A Memory-Aware Sparse Matrix-Matrix Multiplication on Multicore Architectures

DESHUN BI, National University of Defense Technology, Changsha, China
SHENGGUO LI, National University of Defense Technology, Changsha, China
HAOZHONG QIU, National University of Defense Technology, Changsha, China
CHUANFU XU, National University of Defense Technology, Changsha, China
XIAOJIAN YANG, National University of Defense Technology, Changsha, China
FAN YUAN, Xiangtan University, Xiangtan, China
DEZUN DONG, National University of Defense Technology, Changsha, China
TIAOJIE XIAO, Laboratory of Digitizing Software for Frontier Equipment, National University of Defense Technology, Changsha, China
JIE LIU, Laboratory of Digitizing Software for Frontier Equipment, National University of Defense Technology, Changsha, China

Sparse matrix–matrix multiplication (SpMM) is a fundamental operation in scientific computing with broad applications across numerous domains. Tiling is a key optimization technique for improving data locality and is widely adopted in high-performance computing. However, the irregular data access patterns inherent to SpMM make it challenging to exploit tiling effectively for data reuse. In this article, we propose **MaSpMM**, a *memory-aware SpMM* framework that integrates cache-aware tiling with a segment-oriented data layout. MaSpMM stores matrices as continuous segments to enhance data locality within each tile. Moreover, since many sparse matrices in real-world applications exhibit symmetry, we further develop MaSpMM-Sym, an extension that recursively partitions symmetric matrices to eliminate write conflicts and further improve locality. To adapt to diverse scenarios, we finally introduce MaSpMM-Adap, which adaptively selects the most suitable approach for each input matrix. Comprehensive evaluations on both x86 and ARM CPUs demonstrate that MaSpMM-Adap achieves average speedups of up to 1.86× over Intel oneMKL, 1.84× over ASpT, and 1.75× over J-Stream.

Deshun Bi and Shengguo Li contributed equally to this research.

This work is supported by the National Key R&D Program of China (2023YFB3001702), National Natural Science Foundation of China (No. 62032023, 42104078, 62502048, and 61902411).

Authors' Contact Information: Deshun Bi, National University of Defense Technology, Changsha, Hunan, China; e-mail: bds123@nudt.edu.cn; Shengguo Li, National University of Defense Technology, Changsha, China; e-mail: nudtlsg@nudt.edu.cn; Haozhong Qiu, National University of Defense Technology, Changsha, China; e-mail: qiuhaozhong16@nudt.edu.cn; Chuanfu Xu, National University of Defense Technology, Changsha, Hunan, China; e-mail: xuchuanfu@nudt.edu.cn; Xiaojian Yang, National University of Defense Technology, Changsha, Hunan, China; e-mail: yangxj@nudt.edu.cn; Fan Yuan, Xiangtan University, Xiangtan, Hunan, China; e-mail: fyuan@smail.xtu.edu.cn; Dezun Dong (corresponding author), National University of Defense Technology, Changsha, Hunan, China; e-mail: dong@nudt.edu.cn; Tiaojie Xiao (corresponding author), Laboratory of Digitizing Software for Frontier Equipment, National University of Defense Technology, Changsha, China; e-mail: xiaotiaojie@nudt.edu.cn; Jie Liu, Laboratory of Digitizing Software for Frontier Equipment, National University of Defense Technology, Changsha, China; e-mail: liujie@nudt.edu.cn.



This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

© 2026 Copyright held by the owner/author(s).

ACM 1544-3973/2026/06-ART55
https://doi.org/10.1145/3803422

CCS Concepts: • **Mathematics of computing → Solvers**; **Mathematical software performance**; • **Computing methodologies → Shared memory algorithms**; **Vector / streaming algorithms;**

Additional Key Words and Phrases: SpMM, Tiling, Vectorization, Symmetry, Recursive Partitioning, Multicore

ACM Reference Format:
Deshun Bi, Shengguo Li, Haozhong Qiu, Chuanfu Xu, Xiaojian Yang, Fan Yuan, Dezun Dong, Tiaojie Xiao, and Jie Liu. 2026. A Memory-Aware Sparse Matrix-Matrix Multiplication on Multicore Architectures. *ACM Trans. Arch. Code Optim.* 23, 2, Article 55 (June 2026), 21 pages. https://doi.org/10.1145/3803422

1 Introduction

Sparse matrix-matrix multiplication (**SpMM**) is widely used in **high-performance computing** (**HPC**) and **graph neural networks** (**GNNs**) [13, 14]. Tiling is a key technique for exploiting data reuse across cache levels and is widely used in dense linear algebra, convolutional neural networks, and stencil computations [9]. While tiling techniques for regular computations are well understood, optimizing data locality for irregular sparse workloads remains a significant challenge.

An SpMM operation, $C = A \cdot B$, multiplies a sparse matrix **A** of size $M \times N$ by a dense matrix **B** of size $N \times K$, producing a dense matrix **C** of size $M \times K$. Previous studies have shown that data movement in sparse computations depends not only on the number of nonzero elements but also on their spatial distribution [9]. Consequently, the sparsity pattern of a matrix can have a significant impact on overall performance [14]. The key limitation of conventional CSR-based [23] SpMM lies in its poor data locality: each sparse row repeatedly reloads dense matrix from memory, causing high bandwidth pressure. TileSpMM [14] addresses this issue by partitioning both sparse and dense matrices into cache-sized tiles, allowing data within a tile to be reused before being evicted. ASpT [9] and J-Stream [14] are two state-of-the-art TileSpMM implementations on multicore CPUs.

In large-scale applications, the dense input matrix often has a large number of columns K, with dimensions reaching into the hundreds or even thousands [16, 28, 33]. However, existing high-performance SpMM libraries often struggle to maintain good scalability as K increases. We evaluate the SpMM performance of the vendor-optimized Intel oneMKL library on *audikw_1* (Figure 1(a)). As K increases from 32 to 2048, oneMKL exhibits pronounced performance degradation: throughput drops from 205.8 GFLOPS to 69.7 GFLOPS (a 2.95× slowdown), and the L2 hit rate decreases from 98.62% to 94.11%. These results suggest that existing SpMM implementations have difficulty sustaining performance as the dense dimension grows, motivating cache-friendly designs for high-dimensional inputs.

To address these limitations, we propose **MaSpMM**,[1] a memory-aware SpMM that employs cache-aware tiling for both the sparse and dense matrices. On the sparse side, **A** is partitioned into row tiles, with each column stored contiguously as a *column-segment* in **compressed sparse column** (**CSC**) format [23]. This layout enables nonzeros sharing the same column to reuse rows of **B** via SIMD-friendly loads. On the dense side, **B** and **C** are tiled along both row and column dimensions, with each tile stored contiguously in row-major order. SpMM is performed in an outer-product formulation. This organization improves spatial locality and allows each tile of **C** to remain cache-resident during accumulation. The storage layout is illustrated in Figure 2(e). Unlike ASpT and J-Stream, however, MaSpMM explicitly tiles the dense matrices along the column dimension, enabling stable and scalable performance as the dense dimension K increases (see Figure 1(a)).

[1]MaSpMM is available at https://github.com/THSparse/MaSpMM.

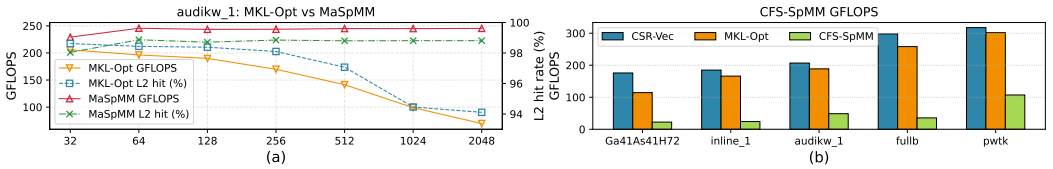


Fig. 1. (a) Performance comparison of MKL-Opt and MaSpMM on matrix *audikw_1* as the dense-column dimension K increases from 32 to 2,048 on x86. The left y-axis reports GFLOPS, and the right y-axis reports the measured L2 cache hit rate using LIKWID [27]. MKL-Opt denotes an optimized MKL; (b) The Performance of CFS-SpMM at $K = 128$ on x86. CFS [5] originally provides only symmetric SpMV, and we extend it using AVX-512 to calculate symmetric SpMM, denoted CFS-SpMM. CSR-Vec is a CSR vectorized implementation.

Many sparse matrices used in practical applications exhibit structural symmetry, such as **Finite Element Analysis** (**FEA**), **Computational Fluid Dynamics** (**CFD**), and graph analytics [1, 2, 5, 7]. Exploiting matrix symmetry can further improve SpMM's memory access efficiency, but symmetric algorithms introduce write conflicts when multiple threads update the result concurrently [1, 2, 5, 7]. While symmetric SpMV has been extensively studied (CFS [5], RACE [1], etc.), symmetric SpMM still lacks effective and scalable solutions. A natural idea is to extend existing symmetric SpMV methods to symmetric SpMM, especially the recently proposed coloring-based conflict avoidance techniques [1, 5]. However, our experiments in Figure 1(b) show that such extensions (e.g., CFS-SpMM) fall short in performance when compared to general-purpose approaches such as MKL-Opt and CSR-Vec. These observations indicate that directly applying or extending existing symmetric SpMV algorithms to SpMM is insufficient for achieving high performance.

To address this issue, we further develop MaSpMM-Sym, an enhanced version of MaSpMM tailored for symmetric matrices. MaSpMM-Sym employs recursive partitioning to eliminate write conflicts and improve data locality. We first construct a conflict graph [5] from the symmetric matrix, where vertices represent rows and edges indicate potential conflicts. Instead of applying traditional graph coloring, we adopt a **divide-and-conquer** (**DC**) paradigm [10] to expose parallelism. A *memory-aware recursive partitioning* is performed on the conflict graph so that each resulting subgraph fits within the fast cache. Subgraphs at each recursion level are separated by a *separator*, which allows conflict-free parallel execution of the symmetric kernel. For the last-level separator, which is typically dense and conflict-prone, we terminate recursion and apply the general MaSpMM kernel instead.

We observe that MaSpMM and MaSpMM-Sym exhibit distinct performance characteristics across matrices and dense widths K. Building on above designs, we further develop MaSpMM-Adap, an adaptive method that selects the appropriate approach for each matrix and K. We evaluate MaSpMM-Adap on a 28-core Intel Xeon Gold 6348 (x86) and a 64-core Kunpeng 920 [29] ARMv8 CPU using sparse matrices from the SuiteSparse collection [4]. We benchmark our method against CSR-Vec, the vendor-optimized Intel oneMKL, and two state-of-the-art SpMM libraries: ASpT and J-Stream. On the x86 platform, MaSpMM-Adap delivers average speedups of up to 1.67× over CSR-Vec, 1.86× over oneMKL, 1.84× over ASpT, and 1.75× over J-Stream. On the ARM platform, it achieves average speedups of up to 1.59× over CSR-Vec, 1.58× over ASpT, and 1.53× over J-Stream.

The main contributions of this article are as follows:

— We propose MaSpMM, a memory-aware SpMM algorithm that combines a tiling strategy with a *segment-oriented* layout to enhance the cache locality of the dense matrix.
— We develop MaSpMM-Sym, which further improves MaSpMM for symmetric matrices by applying *memory-aware partitioning* to enable conflict-free parallel execution while enhancing sparse-side reuse.

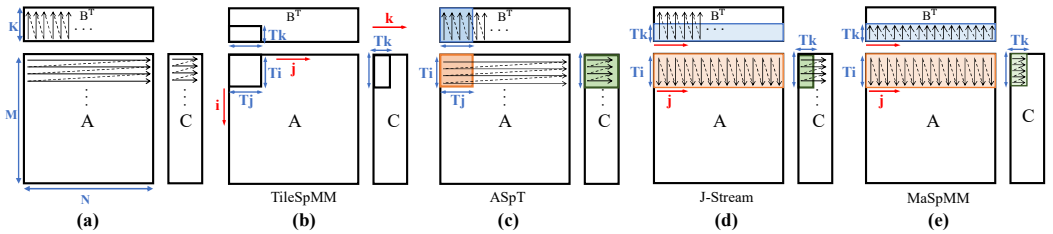


Fig. 2. (a) The CSR SpMM; (b) The TileSpMM; (c) The ASpT workflow; (d) The J-Stream workflow. The *column-segment* format is used in this work, in which each column-segment within a row block of **A** is stored contiguously; (e) The MaSpMM workflow. The *row-segment* format is further adopted in this work, in which the dense matrix is partitioned into column blocks, and each row-segment within a column block is stored contiguously.

ALGORITHM 1: CSR SpMM Algorithm

Input: sparse matrix $A[M][N]$, dense matrix $B[N][K]$.
Output: dense matrix $C[M][K]$.

```
for i = 0 to M − 1 do
    for j = A.rowptr[i] to A.rowptr[i + 1] − 1 do
        for k = 0 to K − 1 do
            C[i][k] += A.value[j] ∗ B[A.colidx[j]][k]
        end
    end
end
```

— We design an adaptive kernel select strategy that chooses the appropriate kernel for different inputs, and experiments show that our approach demonstrates significant performance improvements over state-of-the-art techniques.

2 Background

This section presents the fundamental SpMM algorithm and its symmetric variants, followed by a review of existing parallel approaches for symmetric matrices.

2.1 SpMM

In SpMM, an $M \times N$ sparse input matrix **A** is multiplied by an $N \times K$ dense input matrix **B** to produce an $M \times K$ dense output matrix **C**. We review existing methods that inspire our design, including CSR [24], ASpT [9], and J-Stream [14], which are widely used to enhance vectorization and data locality in SpMM.

CSR. Algorithm 1 presents the sequential SpMM implementations based on the CSR format [24], and Figure 2(a) illustrates the CSR SpMM diagram. In the implementation of Algorithm 1, the outer i-loop iterates over the rows of the sparse matrix **A**. The j-loop processes each nonzero element in the current row of **A**, multiplying it with all elements in the corresponding row of **B** (handled by the k-loop) and accumulating the results into $\mathbf{C}[i][k]$. In order to further improve computation efficiency, the k-loop can be vectorized by utilizing SIMD instructions. Specifically, for each nonzero element in **A**, the corresponding row of **B** is loaded as a vector, and the scalar value from **A** is broadcast into a vector. Then, a vectorized multiplication followed by accumulation into **C** is performed.

TileSpMM. Algorithm 2 shows high-level pseudocode for a TileSpMM algorithm, and Figure 2(b) illustrates the TileSpMM diagram. The outer three layers (**tile-loops**) determine the traversal order

ALGORITHM 2: TileSpMM Algorithm

```
   Input: sparse matrix A[M][N], dense matrix B[N][K].
   Output: dense matrix C[M][K].
1  for k = 0 to ⌈K/T_k⌉ − 1 do
2      kbound = min((k + 1) · T_k, K − 1);
3      for j = 0 to ⌈N/T_j⌉ − 1 do
4          for i = 0 to ⌈M/T_i⌉ − 1 do
5              ibound = min((i + 1) · T_i, M − 1);
6              for ii = i · T_i to ibound do
7                  nnz_begin = A.T_j_tile[j].rowptr[ii];
8                  nnz_end = A.T_j_tile[j].rowptr[ii + 1] − 1;
9                  for jj = nnz_begin to nnz_end do
10                     for kk = k · T_k to kbound do
11                         col = A.colidx[jj];
12                         C[ii][kk]+ = A.value[jj] × B[col][kk];
13                     end
14                 end
15             end
16         end
17     end
18 end
```

of the tiles (Lines 1 to 4). The inner three layers (**intra-tile loops**) determine the calculation order of the elements within the tiles (Lines 6 to 10). For each non-zero element $\mathbf{A(i, j)}$ of **A**, the jth row of the input matrix **B** will be accessed, and the ith row of the output matrix **C** will be written. If nonzero elements within the tile footprint of **A** share the same column index j, they can reuse the corresponding elements of the input matrix **B**. Considering only the **tile-loops**, there are six possible loop permutations [14]. Algorithm 2 illustrates one such permutation, with the i-loop as the innermost loop (Line 4). Our MaSpMM method is also built on TileSpMM and is described in detail in Section 3.1.

ASpT and J-Stream are representative tile-based SpMM implementations. ASpT stores the sparse matrix **A** in row-major order, but computes one 2D tile at a time, restricting accesses to **B** to the tile width and thus enabling cache reuse of **B** [9]. As shown in Figure 2(c), ASpT does not tile the dense operand along the K dimension. As K increases, the dense working set exceeds cache capacity, reducing reuse and lowering performance [9]. In contrast, J-Stream stores the row blocks of the sparse matrix in column major [14]. The access range of each column-segment to the matrix **C** is confined to the segment length, enabling reuse of the **C** tile. As shown in Figure 2(d), although J-Stream tiles along K, its row-major dense layout leads to non-contiguous accesses within each tile. Consequently, its performance degrades as K increases [14]. In addition, J-Stream employs a sparse-matrix signature model [14] to guide tile-size selection by estimating data movement, offering a model-driven alternative to ASpT's empirical tuning approach.

2.2 Symmetric SpMM

A widely used storage format for symmetric sparse matrices is the **Sparse Symmetric Skyline** (**SSS**) format [5, 24], which stores the strictly lower (or upper) triangular part of the matrix using the CSR format, while the main diagonal is kept in a separate array. Algorithm 3 presents a sequential symmetric SpMM implementation leveraging the SSS format. Directly parallelizing the outer loop introduces write contention, as concurrent updates to the output matrix **C** occur (e.g., $C[i][k]$ in

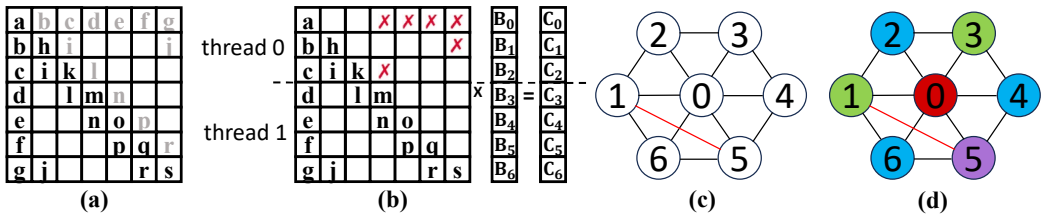


Fig. 3. (a) The symmetric sparse matrix **A**; (b) Write contention (red ×) when two threads are running symmetric SpMM in parallel, and each small square in matrices **B** and **C** represents a row vector of length K; (c) The conflict graph of **A**; (d) A valid coloring (four colors) for the conflict graph of **A**.

ALGORITHM 3: Symmetric SpMM Algorithm

Input: sparse matrix $A[M][N]$, dense matrix $B[N][K]$.
Output: dense matrix $C[M][K]$.

```
1  for i = 0 to M − 1 do
2      // Calculating diagonal elements.
3      for k = 0 to K − 1 do
4          C[i][k] += A.diag[i] ∗ B[i][k]
5      end
6      // Calculate the elements of the lower triangle.
7      for j = A.rowptr[i] to A.rowptr[i + 1] − 1 do
8          col = A.colidx[j]
9          val = A.value[j]
10         for k = 0 to K − 1 do
11             C[i][k] += val ∗ B[col][k]    // Calculation of current position.
12             C[col][k] += val ∗ B[i][k]    // Calculation of symmetric position.
13         end
14     end
15 end
```

Line 11 and $C[col][k]$ in Line 12). As shown in the Figure 3(b), in addition to modifying $\mathbf{C_3}$ through $\mathbf{C_6}$ (current position), thread 1 also updates $\mathbf{C_0}$ through $\mathbf{C_2}$ (symmetric position), resulting in write conflicts between the two threads. Leveraging symmetry can reduce memory footprint and data movement. However, shared-memory parallelization is hindered by write contention. This section provides an overview of existing parallelization approaches.

Atomic and Lock. A straightforward approach involves atomic or locking operations for conflict part [3]. However, this method incurs significant overhead due to the large number of fine-grained synchronization operations, which scale with the number of nonzero elements in the matrix.

Reduction. Another approach, based on reduction, employs thread-local vectors to store intermediate results [2, 7]. Each thread maintains a private vector, which is later merged into the output vector through an additional reduction step. However, this approach increases memory overhead and introduces a reduction step whose cost scales linearly with the number of threads, thereby limiting scalability [5].

Coloring. Prior work [5] introduced the conflict graph to model write conflicts in symmetric calculation. This conflict graph consists of vertices representing matrix rows and edges denoting conflicts between rows. Figure 3(c) shows the conflict graph induced by **A** of Figure 3(a). For

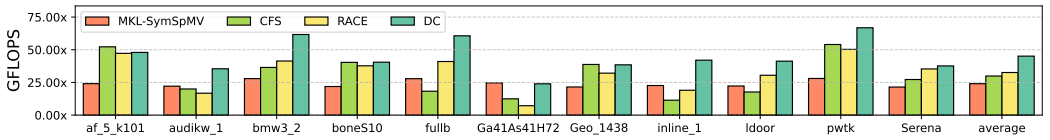


Fig. 4. Symmetric SpMV performance evaluation of the state-of-the-art coloring approaches on x86. MKL-SymSpMV represents the symmetric SpMV interface provided by MKL. CFS denotes the Conflict-Free coloring approach from [5]. RACE denotes the Recursive Algebraic Coloring Engine approach from [1]. We also demonstrated our recursive method applied to SpMV, denoted as DC (For more details, see [22]).

instance, computing the symmetric element of **b** in row 1 induces a *direct conflict* with row 0 in Figure 3(a). Similarly, element **b** in row 1 and element **f** in row 5 both update row 0 at their symmetric positions, which induces an *indirect conflict* between rows 1 and 5. A direct conflict arises between rows i and j when a nonzero of row i has a column index equal to j (or vice versa), and the indirect conflict occurs between rows i and j when they have nonzeros with the same column index [5]. The edges in Figure 3(c) represent all potential write conflicts in the symmetric algorithm on Figure 3(a). If both direct and indirect conflicts exist between a pair of rows, we record only one edge in the conflict graph, since either type of conflict already prevents concurrent execution.

In the coloring method, the matrix-induced conflict graph is first colored so that rows sharing the same color can be executed in parallel, while rows with different colors must be processed sequentially [1, 5]. Figure 3(d) depicts an efficient coloring scheme applied to the conflict graph of matrix **A**. While coloring enables safe parallelism, it inevitably introduces substantial global synchronization among different colors. At least four colors are required in Figure 3(d), indicating four synchronization phases, and the overhead grows proportionally with the number of edges in the conflict graph. Moreover, graph coloring is an NP-hard problem, typically using greedy or heuristic algorithms [12], which often result in poor load balancing across threads. These limitations become more pronounced for large, irregular sparse matrices [1, 22].

Instead of coloring the conflict graph, we adopt a recursive strategy to expose parallelism [22]. The conflict graph is recursively partitioned into subgraphs that fit within the cache, enabling symmetry-based computations, while the remaining regions are fully computed without exploiting symmetry. To highlight the advantages of this recursive approach, we empirically evaluate two recent coloring-based symmetric SpMV methods [1, 5] on a 28-core Intel Xeon Gold 6348 CPU, with results shown in Figure 4. As observed, the coloring-based approaches achieve notable performance gains on *af_5_k101*, *boneS10*, and *pwtk*, compared with the MKL-SymSpMV. However, their performance drops sharply on *inline_1*, *Ga41As41H72*, and *audikw_1*. This is because the corresponding conflict graphs of these matrices are extremely dense (see Table 1), resulting in a large number of race conditions in symmetric SpMV. Our recursive strategy can handle various situations and has the best overall performance. Consequently, we adopt a recursive strategy to implement symmetric SpMM in this work.

3 Methodology

This section presents our proposed memory-aware SpMM design. We first describe MaSpMM, which uses a segment-oriented layout to enhance dense-side tiling locality. Next, we introduce MaSpMM-Sym, a symmetric variant that leverages memory-aware recursive partitioning to improve sparse-side reuse. Finally, we propose the MaSpMM-Adap, an adaptive scheme that dynamically chooses between MaSpMM and MaSpMM-Sym according to matrix structure and dense-matrix width.

3.1 MaSpMM: The Memory-aware SpMM

We first analyze and evaluate the impact of matrix storage layouts on SpMM performance, and then present the design of MaSpMM.

3.1.1 Segment Storage. Different storage methods for sparse matrix **A** correspond to different cache reuse scenarios. Figure 2 illustrates three sparse matrix storage formats: *CSR*, *ASpT*, and *column-segment*. Both CSR and ASpT store **A** in row-major order, but ASpT processes **A** tile by tile to improve reuse of **B** [9]. Like J-Stream, we store the sparse matrix as continuous "*column-segments*" to improve cache reuse of the output matrix **C** (see Figure 2(e)). We compare the memory access volume of SpMM using the column-segment layout with that of the ASpT format. For clarity, tiling along the K dimension is temporarily ignored in the following analysis.

When sparse matrices are stored as column-segments, the memory access volume of **C** is $M \times K$ because **C** can be reused in the cache. Each element of sparse matrix **A** includes a value and a column index, so the access volume of **A** can be expressed as $2 \times NNZ$, where NNZ denotes the number of nonzeros in **A**. For each column-segment of **A**, each element of the input matrix **B** must be accessed once, so the theoretical upper limit of **B** accesses is $\frac{M \times N}{T_i} \times K$. In fact, the memory access of the **B** is only related to the number of *active column-segments* (i.e., column-segments containing non-zero elements), and only these column-segments will cause memory accesses of **B** [14]. The number of active column-segments in matrix **A**, $ActiveColSegs(T_i)$, depends only on the tile size (T_i). Therefore, the memory access volume of **B** is $ActiveColSegs(T_i) \times K$. Then, the total data transfer volume of SpMM when **A** is stored in column-segments is

$$M \times K \; + \; 2 \times NNZ \; + \; ActiveColSegs(T_i) \times K. \tag{1}$$

Similarly, when using ASpT, due to its cache reuse of **B**, its memory access volume is

$$N \times K \; + \; 2 \times NNZ \; + \; 2 \times ActiveRowSegs(T_j) \times K. \tag{2}$$

The last term represents the memory access volume of **C**, where $2 \times ActiveRowSegs(T_j)$ indicates that each *active row-segment* in **A** causes each row of **C** to be read and written once from memory. It can be seen that, compared to cache reuse **B**, the data transmission volume of reuse **C** is generally smaller. The article [14] also argues that, compared with reusing tiles of matrix **B** in the cache, the benefit of caching tiles of matrix **C** is more pronounced, since **C** is subject to both read and write operations. Therefore, we store matrix **A** in the column-segment layout.

The above analysis assumes that the column dimension K of the dense matrix is small enough to fit in fast cache (e.g., L2 cache). In J-Stream, the elements of each dense tile are not stored contiguously in memory (see Figure 2(d)). As a result, it suffers performance degradation when K is large [14]. In contrast to J-Stream, we store the dense matrix in contiguous row segments by tuning the T_k parameter. Consequently, each tile of **B** and **C** is stored contiguously, improving spatial locality and sustaining higher performance for large K.

3.1.2 MaSpMM Algorithm with Segment Storage. Based on the above analysis, we design the MaSpMM computation flow. Figure 2(e) shows the MaSpMM computation method. SpMM is performed on column segments of **A** and row segments of **B** to produce a tile of **C**. By appropriately choosing the tile sizes T_i and T_k, each **C** tile can reside in fast cache until its computation is completed.

Algorithm 4 presents MaSpMM, a novel tile-based SpMM framework. MaSpMM and J-Stream adopt different storage layouts, which lead to fundamentally different computation orders. J-Stream emphasizes the third-level loop as the key to minimizing data movement and argues that optimal access is achieved when this loop streams along the J dimension (Figure 2(d)) [14]. It further suggests that the ordering of the two outer tile loops has only a secondary effect,

ALGORITHM 4: MaSpMM Algorithm

Input: sparse matrix $A[M][N]$, dense matrix $B[N][K]$.
Output: dense matrix $C[M][K]$.

```
1  for k = 0 to ⌈K/T_k⌉ − 1 do
2      kbound = min((k + 1) · T_k, K − 1);
3      for i = 0 to ⌈M/T_i⌉ − 1 do in parallel
4          for jj = act_colSeg[i] to act_colSeg[i + 1] do
5              col = col_idx[jj];
6              for kk = k · T_k to kbound step vecLen do
7                  rowSeg_B = load_vector(B[col][kk : kk + vecLen]);
8                  for ii = colSeg_ptr[jj] to colSeg_ptr[jj + 1] do
9                      row = row_idx[ii];
10                     val = values[ii];
11                     A_broadcast = broadcast(val);
12                     rowSeg_C = load_vector(C[row][kk : kk + vecLen]);
13                     rowSeg_C += A_broadcast × rowSeg_B;
14                     store_vector(C[row][kk : kk + vecLen], rowSeg_C);
15                 end
16             end
17         end
18     end
19 end
```

since its storage scheme does not fully exploit locality from tiling along the K dimension. In contrast, MaSpMM employs a deterministic loop ordering that maximizes data reuse on the dense side.

We choose the first-level loop to iterate over the K dimension of the dense matrix tiles (Line 1), and the second-level loop to traverse the I dimension corresponding to the sparse matrix **A** tiles (Line 3). This ordering is motivated by the fact that accessing dense column blocks typically incurs higher memory pressure than traversing sparse row blocks of **A**. Consequently, MaSpMM prioritizes the reuse of dense column blocks. The third-level loop follows the same strategy as J-Stream, streaming along the J dimension. Since the streaming tile size is one [14] (i.e., $T_j = 1$), this loop is omitted in both J-Stream (see Algorithm 6 in [14]) and MaSpMM. Within each tile, the algorithm first traverses the active column-segments of the current row block to obtain their column indices (Lines 4–5). To maximize reuse of dense row-segments from matrix **B**, the next loop iterates over the row-segment elements within the selected dense column blocks (Line 6). Finally, the innermost loop iterates over the nonzeros of the active column-segment (Line 8), multiplies each value with the corresponding preloaded element from **B** (Line 7), and accumulates the result into the appropriate entry of the output tile in **C** (Line 14).

In Lines 11 to 14 of Algorithm 4, MaSpMM performs vectorized computation within each tile. The nonzero from **A** is first broadcast into a SIMD register (Line 11), and then the corresponding row segment of **C** is loaded into a vector register (Line 12). A vector fused multiply–add then operates on the broadcast value and the preloaded **B**'s row segment, accumulating into the corresponding row segment of the **C** tile, followed by a vector store (Lines 13–14). Since the row segment of **B** is preloaded during the innermost iterations (Line 7), multiple nonzeros sharing the same column index can reuse this segment. The **C** tile remains cache-resident until the computation over the row-segments and column-segments completes. This design reduces redundant accesses to memory and fully exploits SIMD instructions to improve throughput.

3.1.3 Tile Size Optimization. Since MaSpMM also streams along the J dimension in its third loop, we adopt J-Stream's "matrix signature" model for parameter selection—that is, we model total data movement to guide tuning. Detailed analysis is provided in Section 4 of [14]. Let ρ be the density of the sparse matrix $\mathbf{A}$, defined as the $\frac{NNZ}{M \times N}$. In MaSpMM, a $T_i \times T_k$ tile of $\mathbf{C}$, $T_i \times \rho$ nonzeros from the active column-segment of $\mathbf{A}$, and T_k elements from the row-segment of $\mathbf{B}$ are kept in fast memory. Thus, the capacity constraint is

$$T_i \times T_k \ + \ 2 \times T_i \times \rho \ + \ T_k \ \leq \ Cache, \tag{3}$$

where *Cache* is the capacity of the fast memory.

Our objective is to choose tile sizes T_k and T_i so as to minimize the total data movement cost:

$$\min_{T_k, T_i} \left\{ M \ + \ 2 \times \frac{NNZ}{T_k} \ + \ ActiveColSegs(T_i) \right\} K. \tag{4}$$

Note that Equation (4) is a variant of Equation (1) considering K-dimensional tiling. Since M and K are constants, they can be removed from the minimization objective. Thus, the analytic estimate of data movement is a function of active column-segments $ActiveColSegs(T_i)$ and T_k.

In the model implementation [11], T_i is enumerated as powers of two and T_k is restricted to multiples of 32. For a given T_i, we split the rows of A into $\lceil \frac{m}{T_i} \rceil$ row tiles and, for each tile, count the number of distinct column indices among its nonzeros (i.e., active column-segments). These tiles are processed in parallel, and the counts are reduced to obtain ActiveColSegs(T_i). Subject to the cache constraint in Equation (3), we select the (T_i, T_k) pair that minimizes the model-predicted data movement in Equation (4). This model-based tuning is performed offline only once per matrix (and per K), and the selected parameters can be reused in subsequent SpMM executions. By evaluating the estimated data movement across candidate tile sizes, MaSpMM can select the suitable T_i and T_k for a given sparse matrix.

3.2 Symmetric MaSpMM: Enhancing MaSpMM by Exploiting Symmetry

The above discussion outlines the general design of the MaSpMM algorithm. For symmetric matrices, exploiting matrix symmetry can further reduce memory accesses to the sparse matrix. This benefit becomes particularly significant when the dense columns is small, as sparse-side incur noticeable traffic in such cases. The analysis in Section 2.2 reveals the limitations of existing methods in handling symmetry-induced conflicts, which hinder both memory locality and parallel efficiency. Motivated by these observations, we propose the Symmetric MaSpMM (MaSpMM-Sym) algorithm and present its detailed parallel design in the following subsections.

3.2.1 Memory-aware Recursive Partitioning. For a given conflict graph of a sparse matrix, we propose a memory-aware recursive partitioning method. We first employ the **Breadth-First Search** (**BFS**) algorithm [15] to partition the conflict graph. The process begins by selecting an unassigned node as the seed for the current subgraph, followed by a BFS traversal. Since the symmetric kernel updates both the current and the symmetric positions, we constrain the nonzeros within each subgraph (i.e., the corresponding rows in the lower triangular part) to fit in the fast cache size. This ensures that each subgraph remains cache-resident during execution [20].

We partition the graph into *numParts* subgraphs, where *numParts* is set to an integer multiple of CPU cores, ensuring all subgraphs are evenly distributed across the processing cores. This differs from DC-SpMV [22], which simply sets *numParts* to the number of CPU cores. We then identify vertices whose neighbors span multiple subgraphs and collect them into a special subgraph, referred to as the *separator*. To keep the separator small while capturing high-conflict vertices, we prioritize moving vertices with the largest number of neighbors in adjacent subgraphs into the separator [20]. By extracting these high-conflict vertices, we isolate conflict-free subgraphs and expose parallelism

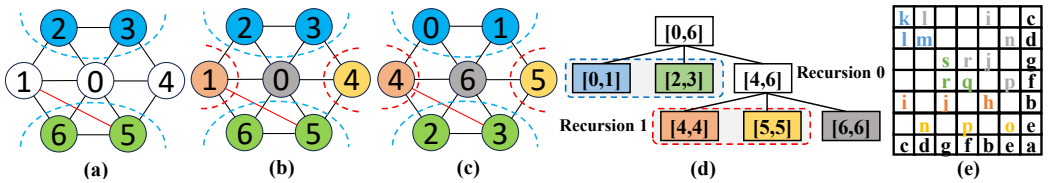


Fig. 5. (a) The conflict graph after the first partition, in which the initial separator is constructed; (b) The conflict graph after the second partition; (c) The reordered conflict graph after recursive partitioning; (d) The recursive tree corresponding to (c). The numbers in brackets indicate the starting and ending indices of the matrix rows. [4,6] and [6,6] are the separators for the first and second recursions, respectively; (e) The hybrid storage of A. The colored nonzeros are calculated by symmetric algorithm, and the black nonzeros are calculated by MaSpMM without using symmetry.

across subgraphs. An example is shown in Figure 5(a), where the conflict graph is isolated into two subgraphs during the first partitioning process (blue curve).

The separator undergoes recursive partitioning up to L_{sym} levels, beyond which further division is avoided. Here, L_{sym} is the optimal recursion depth, determined by the matrix structure and hardware characteristics. Figure 5(b) shows the conflict graph after two levels of partitioning ($L_{\text{sym}} = 2$), with blue and red curve marking subgraphs from each recursion. We then permute the vertex ordering by grouping each subgraph into a contiguous index interval and appending the separator interval at the end. As shown in Figure 5(c), this improves the memory locality of input/output matrices in SpMM. Figure 5(d) shows the corresponding recursive tree, where symmetric SpMM kernels can operate in parallel across subgraphs at the same level without data conflicts. Each subgraph corresponds to a contiguous range of matrix rows, with start and end indices recorded for subsequent parallel execution.

3.2.2 Hybrid Strategy of Symmetric MaSpMM. For the high-conflict separator, even multi-level partitioning yields only limited parallelism [5, 7, 22]. Therefore, we bound recursive partitioning by a parameter L_{sym} and adopt a hybrid symmetric SpMM strategy. Rows corresponding to conflict-free subgraphs are stored in column segments and processed using symmetry, whereas the high-conflict separator is handled using the general MaSpMM without using symmetry. When $L_{\text{sym}} = 0$, the symmetric approach reduces to standard MaSpMM (i.e., without recursive partitioning).

Algorithm 5 presents the hybrid MaSpMM-Sym implementation. For the first L_{sym} recursion levels, the symmetric kernel is executed in parallel over the subgraphs at each level, with `startRow` and `endRow` recording the row ranges of each subgraph (Lines 1–3). For each subgraph, the symmetric algorithm invokes MaSpMM to update both the original and symmetric entries (Line 4). At the final recursion level, the separator is computed using MaSpMM with full storage to maximize parallel efficiency (Line 8). Figure 5(e) illustrates the resulting hybrid storage scheme derived from Figure 5(d): subgraphs of different colors within the same recursion level execute the symmetric kernel without write conflicts, while the final separator is processed using the general-purpose MaSpMM.

3.2.3 Parameters Selection. The MaSpMM-Sym algorithm introduces two key parameters: the number of partitions *numparts* and the recursion depth L_{sym}. Although an exhaustive search over all ($numparts, L_{\text{sym}}$) combinations would identify the optimal configuration, it is prohibitively expensive in practice. Instead, we propose a low-overhead heuristic that efficiently approximates the optimal setting. For convenience, we define *parts* as the average number of subgraphs assigned to each CPU core; with p CPU cores, this gives $numparts = parts \times p$.

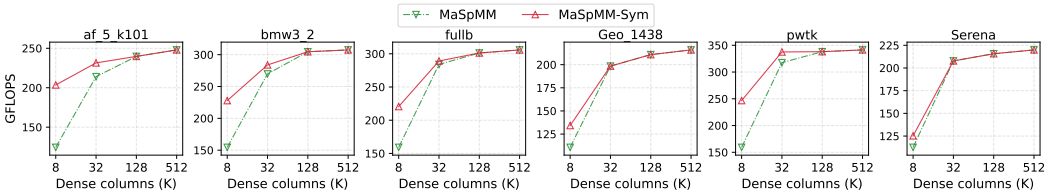


Fig. 6. Performance of MaSpMM and MaSpMM-Sym on x86.

ALGORITHM 5: MaSpMM-Sym Algorithm

Input: sparse matrix $A[M][N]$, dense matrix $B[N][K]$.
Output: dense matrix $C[M][K]$.
`/* Calculate the low-conflict part using symmetry */`
for $l = 0$ *to* $L_{sym} - 1$ **do**
  for $p = 0$ *to* $numParts - 1$ **do in parallel**
    for $i = startRow[l][p]$ *to* $endRow[l][p]$ **do**
      Using MaSpMM to calculate the current and symmetric positions of subgraph
    end
  end
end
`/* Calculate the high-conflict part using MaSpMM */`
Using MaSpMM to compute the separator

We perform a lightweight calibration during preprocessing to select $(parts, L_{\text{sym}})$ for a given matrix and dense dimension K. Specifically, we (i) scan $parts \in \{1, 2, 4, 8, \dots\}$ and stop when further increases no longer improve throughput, and (ii) fix $parts$ to the best observed value and increment L_{sym} starting from 1, stopping once deeper recursion yields no additional benefit. To reduce measurement noise, improvements smaller than a threshold ϵ are treated as negligible; ϵ can be absolute or relative (e.g., 3 GFLOP/s or 2% of throughput). If the best symmetric configuration does not outperform MaSpMM, we fall back to MaSpMM to avoid unnecessary symmetric overhead. In practice, this procedure evaluates only a small number of configurations and can be executed offline once for each matrix and dense width K. Across the tested symmetric matrices, shallow recursion (typically $L_{\text{sym}} < 4$) is sufficient, with deeper levels showing diminishing returns. Moreover, performance is relatively insensitive to $parts$ near the optimum, which motivates the early-termination heuristics and ensures robust parameter selection.

3.3 The Adaptive MaSpMM

3.3.1 Impact of K Dimension on MaSpMM-Sym. The design of MaSpMM-Sym aims to further improve MaSpMM by exploiting symmetry in sparse matrices. However, for some matrices, we observe that MaSpMM-Sym effectively degenerates to MaSpMM (i.e., $L_{\text{sym}} = 0$) as the dense dimension K increases. As shown in Figure 6, when $K = 8$, MaSpMM-Sym clearly outperforms MaSpMM, but its advantage diminishes as K grows, eventually matching MaSpMM's performance. This behavior arises because symmetric SpMM simultaneously updates (i, j) and its symmetric counterpart (j, i) (see Lines 11–12 in Algorithm 3), leading to non-contiguous accesses to the dense matrices. When dense-side traffic dominates memory bandwidth, this irregular access pattern negates the benefits of symmetry. In the next section, we introduce an adaptive scheme that selects the most appropriate SpMM strategy for different execution scenarios.

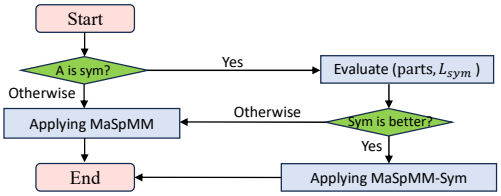


Fig. 7. The MaSpMM-Adap workflow.

Table 1. Input Matrices Used in Our Evaluation

Matrix	ID	Rows(M)	NNZ	NNZ/M	NNZ_{std}	$BW_{avg}/M(\%)$	C_{avg}
af_5_k101	0	503625	17550675	34.84	1.52	0.17	48.65
audikw_1	1	943695	77651847	82.28	42.45	43.82	615.63
bmw3_2	2	227362	11288630	49.65	13.83	4.46	90.86
boneS10	3	914898	55468422	60.62	20.38	0.76	123.85
fullb	4	199187	11708077	58.77	12.69	21.34	117.02
Ga41As41H72	5	268096	18488476	68.96	105.39	8.61	360.12
Geo_1438	6	1437960	63156690	43.92	4.49	1.33	87.85
inline_1	7	503712	36816342	73.09	35.63	22.64	424.94
ldoor	8	952203	46522475	48.85	11.98	14.02	106.30
pwtk	9	217918	11634424	53.38	4.76	3.87	79.06
Serena	10	1391349	64531701	46.38	9.25	3.99	113.45
CoupCons3D	11	416800	22322336	53.56	8.74	16.05	—
marine1	12	400320	6226538	15.55	2.42	3.83	—
radiation	13	223104	7637688	34.23	15.16	30.17	—
test1	14	392908	12968200	33.01	10.15	32.28	—

"—" denotes C_{avg} undefined for non-symmetric matrices.

3.3.2 Adaptive MaSpMM. We have introduced MaSpMM and its symmetric variant, MaSpMM-Sym. MaSpMM targets general SpMM, while MaSpMM-Sym exploits matrix symmetry to further improve performance. To handle diverse matrices robustly, we design an adaptive kernel-selection scheme. Figure 7 illustrates the MaSpMM-Adap workflow. Given an input matrix **A**, MaSpMM-Adap first checks whether **A** is symmetric. If not, it directly applies the general MaSpMM kernel. If **A** is symmetric, it evaluates a small set of ($parts, L_{sym}$) configurations and applies MaSpMM-Sym only if the best configuration outperforms MaSpMM; otherwise, it falls back to MaSpMM.

4 Experimental Results

4.1 Experimental Setup

Sparse Matrices. Table 1 summarizes 15 matrices from the SuiteSparse Collection [4], of which the first 11 are symmetric. These matrices span a wide range of sizes, sparsity patterns, and application domains, providing a diverse set of SpMM workloads. We report the following sparsity features: the number of rows (M), the total number of nonzeros (NNZ), the average nonzeros per row (NNZ/M), the standard deviation of nonzeros per row (NNZ_{std}), and the average bandwidth normalized by matrix size ($BW_{avg}/M(\%)$). To capture structural conflicts, we additionally report the average number of conflicts per row (C_{avg}), computed from the matrix's conflict graph.

Table 2. Detailed Configurations of Multi-Core CPUs

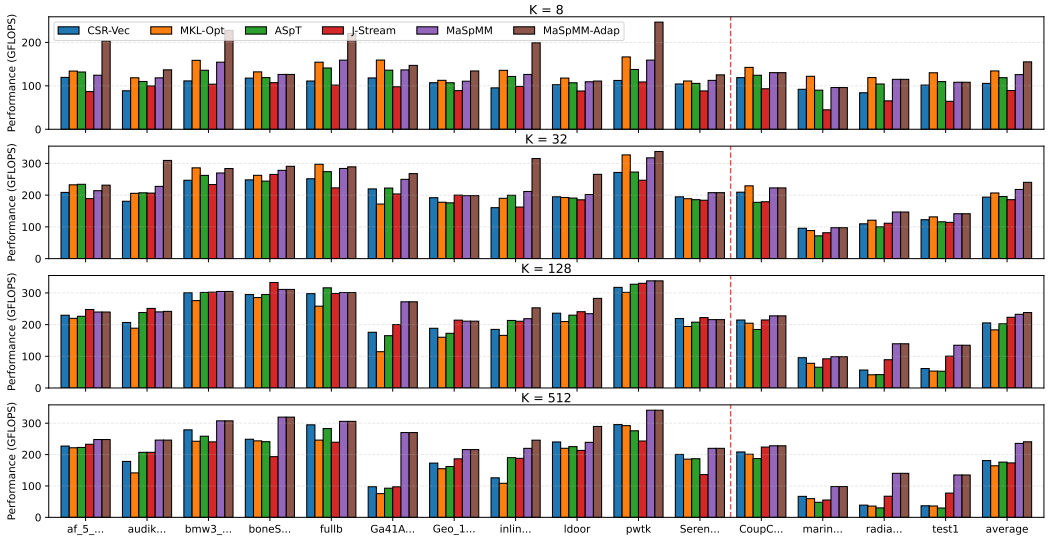
	Xeon Gold 6348	**Kunpeng 920**
Core frequency	2.6 GHz	2.6 GHz
Core number	28	64
#NUMAs	1	2
L1 cache (I/D)	80 KB	64 KB
L2 cache (per core)	1.25 MB	512 KB
L3 cache	42 MB	64 MB
SIMD	AVX-512	NEON-128

Table 3. Parameter Choices (T_i, T_k) for MaSpMM Under Different K

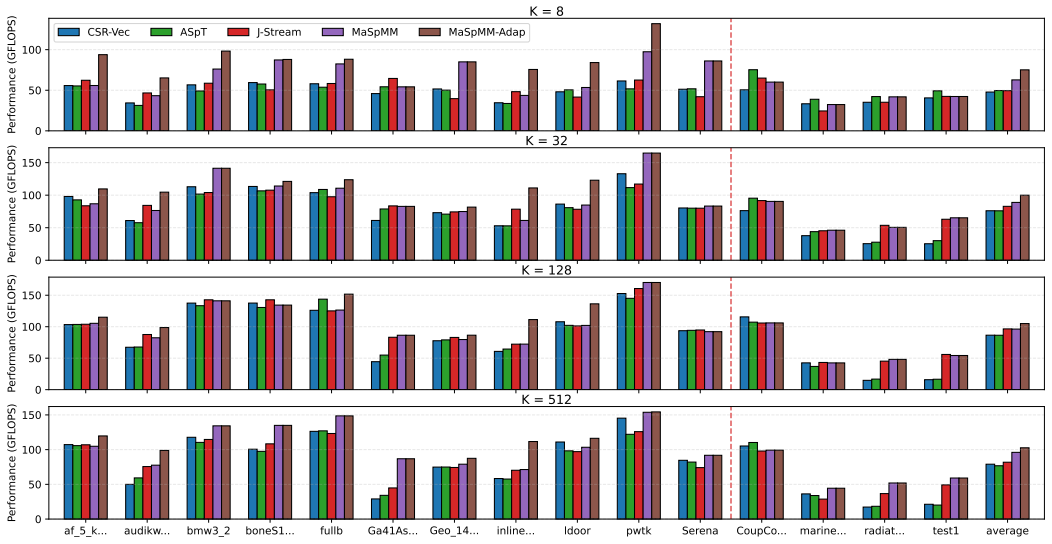
Matrix	$K=8$		$K=32$		$K=128$		$K=512$		$MaSpMM_{pre}$
	T_i	T_k	T_i	T_k	T_i	T_k	T_i	T_k	
af_5_k101	256	8	64	32	32	64	32	64	4
audikw_1	2,048	8	2,048	32	1,024	64	1,024	64	2
bmw3_2	2,048	8	1,024	32	512	64	512	64	4
boneS10	1,024	8	512	32	256	64	256	64	4
fullb	512	8	128	32	256	64	128	128	4
Ga41As41H72	2,048	8	1,024	32	512	64	512	64	5
Geo_1438	2,048	8	256	32	128	64	128	64	5
inline_1	1,024	8	1,024	32	1,024	64	1,024	64	3
ldoor	512	8	512	32	64	128	64	128	4
pwtk	2,048	8	1,024	32	32	64	32	64	4
Serena	1,024	8	512	32	256	64	256	64	5
CoupCons3D	2,048	8	2,048	32	32	64	32	64	4
marinel	2,048	8	512	32	256	64	32	64	4
radiation	2,048	8	2,048	32	2,048	32	2,048	32	4
test1	1,024	8	2,048	32	1,024	32	1,024	32	3

Hardware and Software. Our experiments are conducted on two multi-core platforms: an x86 system with an Intel Xeon Gold 6348 CPU (2.6 GHz) and an ARMv8 system with a Kunpeng 920 CPU (2.6 GHz). Detailed hardware specifications are provided in Table 2. All implementations use double-precision arithmetic and are compiled with the `-O3` optimization flag. We use Intel oneAPI 2023.0 for the x86 platform and GCC 9.3.0 for the ARM platform.

Methods for Evaluation. On the x86 platform, we compare our approach against the following SpMM implementations: ① **CSR-Vec**: a conventional CSR-based vectorized SpMM implementation; ② **MKL-Opt**: Intel oneMKL SpMM in Inspector–Executor mode, which first analyzes the matrix structure and then selects an optimized execution strategy; ③ **ASpT** from [9]; ④ **J-Stream** from [14]. On the ARM platform, we compare against CSR-Vec, ASpT, and J-Stream. MaSpMM adopts the matrix signature model from J-Stream to select tile sizes, while MaSpMM-Sym determines the parameters *parts* and L_{sym} using the heuristic described in Section 3.2.3. The selected parameter values on x86 are reported in Tables 3 and 4, and are further discussed in Section 4.2.5.



(a) Xeon Gold 6348



(b) Kunpeng 920

Fig. 8. Performance comparison of our approach with other SpMM methods on the two architectures.

4.2 Performance Evaluation

4.2.1 Performance of Our Approach. We evaluate dense dimensions $K \in \{8, 32, 128, 512\}$, which are commonly used in prior work [8, 9, 13, 14]. Figure 8 reports the performance of different methods on both platforms, with the red dashed line separating symmetric and nonsymmetric matrices. As shown in Figure 8(a), our approach consistently outperforms the others across a wide range of scenarios on x86. Compared with CSR-Vec, MaSpMM and MaSpMM-Adap achieve average speedups of 1.19× and 1.47× at $K = 8$, 1.13× and 1.25× at $K = 32$, 1.26× and 1.29× at $K = 128$, and 1.65× and 1.68× at $K = 512$, respectively. Without adaptive selection, MaSpMM attains the highest

Table 4. Parameter Choices ($parts, L_{sym}$) for MaSpMM-Sym Under Different K

Matrix	$K = 8$		$K = 32$		$K = 128$		$K = 512$		DC_{pre}	$Adap_{pre}$
	$parts$	L_{sym}	$parts$	L_{sym}	$parts$	L_{sym}	$parts$	L_{sym}		
af_5_k101	8	4	16	3	0	0	0	0	4	8
audikw_1	128	1	256	1	2,048	1	0	0	12	5
bmw3_2	1	2	4	2	0	0	0	0	3	8
boneS10	0	0	16	1	0	0	0	0	7	7
fullb	1	3	4	1	0	0	0	0	3	6
Ga41As41H72	64	1	64	1	0	0	0	0	10	7
Geo_1438	8	5	0	0	0	0	0	0	6	8
inline_1	1	4	512	1	512	1	8	1	9	5
ldoor	4	3	16	2	32	2	32	2	6	8
pwtk	2	3	4	2	0	0	0	0	2	7
Serena	8	3	0	0	0	0	0	0	8	9

average performance across most cases (rightmost bars), except when $K = 8$, where it ranks second to MKL-Opt.

As shown in Figure 8(a), when $K = 8$ and $K = 32$, MaSpMM-Adap delivers the highest performance on symmetric matrices among all methods. At small K, the dense operand exhibits strong cache locality, and the symmetric kernels further reduce memory traffic by caching and halving accesses on the sparse side. As summarized in Table 4, MaSpMM-Adap employs the symmetric kernel for most matrices in this regime (i.e., $L_{\text{sym}} > 0$). For symmetric matrices, MaSpMM-Adap (or MaSpMM-Sym) achieves speedups of 1.29× at $K = 8$ and 1.14× at $K = 32$ over MaSpMM. Matrices with large C_{avg} values, such as *audikw_1*, *inline_1*, and *Ga41As41H72*, tend to induce significant conflicts. To address this, MaSpMM-Sym adopts a hybrid strategy that applies different SpMM kernels to achieve high performance. Specifically, when $K = 8$ and $K = 32$, MaSpMM-Sym achieves speedups of 1.54× and 1.71× on *audikw_1*, 2.09× and 1.96× on *inline_1*, and 1.24× and 1.22× on *Ga41As41H72*, respectively, compared with CSR-Vec.

As K increases (e.g., $K = 128$ and $K = 512$), accesses to dense matrices progressively dominate memory bandwidth. In symmetric SpMM, non-contiguous memory accesses degrade locality, and the additional synchronization further limits parallel efficiency. Therefore, compared with MaSpMM-Sym, the general MaSpMM is more competitive due to its better locality and higher parallelism. As shown in Table 4, MaSpMM-Adap employs the MaSpMM kernel for most matrices at this stage (i.e., $L_{\text{sym}} = 0$). We observe that at $K = 512$, MaSpMM-Adap still selects the symmetric kernels for *inline_1* and *ldoor*.

On the ARM platform, our approach also demonstrates significant performance advantages. As shown in Figure 8(b), MaSpMM and MaSpMM-Adap consistently achieve the best average performance across different values of K. Compared to CSR-Vec, MaSpMM and MaSpMM-Adap deliver average speedups of 1.29× and 1.55× ($K = 8$), 1.28× and 1.44× ($K = 32$), 1.40× and 1.51× ($K = 128$), and 1.50× and 1.60× ($K = 512$), respectively. Relative to MaSpMM, MaSpMM-Adap achieves additional average speedups of 1.21× ($K = 8$), 1.14× ($K = 32$), 1.10× ($K = 128$), and 1.08× ($K = 512$). We observe that, compared with x86, MaSpMM-Adap selects symmetric kernels more frequently on ARM. For example, at $K = 512$ it also triggers on *af_5_k101*, *audikw_1*, and *Geo_1438*.

4.2.2 Performance of Other SpMM Methods. We also compare against other methods. When the dense column dimension is small, MKL-Opt performs particularly well: excluding our approach,

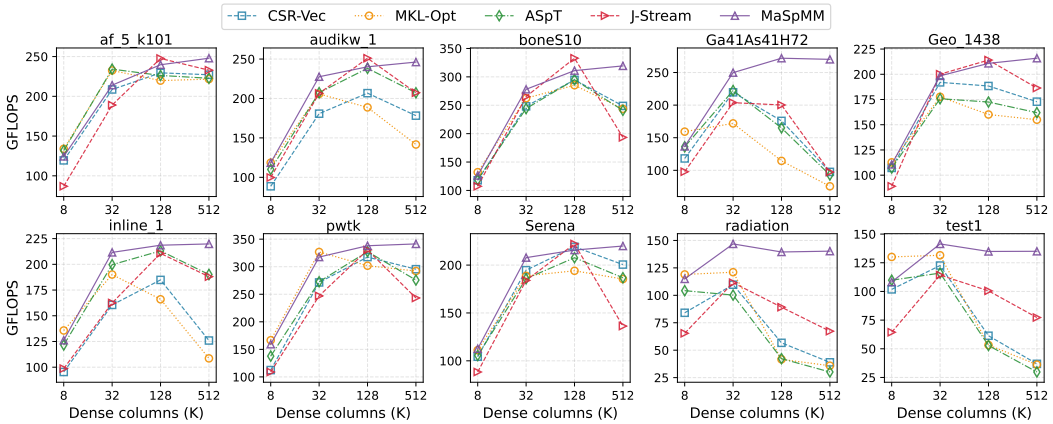


Fig. 9. Performance comparison of SpMM methods at different dense dimensions on x86.

it achieves the best performance at $K = 8$ and $K = 32$. However, its advantage diminishes as K increases due to growing memory pressure. The average performance of ASpT surpasses the CSR-Vec baseline only at $K = 8$ on x86. For other cases, ASpT shows limited competitiveness relative to the CSR-Vec on both x86 and ARM. When $K = 8$ or $K = 32$, J-Stream performs poorly because its implementation [11] pads dense operands for cache-line alignment, which introduces nontrivial overhead at small K. At $K = 128$, J-Stream performs comparably to MaSpMM; however, as K increases to 512, MaSpMM's enhanced dense-side locality yields a clear performance advantage over the other methods.

4.2.3 The Impact of K Dimension. This section examines the impact of the dense dimension K on SpMM performance. Figure 9 shows results for $K \in \{8, 32, 128, 512\}$ on x86. Overall, most prior methods peak at $K = 32$ or 128 but degrade at $K = 512$, especially on matrices such as *Ga41As41H72*, *inline_1*, *boneS10*, and *radiation*. As indicated by NNZ_{std} in Table 1, these matrices are relatively irregular, exhibiting notable variation in the number of nonzeros per row. In contrast, MaSpMM delivers the highest throughput and remains stable as K increases.

We observe that the impact of the dense dimension is less pronounced for relatively regular matrices. In particular, band matrices concentrate nonzeros around the main diagonal, improving locality and benefiting sparse computations [9, 31]. For instance, *af_5_k101* is highly banded (average bandwidth = 0.17% of the row dimension) and has low row-length variation ($NNZ_{\text{std}} = 1.52$). As shown in the Figure 9, prior methods exhibit only mild degradation as K increases, whereas MaSpMM still maintains a clear advantage at $K = 512$. By contrast, *boneS10* has a similarly narrow bandwidth (0.76% of the row dimension) but much larger row-length variation ($NNZ_{\text{std}} = 20.38$), causing existing SpMM methods to degrade more noticeably.

4.2.4 The Impact of Sparse Storage Format. From Figure 8 at $K = 8$ and $K = 32$, where no tiling is applied along the dense dimension K, the performance of CSR-Vec, ASpT, and MaSpMM highlights the impact of the sparse matrix $\mathbf{A}$'s format on general SpMM. Overall, the column-segment layout adopted by MaSpMM proves more favorable for general SpMM performance.

To further assess the impact of $\mathbf{A}$'s storage format on symmetric SpMM, we also implemented symmetric SpMM using CSR-based subgraph storage (SSS format) and ASpT, referred to as SymSpMM-CSR and SymSpMM-ASpT, respectively. Figure 6 shows that symmetric SpMM performs best at small dense dimensions; we therefore evaluate these methods at $K = 8$ and $K = 32$. As shown in

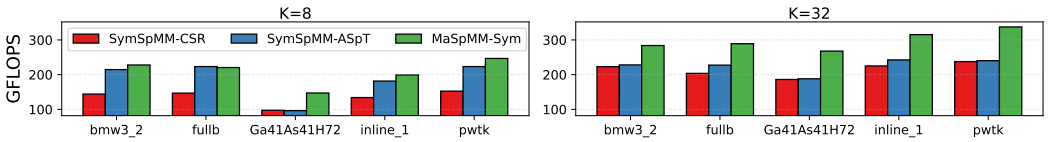


Fig. 10. Performance comparison of symmetric SpMM method on x86 when K is 8 and 32.

Figure 10, MaSpMM-Sym significantly outperforms both SymSpMM-CSR and SymSpMM-ASpT, validating the advantage of column-segment storage for symmetric SpMM.

4.2.5 Parameter and Preprocessing Evaluation. Tables 3 and 4 show the parameter selection of our method. From Table 3, MaSpMM selects a larger row-tile size T_i at small K to enhance dense-side cache reuse. Constrained by the cache capacity, T_i gradually decreases as K grows. The column-tile size T_k increases initially and then plateaus (e.g., around 64), because a larger T_k will not bring more caching benefits. Table 4 shows that MaSpMM-Sym favors deeper L_{sym} at small K, which reduces sparse-side traffic by recursive partitioning. As K grows, dense-side accesses dominate traffic, while the irregular access patterns and synchronization overhead of symmetric updates become more pronounced. Consequently, the optimal L_{sym} decreases and, for sufficiently large K, the method often falls back to the general SpMM formulation.

The last columns of Tables 3 and 4 report preprocessing overheads at $K = 128$, quantified as the equivalent number of parallel CSR SpMM iterations. $MaSpMM_{pre}$ measures the cost of converting sparse and dense matrices into segment layouts, while DC_{pre} denotes the overhead of recursive partitioning in MaSpMM-Sym. In addition, $Adap_{pre}$ reports the overhead of the adaptive parameter selection in MaSpMM-Adap, which evaluates a small number of candidate configurations to determine the most suitable strategy. Because these preprocessing steps are parallelized with OpenMP and performed offline only once per matrix, they incur negligible overhead for most matrices.

5 Related Work

In recent years, several optimization strategies have been proposed to improve SpMM performance. ASpT partitions the sparse matrix into row blocks and dynamically selects between 2D-tiled and CSR-like execution based on column-segment density, enhancing data reuse at the cost of added control complexity [9]. J-Stream explores tile-loop orderings and uses a signature-based model to select tile sizes according to predicted data movement [14]. Despite these advances, existing SpMM libraries often fail to sustain stable performance as the dense dimension K increases. To address this issue, we propose **MaSpMM**, a memory-aware SpMM algorithm that combines cache-aware tiling with a *segment-oriented* data layout. This design enables contiguous, cache-friendly accesses within each tile and keeps the corresponding **C** tile resident in cache during accumulation, yielding stable and scalable performance as K grows.

Modern CPUs and GPUs incorporate matrix acceleration units, such as NVIDIA Tensor Cores, POWER10 MMA, and Intel AMX [32]. However, real-world sparse matrices are often extremely sparse, leaving these dense-focused units underutilized [32]. Consequently, developing high-efficiency SpMV and SpMM kernels for such accelerators has become a key research direction [6, 17, 18, 25, 32, 34].

Symmetric sparse matrices offer substantial opportunities for memory optimization, but parallel execution often incurs write conflicts. Batista et al. [2] proposed a reduction-based symmetric SpMV for shared memory, where each thread updates only its private range, reducing traffic and

work; however, the reduction phase incurs memory traffic that grows linearly with thread count, limiting scalability on modern multicores [5]. Elafrou et al. introduced **Conflict-Free Symmetry** (**CFS**) SpMV [5], which applies a posterior coloring strategy that assigns contiguous matrix rows to cores before coloring, reducing the number of colors in the conflict graph. While effective for low-bandwidth matrices, CFS performs poorly on some high-bandwidth cases. Alappat et al. proposed the **Recursive Algebraic Coloring Engine** (**RACE**) [1], which leverages **Reverse Cuthill–McKee** (**RCM**) ordering to reduce matrix bandwidth and constructs a distance-2 coloring through multi-level row grouping. Qiu et al. [22] evaluated RACE on roughly 30 matrices and reported slightly better average performance than CFS; however, as shown in Figure 4, RACE underperforms CFS on several high-conflict matrices.

Recursive strategies have been widely used in parallel algorithms [10, 21, 30]. Thébault et al. proposed a Divide-and-Conquer approach for shared memory FEA matrix assembly on unstructured grids [26]. Qiu et al. proposed **Task Dependency Tree** (**TDT**) [19], which solves the data conflict and dependency problems in complex unstructured CFD applications. Qiu et al. also extended recursive methods to symmetric SpMV, addressing its write-conflict challenges [22]. These works inspired our approach. We propose a novel symmetric SpMM method that improves sparse-side data locality via *memory-aware* partitioning. Unlike reduction or coloring-based methods, we employ a hybrid recursive strategy, achieving higher adaptability across different symmetric matrices. Observing that MaSpMM and MaSpMM-Sym have complementary strengths depending on the matrix and dense dimension K, we further introduce MaSpMM-Adap, a lightweight selector that chooses the optimal kernel per matrix and K.

6 Conclusion

In this article, we propose MaSpMM, a memory-aware SpMM algorithm that enhances cache locality and scalability by coupling cache-aware tiling with a segment-oriented layout. Building on MaSpMM, we develop MaSpMM-Sym, which further improves performance by exploiting matrix symmetry through memory-aware partitioning and a hybrid execution strategy. Finally, we introduce MaSpMM-Adap, an adaptive framework that dynamically selects between MaSpMM and MaSpMM-Sym based on matrix characteristics. Extensive evaluations on both x86 and ARM CPUs demonstrate that the proposed methods consistently outperform state-of-the-art baselines across diverse matrix types and dense dimensions, achieving up to 1.86× average speedup over vendor-optimized libraries.

References

[1] Christie Alappat, Achim Basermann, Alan R. Bishop, Holger Fehske, Georg Hager, Olaf Schenk, Jonas Thies, and Gerhard Wellein. 2020. A recursive algebraic coloring technique for hardware-efficient symmetric sparse matrix-vector multiplication. *ACM Transactions on Parallel Computing* 7, 3 (2020), 1–37.

[2] Vicente H. F. Batista, George O. Ainsworth Jr., and Fernando L. B. Ribeiro. 2010. Parallel structurally-symmetric sparse matrix-vector products on multi-core processors. *arXiv preprint arXiv:1003.0952*.

[3] Aydin Buluç, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson. 2009. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the 21st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 233–244.

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

[5] Athena Elafrou, Georgios Goumas, and Nectarios Koziris. 2019. Conflict-free symmetric sparse matrix-vector multiplication on multicore architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*. 1–15.

[6] Ruibo Fan, Wei Wang, and Xiaowen Chu. 2024. Dtc-SpMM: Bridging the gap in accelerating general sparse matrix multiplication with tensor cores. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 253–267.

[7] Theodoros Gkountouvas, Vasileios Karakasis, Kornilios Kourtis, Georgios Goumas, and Nectarios Koziris. 2013. Improving the performance of the symmetric sparse matrix-vector multiplication in multicore. In *Proceedings of the 27th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 273–283.

[8] Changwan Hong, Aravind Sukumaran-Rajam, Bortik Bandyopadhyay, Jinsung Kim, Süreyya Emre Kurt, Israt Nisa, Shivani Sabhlok, Ümit V. Çatalyürek, Srinivasan Parthasarathy, and P. Sadayappan. 2018. Efficient sparse-matrix multi-vector product on GPUs. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*. 66–79.

[9] 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 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 300–314.

[10] Horowitz and Zorat. 1983. Divide-and-conquer for parallel processing. *IEEE Transactions on Computers* 100, 6 (1983), 582–585.

[11] HPCRL. Accessed: 2025-09. Jstream_SpMM. https://github.com/HPCRL/jstream_SpMM

[12] Takeshi Iwashita, Hiroshi Nakashima, and Yasuhito Takahashi. 2012. Algebraic block multi-color ordering method for parallel multi-threaded sparse triangular solver in ICCG method. In *Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 474–483.

[13] Peng Jiang, Changwan Hong, and Gagan Agrawal. 2020. A novel data transformation and execution strategy for accelerating sparse matrix multiplication on GPUs. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 376–388.

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

[15] Charles Eric Leiserson, Ronald L. Rivest, Thomas H. Cormen, and Clifford Stein. 1994. *Introduction to Algorithms*. MIT Press Cambridge, MA, USA.

[16] Haiyang Lin, Mingyu Yan, Xiaochun Ye, Dongrui Fan, Shirui Pan, Wenguang Chen, and Yuan Xie. 2023. A comprehensive survey on distributed training of graph neural networks. *Proceedings of the IEEE* 111, 12 (2023), 1572–1606.

[17] Yuechen Lu and Weifeng Liu. 2023. DASP: Specific dense matrix multiply-accumulate units accelerated general sparse matrix-vector multiplication. In *Proceedings of the 36th ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*. ACM/IEEE.

[18] Patrik Okanovic, Grzegorz Kwasniewski, Paolo Sylos Labini, Maciej Besta, Flavio Vella, and Torsten Hoefler. 2024. High performance unstructured SpMM computation using tensor cores. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*. IEEE, 1–14.

[19] Haozhong Qiu, Chuanfu Xu, Jianbin Fang, Liang Deng, Jian Zhang, Qingsong Wang, Yue Ding, Zhe Dai, Yonggang Che, Shizhao Chen, et al. 2024. Towards scalable unstructured mesh computations on shared memory many-cores. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 109–119.

[20] Haozhong Qiu, Chuanfu Xu, Jianbin Fang, Shengguo Li, Liang Deng, Jian Zhang, Zhe Dai, Yue Ding, Yue Wang, Zhimeng Han, et al. 2025. Me-MPK: Accelerating krylov subspace solvers via memory-efficient matrix-power kernel. In *Proceedings of the 62nd ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–7.

[21] Haozhong Qiu, Chuanfu Xu, Jianbin Fang, Jian Zhang, Liang Deng, Zhe Dai, Yue Ding, Yue Wang, Zhimeng Han, Yonggang Che, et al. 2025. Dcsolver: Accelerating sparse iterative solvers via divide-and-conquer on GPUs. *ACM Transactions on Architecture and Code Optimization* 22, 3 (2025), 1–25.

[22] Haozhong Qiu, Chuanfu Xu, Jianbin Fang, Jian Zhang, Liang Deng, Yue Ding, Qingsong Wang, Shizhao Chen, Yonggang Che, and Jie Liu. 2024. A conflict-aware divide-and-conquer algorithm for symmetric sparse matrix-vector multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*. IEEE, 1–15.

[23] Yousef Saad. 2003. *Iterative Methods for Sparse Linear Systems*. SIAM.

[24] Yousef Saad. 2011. *Numerical Methods for Large Eigenvalue Problems: Revised Edition*. SIAM.

[25] Jinliang Shi, Shigang Li, Youxuan Xu, Rongtian Fu, Xueying Wang, and Tong Wu. 2025. Flashsparse: Minimizing computation redundancy for fast sparse matrix multiplications on tensor cores. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 312–325.

[26] Loïc Thébault, Eric Petit, and Quang Dinh. 2015. Scalable and efficient implementation of 3D unstructured meshes computation: A case study on matrix assembly. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 120–129.

[27] Jan Treibig, Georg Hager, and Gerhard Wellein. 2010. LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of the 39th International Conference on Parallel Processing Workshops (ICPPW)*. IEEE, 207–216.

[28] Jan Winkelmann, Paul Springer, and Edoardo Di Napoli. 2019. ChASE: Chebyshev accelerated subspace iteration eigensolver for sequences of hermitian eigenvalue problems. *ACM Transactions on Mathematical Software* 45, 2 (2019), 1–34.

[29] Jing Xia, Chuanning Cheng, Xiping Zhou, Yuxing Hu, and Peter Chun. 2021. Kunpeng 920: The first 7-nm chiplet-based 64-core arm SoC for cloud services. *IEEE Micro* 41, 5 (2021), 67–75.

[30] Chuanfu Xu, Haozhong Qiu, Liang Deng, Jian Zhang, Xiang Gao, Jianbin Fang, Qingsong Wang, Yue Ding, Yue Wang, Zhimeng Han, et al. 2026. Efficient and scalable hybrid parallelization of unstructured computational fluid dynamics with geometric multigrid. *ACM Transactions on Architecture and Code Optimization* 23, 1 (2026), 1–24.

[31] Serif Yesil, Azin Heidarshenas, Adam Morrison, and Josep Torrellas. 2023. WISE: Predicting the performance of sparse matrix vector multiplication with machine learning. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 329–341.

[32] Serif Yesil, José E. Moreira, and Josep Torrellas. 2022. Dense dynamic blocks: Optimizing SpMM for processors with vector and matrix units using machine learning techniques. In *Proceedings of the 36th ACM International Conference on Supercomputing*. 1–14.

[33] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. 2018. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*. 974–983.

[34] Haisha Zhao, San Li, Jiaheng Wang, Chunbao Zhou, Jue Wang, Zhikuang Xin, Shunde Li, Zhiqiang Liang, Zhijie Pan, Fang Liu, et al. 2025. Acc-SpMM: Accelerating general-purpose sparse matrix-matrix multiplication with GPU tensor cores. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 326–338.

Received 1 November 2025; revised 1 February 2026; accepted 2 March 2026