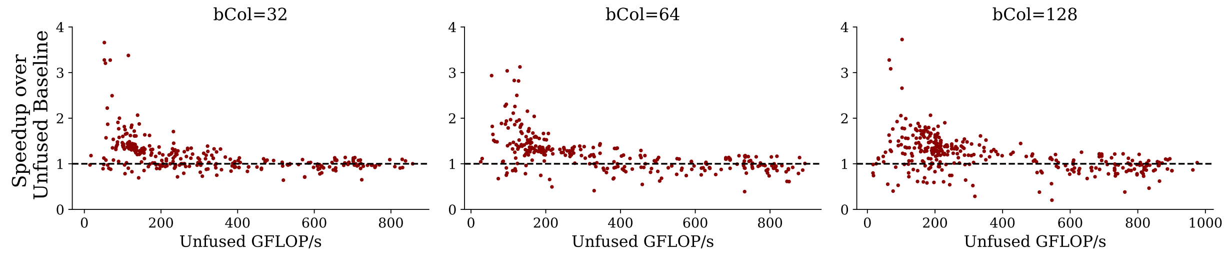


Figure 11: Tile Fusion improves the SpMM-SpMM performance on A100 GPU across bCol values.

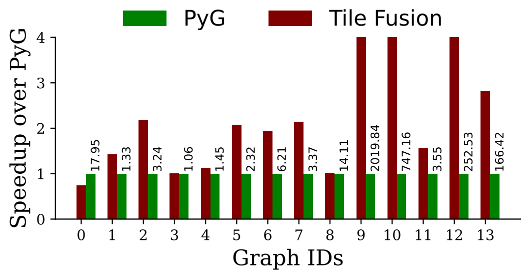


Figure 12: The GCN training performance for graphs in Table 4 is presented in seconds using PyG as a baseline.

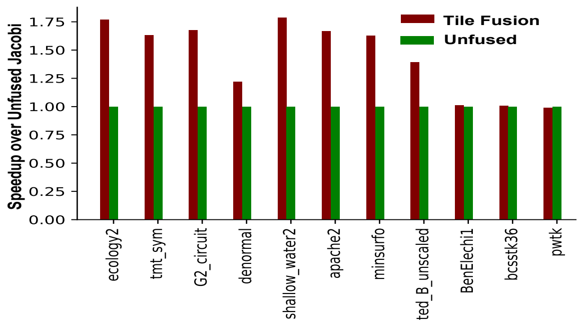


Figure 13: Tile fused SpMM-SpMM improves the Jacobi solver performance over unfused code.

6 Case Studies

6.1 GNN Training

We use training a standard Graph Convolutional Network (GCN) as a case study to evaluate the performance of tile fusion. GCNs, as introduced in Kipf and Welling (2016) [29], are a common type of GNN where each vertex learns from its neighbors within the graph. GCN training involves an initial normalization of the input sparse graph A followed by iterative minimization of a loss function to compute hyperparameters. Both the forward and backward passes within each iteration include GeMM and SpMM operations, which collectively account for 50-90% of the training time. Given that the sparsity pattern of A remains constant during GCN training, the tile fusion schedule can be computed once and reused throughout the training process.

The GCN training process, enhanced with tile fusion, takes as input the graph's adjacency structure A, node feature data F, embedding dimensions $eDim$, the number of processing layers $nLayers$, and optimization settings $Optimizer$, and produces the learned model parameters W and the calculated loss L. For node classification, the correct node labels $Labels$ are used to compute the loss. The process begins with normalization steps, $\widehat{A} = E^{-1/2}(\widehat{A}+I_n)E^{-1/2}$, where I_n represents an identity matrix and E is a diagonal matrix of node degrees. Subsequently, a schedule is created to optimize matrix operations based on the structure of A. The training involves repeated forward and backward computations. The forward pass computes $H_l = \widehat{A}H_{l-1}W_l$, where H_l is the output of the l^{th} layer, and W_l is the corresponding weight matrix. This operation is performed with `TileFusedGeMMSpMM`.

This evaluation examines the performance of GCN training using the optimized matrix multiplication implementation. The number of layers was set to 2, and the maximum number of training iterations was set to 100, aligning with typical GCN [29] model configurations. The optimized implementation was integrated into PyTorch for direct comparison with PyG. Datasets in Table 4, representing a range of medium to large-scale graphs, were used. All experiments were conducted on Ice Lake.

Figure 12 presents the performance of GCN training using tile fusion and PyG. Tile fusion achieves geometric mean (GMean) speedups of 3.84×, 2.33×, and 1.34× for $eDim$ of 32, 64, and 128, respectively. While tile fusion performs well across $bCol$ values (shown in Table 2), PyG performs better with larger $eDim$ values, altering the speedup trend. We attribute this PyG behavior to matrix caching between forward and backward passes. A 2-layer GCN with 100 training iterations incurs 400 GeMM-SpMM calls, with 50% occurring during the forward pass and the remaining 50% during the backward pass. GeMM-SpMM computations account for up to 90% of the total training time for large datasets. The scheduler is invoked only once during training and contributes less than 0.5% to the overall execution time.

6.2 Jacobi Linear Solver

To show the efficiency of the fused SpMM-SpMM in an end-to-end application, we selected the Jacobi method for solving a linear system of equations $AX = B$, where X and B are $n \times bCol$ matrices. The Jacobi method is an iterative solver where in each iteration k, the solution is computed as $X^{(k+1)} = D^{-1}(B - (L+U)X^{(k)})$, where A is decomposed into $L+U+D$, and L, U, and D are the lower, upper, and diagonal matrices, respectively. In each iteration, the convergence with respect to each column of B is calculated, and a convergence flag corresponding to that column is set separately.

As shown, in order to compute $(L+U)X^{(k)}$, an SpMM should be performed. To enable the application of fusing two consecutive SpMM operations, two iterations of Jacobi are unrolled, and then tile fusion is applied. The code is implemented using double precision. Figure 13 shows the speedup when $bCol = 64$ over Jacobi with an unfused implementation. Tiled fused Jacobi provides a GMean speedup of 1.32× over the unfused baseline.

7 Related Work

Loop transformation for sparse codes. Loop tiling and fusion for sparse codes are established techniques for enhancing data locality. Extensive research has focused on applying these techniques to SpMM [1, 25, 34, 42] and GeMM [33, 57]. Tile fusion preserves data locality across iterations of both GeMM and SpMM, leveraging existing optimizations for these operations. Similar to the matrix signature approach [34] for SpMM-CSR, tile fusion employs data communication costs to determine tile sizes. However, as opposed to matrix signature that is applied to parallel tiling, when applying to loop fusion, memory access within loop bodies and shared data accesses should be taken into account. Tile fusion uniquely considers shared elements between the two operations when constructing fused tiles. Tile fusion also considers non-affine dependence due to fusion that does not occur when tiling SpMM individually.

Compile-time fusion is commonly applied in scenarios with affine dependencies [2]. While indirect memory accesses limit fusion opportunities, some optimizations remain feasible. Tensor expression compilers [3, 18, 30, 37, 41, 60] generate code for tensor expressions. Sparse tensor compilers [18, 30, 41] specifically support fusing chain multiplications. However, the fused code generated by SparseLNR [18] and TACO [30] transforms consecutive matrix operations into matrix-vector operations, leading to suboptimal utilization of the fast memory. There is a group of sparsity-specific compilers for matrix multiplications such as DASTAC [20, 21], SPLAT [23], sparse register tiling [59], and partially strided codelets [8] that use sparsity for optimizing code, but they do not use it to enable fusion across multiple tensor operations. Loop-level dependence analysis within a loop body in sparse codes is also supported in prior methods such as the Sparse polyhedral model [40] to enable parallelism. However, inter-loop dependence analysis is additionally needed for loop fusion, which is addressed in tile fusion.

Runtime schedulers. Modeling parallel loops, such as consecutive matrix multiplications, as graphs [16, 32, 53] or hypergraphs [45] is commonly used to enhance cache reuse. These methods, tailored for shared memory processors, often rely on either synchronization [32] or overlapped computation [16] to manage dependencies. The fused ratio and its associated cost model in tile fusion can improve these methods. Sympiler [7, 9] uses DAG schedulers [10, 62] to build an initial schedule of iterations and then fuses the schedule with another loop using sparse fusion [12, 13]. The sparse fusion scheduler is driven by a loop-carried dependence that commonly occur in scientific solvers [11]. Tile fusion, instead, uses a cost model to determine fused tiles since the loops are parallel matrix multiplications.

GNN models such as FusedMM [4, 46] have focused on fusing SDDMM-SpMM in multi-cores. For GCNs, which predominantly use GeMM-SpMM [19, 58], FusedMM offers an efficient SpMM implementation, akin to an unfused implementation. Prior studies have optimized GNNs through sampling techniques [38]. While this paper concentrates on full-graph computations, sampling methods can further enhance fused ratios. Existing GNN optimization approaches primarily focus on optimizing the SpMM operation [27], transforming GeMM-SpMM into vector operations [22], or employing graph partitioning schemes [6, 38, 61], introducing redundant computations [6] or synchronization overhead [38].

8 Summary and Conclusion

This paper presents tile fusion to enable the fusion of parallel loops with sparse dependencies, such as those found in GeMM-SpMM and SpMM-SpMM. Tile fusion employs a compile-time transformation to fuse two matrix multiplication codes. The partial order and efficiency of the fused code are ensured through a runtime scheduler. Tile fusion provides a GMean speedup between 1.13× and 1.73× over unfused code across CPUs and between 1.15× and 1.18× on an A100 GPU. Tile fusion demonstrates an end-to-end speedup of 2.33× and 1.3× in GNNs and Jacobi solvers.

Acknowledgments

This work is supported by NSERC discovery grants (RGPIN-2023-04897, DGECR-2023-00133), NSERC alliance grant (ALLRP 586319-23), Chameleon cloud [28], and the Digital Research Alliance of Canada (www.alliancecan.ca). We would also like to thank the ICS reviewers for their constructive feedback.

References

[1] Khalid Ahmad, Cris Cecka, Michael Garland, and Mary Hall. 2024. Exploring data layout for sparse tensor times dense matrix on GPUs. *ACM Transactions on Architecture and Code Optimization* 21, 1 (2024), 1–20.

[2] Arash Ashari, Shirish Tatikonda, Matthias Boehm, Berthold Reinwald, Keith Campbell, John Keenleyside, and P. Sadayappan. 2015. On optimizing machine learning workloads via kernel fusion. *SIGPLAN Not.* 50, 8 (jan 2015), 173–182. https://doi.org/10.1145/2858788.2688521

[3] Henrik Barthels, Marcin Copik, and Paolo Bientinesi. 2018. The generalized matrix chain algorithm. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. 138–148.

[4] Vivek Bharadwaj, Aydın Buluç, and James Demmel. 2022. Distributed-memory sparse kernels for machine learning. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 47–58.

[5] Aleksandar Bojchevski and Stephan Günnemann. 2018. Deep Gaussian Embedding of Graphs: Unsupervised Inductive Learning via Ranking. arXiv:1707.03815 [stat.ML] https://arxiv.org/abs/1707.03815

[6] Zhenkun Cai, Xiao Yan, Yidi Wu, Kaihao Ma, James Cheng, and Fan Yu. 2021. DGCL: an efficient communication library for distributed GNN training. In *Proceedings of the Sixteenth European Conference on Computer Systems*. 130–144.

[7] Kazem Cheshmi. 2022. *Transforming Sparse Matrix Computations*. Ph. D. Dissertation. University of Toronto (Canada).

[8] Kazem Cheshmi, Zachary Cetinic, and Maryam Mehri Dehnavi. 2022. Vectorizing sparse matrix computations with partially-strided codelets. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[9] 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.

[10] 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.

[11] 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.

[12] Kazem Cheshmi, Michelle Strout, and Maryam Mehri Dehnavi. 2023. Runtime composition of iterations for fusing loop-carried sparse dependence. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[13] Kazem Cheshmi, Michelle Mills Strout, and Maryam Mehri Dehnavi. 2022. Optimizing sparse computations jointly. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 459–460.

[14] Elizabeth Cuthill and James McKee. 1969. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th national conference*. 157–172.

[15] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1–25.

[16] James Demmel, Mark Hoemmen, Marghoob Mohiyuddin, and Katherine Yelick. 2008. Avoiding communication in sparse matrix computations. In *2008 IEEE International Symposium on Parallel and Distributed Processing*. IEEE, 1–12.

[17] Mohammad Mahdi Salehi Dezfuli and Kazem Cheshmi. 2024. Improving Locality in Sparse and Dense Matrix Multiplications. *arXiv preprint arXiv:2407.00243* (2024).

[18] 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.

[19] Matthias Fey and Jan Eric Lenssen. 2019. Fast graph representation learning with PyTorch Geometric. *arXiv preprint arXiv:1903.02428* (2019).

[20] Mahdi Ghorbani, Emilien Bauer, Tobias Grosser, and Amir Shaikhha. 2025. Compressed and Parallelized Structured Tensor Algebra. *Proceedings of the ACM on Programming Languages* 9, OOPSLA1 (2025), 1717–1745.

[21] Mahdi Ghorbani, Mathieu Huot, Shideh Hashemian, and Amir Shaikhha. 2023. Compiling structured tensor algebra. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2 (2023), 204–233.

[22] Zhangxiaowen Gong, Houxiang Ji, Yao Yao, Christopher W Fletcher, Christopher J Hughes, and Josep Torrellas. 2022. Graphite: optimizing graph neural networks on CPUs through cooperative software-hardware techniques. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 916–931.

[23] Ahan Gupta, Yueming Yuan, Devansh Jain, Yuhao Ge, David Aponte, Yanqi Zhou, and Charith Mendis. 2025. SPLAT: A framework for optimised GPU code-generation for SParse reguLar ATtention. *Proceedings of the ACM on Programming Languages* 9, OOPSLA1 (2025), 1632–1660.

[24] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *Advances in neural information processing systems* 30 (2017).

[25] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P Sadayappan. 2019. Adaptive sparse tiling for sparse matrix multiplication. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. 300–314.

[26] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2021. Open Graph Benchmark: Datasets for Machine Learning on Graphs. arXiv:2005.00687 [cs.LG] https://arxiv.org/abs/2005.00687

[27] Guyue Huang, Guohao Dai, Yu Wang, and Huazhong Yang. 2020. Gespmm: General-purpose sparse matrix-matrix multiplication on gpus for graph neural networks. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–12.

[28] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbach, Alex Rocha, and Joe Stubbs. 2020. Lessons Learned from the Chameleon Testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC ’20)*. USENIX Association.

[29] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[30] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–29.

[31] Boris Krasnopolsky and Alexey Medvedev. 2021. XAMG: A library for solving linear systems with multiple right-hand side vectors. *SoftwareX* 14 (June 2021), 100695. https://doi.org/10.1016/j.softx.2021.100695

[32] 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.

[33] HT Kung, Vikas Natesh, and Andrew Sabot. 2021. Cake: matrix multiplication using constant-bandwidth blocks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[34] Süreyya Emre Kurt, Aravind Sukumaran-Rajam, Fabrice Rastello, and Ponnuswamy Sadayyapan. 2020. Efficient tiled sparse matrix multiplication through matrix signatures. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–14.

[35] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. 2015. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. 1–6.

[36] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004*. IEEE, 75–86.

[37] Peiming Liu, Alexander J Root, Anlun Xu, Yinying Li, Fredrik Kjolstad, and Aart JC Bik. 2024. Compiler Support for Sparse Tensor Convolutions. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2 (2024), 275–303.

[38] Vasimuddin Md, Sanchit Misra, Guixiang Ma, Ramanarayan Mohanty, Evangelos Georganas, Alexander Heinecke, Dhiraj Kalamkar, Nesreen K Ahmed, and Sasikanth Avancha. 2021. Distgnn: Scalable distributed training for large-scale graph neural networks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[39] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, Amit Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. 2017. SymPy: symbolic computing in Python. *PeerJ Computer Science* 3 (Jan. 2017), e103. https://doi.org/10.7717/peerj-cs.103

[40] 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.

[41] Erdal Mutlu, Ruiqin Tian, Bin Ren, Sriram Krishnamoorthy, Roberto Gioiosa, Jacques Pienaar, and Gokcen" Kestor. 2022. COMET: A Domain-Specific Compilation of High-Performance Computational Chemistry. In *Languages and Compilers for Parallel Computing*, Barbara Chapman and José Moreira (Eds.). Springer International Publishing, Cham, 87–103.

[42] Vikas Natesh, Andrew Sabot, HT Kung, and Mark Ting. 2023. Rosko: Row Skipping Outer Products for Sparse Matrix Multiplication Kernels. *arXiv preprint arXiv:2307.03930* (2023).

[43] Jianmo Ni, Jiacheng Li, and Julian McAuley. 2019. Justifying Recommendations using Distantly-Labeled Reviews and Fine-Grained Aspects. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan (Eds.). Association for Computational Linguistics, Hong Kong, China, 188–197. https://doi.org/10.18653/v1/D19-1018

[44] Dianne P O'Leary. 1980. The block conjugate gradient algorithm and related methods. *Linear algebra and its applications* 29 (1980), 293–322.

[45] Filip Pawłowski, Rob H Bisseling, Bora Uçar, and AN Yzelman. 2020. Combinatorial Tiling for Sparse Neural Networks. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.

[46] Md Khaledur Rahman, Majedul Haque Sujon, and Ariful Azad. 2021. Fusedmm: A unified sddmm-spmm kernel for graph embedding and graph neural networks. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 256–266.

[47] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Virtual Event, CA, USA) *(KDD '20)*. Association for Computing Machinery, New York, NY, USA, 3505–3506. https://doi.org/10.1145/3394486.3406703

[48] Benedek Rozemberczki, Carl Allen, and Rik Sarkar. 2021. Multi-scale attributed node embedding. *Journal of Complex Networks* 9, 2 (2021), cnab014.

[49] Benedek Rozemberczki and Rik Sarkar. 2020. Characteristic functions on graphs: Birds of a feather, from statistical descriptors to parametric models. In *Proceedings of the 29th ACM international conference on information & knowledge management*. 1325–1334.

[50] Y. Saad. 2003. *Iterative Methods for Sparse Linear Systems* (2nd ed.). Society for Industrial and Applied Mathematics, USA.

[51] Michael Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. 2017. Modeling Relational Data with Graph Convolutional Networks. arXiv:1703.06103 [stat.ML] https://arxiv.org/abs/1703.06103

[52] Oleksandr Shchur, Maximilian Mumme, Aleksandar Bojchevski, and Stephan Günnemann. 2018. Pitfalls of Graph Neural Network Evaluation. *ArXiv* abs/1811.05868 (2018). https://api.semanticscholar.org/CorpusID:53303554

[53] 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.

[54] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. 2010. Collecting performance data with PAPI-C. In *Tools for High Performance Computing 2009: Proceedings of the 3rd International Workshop on Parallel Tools for High Performance Computing, September 2009, ZIH, Dresden*. Springer, 157–173.

[55] Field G Van Zee and Robert A Van De Geijn. 2015. BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Transactions on Mathematical Software (TOMS)* 41, 3 (2015), 1–33.

[56] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. arXiv:1710.10903 [stat.ML] https://arxiv.org/abs/1710.10903

[57] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, Yajuan Wang, Endong Wang, Qing Zhang, Bo Shen, et al. 2014. Intel math kernel library. *High-Performance Computing on the Intel® Xeon Phi™: How to Fully Exploit MIC Architectures* (2014), 167–188.

[58] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, et al. 2019. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315* (2019).

[59] Lucas Wilkinson, Kazem Cheshmi, and Maryam Mehri Dehnavi. 2023. Register Tiling for Unstructured Sparsity in Neural Network Inference. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1995–2020.

[60] Jaeyeon Won, Charith Mendis, Joel S Emer, and Saman Amarasinghe. 2023. WACO: learning workload-aware co-optimization of the format and schedule of a sparse tensor program. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 920–934.

[61] Wenchao Wu, Xuanhua Shi, Ligang He, and Hai Jin. 2023. TurboMGNN: Improving Concurrent GNN Training Tasks on GPU With Fine-Grained Kernel Fusion. *IEEE Transactions on Parallel and Distributed Systems* (2023).

[62] 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.

[63] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. 2019. Graphsaint: Graph sampling based inductive learning method. *arXiv preprint arXiv:1907.04931* (2019).

[64] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. 2020. Graph neural networks: A review of methods and applications. *AI open* 1 (2020), 57–81.

[65] Marinka Zitnik and Jure Leskovec. 2017. Predicting multicellular function through multi-layer tissue networks. *Bioinformatics* 33, 14 (July 2017), i190–i198. https://doi.org/10.1093/bioinformatics/btx252