Memory-Efficient Symmetry-Exploiting Sparse Triple Matrix Multiplication for Algebraic Multigrid

Ying Jin*, Fan Yuan*, Deshun Bi*, Shengguo Li†, Chuanfu Xu†, Zhong Liu†, Tiaojie Xiao, Chunye Gong, Li Zhao
College of Computer Science and Technology, National University of Defense Technology, Changsha 410073, China
{jinying,yuanfan,bds123,nudtlsg,xuchuanfu,zhongliu,xiaotiaojie,gongchunye,lizhao}@nudt.edu.cn

***Abstract*—Sparse triple matrix multiplication (RAP) is a key operation in the setup phase of algebraic multigrid (AMG) methods and often constitutes a major performance bottleneck. In many AMG applications, the system matrix is symmetric and the restriction and prolongation operators satisfy $R = P^T$. However, existing RAP implementations do not fully exploit these structural properties. In this paper, we propose a memory-efficient symmetry-exploiting RAP algorithm for AMG. The proposed method computes RAP using only one triangular part of the symmetric matrix, thereby reducing redundant computation and memory accesses. We further introduce a fused multiplication–transposition strategy that integrates transpose into sparse matrix multiplication to reduce memory traffic and transpose overhead. In addition, a load-balanced transpose strategy based on nonzero-aware row partitioning is employed to improve parallel efficiency for irregular sparse matrices on multicore architectures. Experiments on Intel- and ARM-based multicore platforms with 125 symmetric SuiteSparse matrices demonstrate that the proposed method consistently outperforms existing RAP implementations while significantly reducing memory-access overhead. Compared with baseline RAP implementations without symmetry exploitation, the proposed approach achieves an average speedup of up to $1.62\times$. Furthermore, it attains average speedups of $2.41\times$ over Intel MKL, $2.10\times$ over HYPRE-best, and $2.66\times$ over the row-fused method.**

***Index Terms*—SpGEMM, RAP, Symmetry, Transpose, AMG**

I. INTRODUCTION

Large-scale sparse linear systems arise in many scientific and engineering applications, including computational fluid dynamics [1], structural analysis [2], and electromagnetics [3]. Among existing approaches, algebraic multigrid (AMG) is widely regarded as one of the most effective iterative methods because it can achieve near-linear computational complexity while avoiding explicit dependence on geometric information [4]–[6].

AMG typically consists of a setup phase and a solve phase. During setup, coarse-grid operators are constructed through the Galerkin projection [7]

$$A_c = RAP,$$

where A is the fine-grid matrix, and R and P denote the restriction and prolongation operators, respectively. This RAP process is essentially a sparse triple matrix multiplication. Because RAP operates on highly irregular sparse matrices and may generate large intermediate data structures, it often becomes the primary performance and memory bottleneck in AMG setup [8]–[10]. A large body of work has focused on accelerating SpGEMM and RAP computations [11]–[14]. Among these approaches, SPA-based methods [15] are widely adopted because they provide robust performance and relatively low memory overhead. RAP can be computed via either two SpGEMM operations or fused RAP methods [16], [17], the latter reducing intermediate storage and redundant computation. However, existing methods remain limited by irregular memory accesses and underutilization of matrix structure.

In many AMG applications, the system matrix is symmetric and the operators satisfy $R = P^T$. However, none of the existing RAP implementations fully exploit this property, resulting in redundant computations and excessive memory traffic. Moreover, exploiting matrix symmetry often requires additional sparse transpose operations, which are inherently memory-bound due to their irregular memory access patterns. Due to memory-bandwidth limitations, their overhead can become increasingly significant in the overall execution runtime [18].

To address these challenges, we propose a memory-efficient symmetric RAP algorithm that explicitly exploits matrix symmetry and the relation between restriction and prolongation operators. The proposed method computes RAP using one triangular part of the symmetric matrix, thereby reducing redundant arithmetic operations and memory accesses. Furthermore, we introduce a fused multiply-transpose scheme that integrates transpose generation directly into sparse multiplication, avoiding additional transpose traversals and lowering memory traffic. To improve performance on irregular sparse matrices, we further develop a load-balanced transpose strategy based on nonzero-aware row partitioning, which alleviates thread imbalance and improves parallel efficiency for irregular sparsity patterns.

We incorporate the proposed techniques into both two-stage and single-phase RAP implementations, and evaluate them on Intel and ARM multicore platforms using 125 symmetric sparse matrices from the SuiteSparse Matrix Collection [19]. We compare our method with our baseline implementations without symmetry exploitation, as well as Intel MKL, HYPRE-best [16], and the row-fused RAP method [17]. Experimental results demonstrate that our optimized approach achieves consistent performance improvements, with an aver-

* These authors contribute equally to this research.
† Corresponding author.

age speedup of $1.62\times$ over the baseline. On the Intel platform, it further achieves average speedups of $2.41\times$ over Intel MKL, $2.10\times$ over HYPRE-best, and $2.66\times$ over the row-fused method.

This paper makes the following contributions:

- We present a RAP algorithm that exploits matrix symmetry and the relation $R = P^T$ to avoid redundant computations and reduce memory traffic.
- We develop a fused multiply-transpose scheme that further reduces transpose overhead and memory-access costs.
- We propose a load-balanced transpose strategy based on nonzero-aware row partitioning to improve parallel efficiency for irregular sparse matrices.

II. BACKGROUND

A. AMG and RAP

Solving large-scale sparse linear systems is a fundamental challenge in scientific and engineering computing. Multigrid methods are widely regarded as highly effective solvers for such problems because they reduce error components across multiple scales and can achieve near-linear computational complexity. Algebraic multigrid (AMG) [4], [5], a prominent multigrid variant, eliminates the dependence on geometric mesh information by constructing a multilevel hierarchy directly from the algebraic properties of the coefficient matrix. This makes AMG particularly attractive for applications where geometric data are unavailable or difficult to exploit.

AMG typically consists of two phases, *setup* and *solve*: the former builds the hierarchy and associated operators, whereas the latter accelerates convergence through smoothing on the fine level and correction on coarse levels. The quality of the coarse levels largely determines both convergence and efficiency, and the construction of the coarse-grid matrix is especially critical [20], [21]. Since geometric coarsening is not applicable, AMG commonly adopts the Galerkin projection principle [7] and forms the coarse-grid operator via the triple sparse matrix product RAP. This RAP procedure comprises sparse matrix-matrix multiplications (SpGEMM) and is often the dominant kernel in the setup phase, thereby limiting parallel scalability. Consequently, optimizing RAP is essential for improving the efficiency and scalability of AMG.

B. SpGEMM Method

Sparse matrix–matrix multiplication (SpGEMM) is the core computational kernel of the RAP procedure in AMG. Its efficiency, memory usage, and numerical robustness directly affect RAP construction and overall AMG performance. Over the past decades, extensive research has led to a variety of SpGEMM algorithms targeting different sparsity patterns and hardware architectures. Among them, the Gustavson algorithm [22] is one of the earliest and most representative approaches. It adopts a row-oriented traversal strategy and computes the product using the triple-nested loop structure shown in Fig. 1 and Algorithm 1. Based on this formulation, many optimized methods have been developed, including sparse accumulator (SPA)-based techniques [15], hash-based accumulation [23], expand–sort–compress (ESC) strategies [24], and heap-based merge schemes [23], to improve performance on irregular sparse matrices and modern hardware platforms.

Algorithm 1 SPA SpGEMM.

Input: $A \in \mathbb{R}^{M\times K}, B \in \mathbb{R}^{K\times N}$
Output: $C = AB \in \mathbb{R}^{M\times N}$
Allocate the `SPA[0:N]` and the `marker[0:N]`
for $i = 0$ **to** $M - 1$ **do**
  Initialize the `SPA[:]` to 0
  Initialize the `marker[:]` to `FALSE`
  for all nonzero A_{ik} in $A_{i,:}$ **do**
    for all nonzero B_{kj} in $B_{k,:}$ **do**
      if `marker[j]` = `FALSE` **then**
        // Accessing this place first time
        `marker[j]` = `TRUE`
        `SPA[j]` = $A_{ik} \times B_{kj}$
      else
        `SPA[j]` += $A_{ik} \times B_{kj}$
      end if
    end for
  end for
  Save `SPA` to $C_{i,:}$
end for

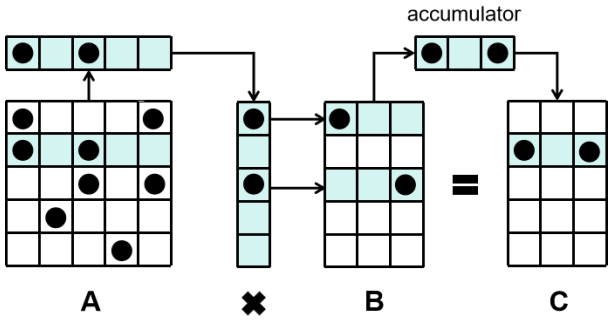


Fig. 1: The row-wise Gustavson algorithm.

The SPA algorithm is a canonical optimization derived from the Gustavson formulation and is widely regarded as one of the most effective accumulator-based strategies for SpGEMM. Its key refinement is to assign an independent accumulator to each output row and to store only the column indices and partial sums associated with nonzero contributions generated in that row, thereby avoiding unnecessary access to and storage of zero entries. Conceptually, the SPA-based procedure can be organized into three stages: preprocessing, accumulation, and output assembly. By selectively tracking only the active columns, SPA reduces memory traffic and eliminates redundant checks, yielding strong performance for products of moderately sparse matrices and making it well suited to typical RAP workloads in AMG. Algorithm 1 outlines the SpGEMM algorithm using SPA.

Fig. 2 compares several representative SpGEMM algorithms. HASH, HASH_VEC, and HEAP follow [23], while ESC corresponds to PBSpGEMM [24]. Among them, the SPA-

Algorithm 2 Single-phase RAP.

1: **Input:** $R \in \mathbb{R}^{M\times K}, A \in \mathbb{R}^{K\times K}, P \in \mathbb{R}^{K\times M}$
2: **Output:** $C = RAP \in \mathbb{R}^{M\times M}$
3: **for** $i = 0$ **to** $M - 1$ **do**
4: Initialize $C_{i,:}$ to 0
5: **for all** nonzero R_{ik} in $R_{i,:}$ **do**
6: **for all** nonzero A_{kl} in $A_{k,:}$ **do**
7: tmp = $R_{ik} \times A_{kl}$
8: **for all** nonzero P_{lj} in $P_{l,:}$ **do**
9: C_{ij} += tmp $\times P_{lj}$
10: **end for**
11: **end for**
12: **end for**
13: **end for**

based method provides the most robust and consistently high performance across the tested matrices. Hence, it is used as the baseline in this work.

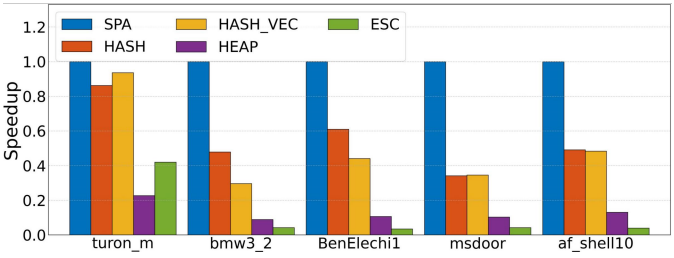


Fig. 2: Comparison of different accumulators.

C. Two-stage and Single-phase RAP

RAP algorithms used in practice can generally be classified into two categories: two-stage algorithms and single-phase algorithms. A two-stage algorithm computes RAP in two consecutive steps by first forming $B = RA$ and then computing $C = BP$. This is the most straightforward approach to evaluating the RAP product.

In contrast, single-phase algorithms combine the two multiplications into a single procedure [16], reducing intermediate memory allocation and, in some cases, lowering the number of floating-point operations. The classical implementation in HYPRE adopts such a single-phase strategy by avoiding the explicit construction of the intermediate matrix. Algorithm 2 illustrates the single-phase RAP procedure used in HYPRE. Park et al. [17] further proposed a single-phase variant that immediately uses each computed row B_i to form the corresponding row C_i, thereby reducing the overall operation count.

Our experiments evaluate both the two-stage RAP algorithm and the classical HYPRE-based single-phase RAP algorithm. Hereafter, *single-phase method* refers to the classical HYPRE-style single-phase RAP approach.

III. Approach

We propose a memory-efficient strategy for symmetric triple matrix multiplication that supports both two-stage and single-phase RAP algorithms. For clarity, we present the two-stage case. By exploiting the symmetry of A and fusing transposition with multiplication, the proposed method reduces redundant computation, memory traffic, and overall execution time. For matrices with highly irregular sparsity patterns, we further introduce a load-balanced transpose strategy based on nonzero-aware row partitioning to mitigate workload imbalance and improve parallel efficiency.

A. Symmetric RAP algorithm

Our algorithm targets symmetric sparse matrices and accelerates the overall computation by explicitly exploiting matrix symmetry. The method is designed for the symmetric algebraic multigrid (AMG) setting. Given that the fine-level operator A is symmetric, we store and access one triangular part of A during the computation.

In the AMG coarsening phase, the restriction and prolongation operators are assumed to satisfy $R = P^{\mathrm{T}}$. By exploiting both this relation and the symmetry of $A = L + L^T$, where L consists of the strictly lower-triangular part of A and half of its diagonal entries, the proposed method computes the triple product using only L,

$$C = RAP = P^T(L + L^T)P = P^T LP + (P^T LP)^T.$$

Fig. 3 illustrates the overall workflow. In the figure, L represents the stored lower-triangular part of A, including the full diagonal entries. Before RLP computation, its diagonal entries are scaled by $1/2$ so that $A = L + L^T$ holds exactly. Path 1 (ⓐ–ⓑ–ⓓ–ⓔ) first computes RLP and partitions the intermediate result into lower- and upper-triangular parts, denoted by S_L and S_U. The upper part S_U is then transposed to obtain S_U^T, after which S_L and S_U^T are accumulated to form the lower-triangular output matrix C_L. By avoiding redundant computations and transposing only S_U rather than the full matrix, this approach reduces both memory overhead and transpose cost. Path 2 (ⓐ–ⓒ–ⓓ–ⓔ) further fuses the transpose operation into the computation, directly generating S_L and S_U^T. Details are presented in the following sections.

As illustrated in Fig. 3, black entries represent complete contributions, while gray entries represent partial contributions arising from the use of a single triangular part of the symmetric matrix. Since S_L and S_U^T have irregular and only partially overlapping sparsity patterns, the final lower-triangular matrix cannot be reconstructed by simple scaling. Therefore, we explicitly accumulate S_L and S_U^T, with diagonal entries handled as full values.

Algorithm 3 presents an SPA-based implementation of the triple product using two SpGEMM operations, where the input L is the stored lower-triangular part of A. Line 3 scales the diagonal entries of L by $1/2$ to ensure $A = L + L^T$. Lines 4–5 compute RLP and partition the intermediate result into S_L and S_U. Line 6 performs a load-balanced transpose of S_U to obtain S_U^T, while line 7 accumulates S_L and S_U^T to form the final result C_L.

The transpose stage is memory-bound and sensitive to sparsity irregularity, and thus can become a performance bottleneck in symmetric RAP computation. To improve parallel efficiency, Algorithm 4 adopts a nonzero-aware load-balancing

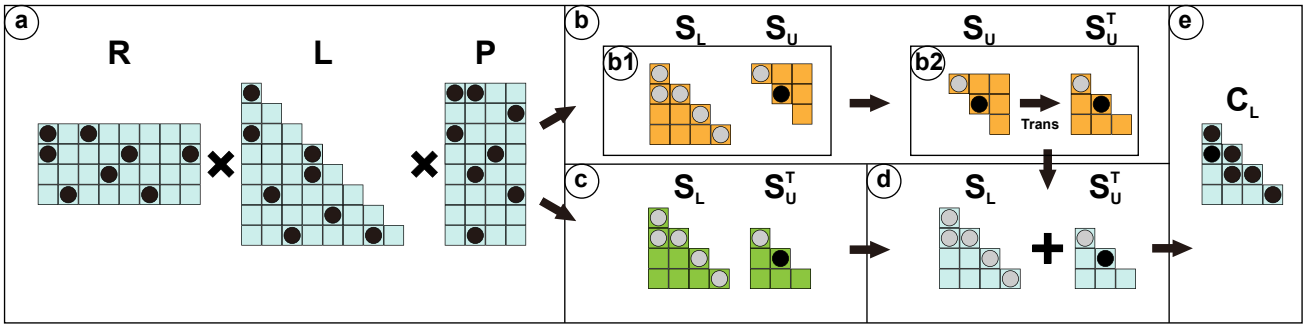


Fig. 3: Overall workflow of the proposed algorithm. The computation is performed using the lower-triangular part of the symmetric matrix A. By exploiting matrix symmetry and transpose operations, the final result can be obtained via two alternative paths. Path 1 is ⓐ–ⓑ–ⓓ–ⓔ, and Path 2 is ⓐ–ⓒ–ⓓ–ⓔ.

Algorithm 3 Symmetric RAP.

1: **Input:** $R \in \mathbb{R}^{M\times K}$, $L \in \mathbb{R}^{K\times K}$, $P \in \mathbb{R}^{K\times M}$
2: **Output:** $C_L \in \mathbb{R}^{M\times M}$
3: $L_{ii} \leftarrow \frac{1}{2}L_{ii}, \quad i = 1, \ldots, K$
4: RL = SPA_SpGEMM(R, L)
5: $[S_L, S_U]$ = SPA_SpGEMM_LU(RL, P)
6: S_U^T = Transpose(S_U)
7: $C_L = S_L + S_U^T$

strategy. The key step is Line 5, where `nnz_balanced` partitions the input matrix by the number of nonzeros rather than by the number of rows. The resulting `RowStart` and `RowEnd` arrays are used in the subsequent parallel counting and transposition phases to reduce thread imbalance. We compare MKL, `scanTrans` [18], and `mergeTrans` [18] with our two optimized implementations, `OurTrans_V0` and `OurTrans_V1`. The latter additionally incorporates the nonzero-aware load-balancing optimization. `OurTrans_V0` already outperforms existing methods on most matrices by using per-thread private counting arrays `cnt` and direct offset-based writes, while `OurTrans_V1` further improves load balance. As shown in Fig. 4, `OurTrans_V1` achieves the best performance on most matrices.

Overall, the procedure consists of three steps: computing the triple product RLP, transposing S_U to obtain S_U^T, and accumulating the two terms to form the output. By exploiting matrix symmetry and the relation $R = P^{\mathrm{T}}$, the proposed approach eliminates redundant computations and improves efficiency.

B. Fused Multiplication and Transposition

To further reduce memory traffic and eliminate redundant operations, we fuse the transposition with the multiplication. As shown in Fig. 3, the final result can also be obtained through Path 2 (ⓐ–ⓒ–ⓓ–ⓔ). The operations ⓐ–ⓓ–ⓔ are shared by both paths, while the difference lies in steps ⓑ and ⓒ. In Path 1, it first produces S_L and S_U, and then explicitly transposes S_U to obtain S_U^T. In contrast, Path 2 fuses the transpose operation into the multiplication, directly generating S_U^T in step ⓒ. This avoids both the storage overhead of S_U and the cost of a separate transpose step.

Algorithm 4 Load-Balanced Sparse Matrix Transposition.

1: **Input:** $U \in \mathbb{R}^{M\times M}$
2: **Output:** $U^T \in \mathbb{R}^{M\times M}$
3: `NumThread` = `omp_get_max_threads()`
4: Initialize $cnt[:] = 0$
5: [`RowStart`[:], `RowEnd`[:]] = `nnz_balanced`(U)
6: **#pragma omp parallel for**
7: **for** $t = 0$ **to** `NumThread` $- 1$ **do**
8: **for** i = `RowStart[t]` **to** `RowEnd[t]` **do**
9: **for all** nonzero U_{ij} in $U_{i,:}$ **do**
10: $cnt[M \times t + j]$++
11: **end for**
12: **end for**
13: **end for**
14: Execute `Prefixsum`(cnt)
15: **#pragma omp parallel for**
16: **for** $t = 0$ **to** `NumThread` $- 1$ **do**
17: **for** i = `RowStart[t]` **to** `RowEnd[t]` **do**
18: **for all** nonzero U_{ij} in $U_{i,:}$ **do**
19: pos = $cnt[M \times t + j]$++
20: $U^T[pos] = U_{ij}$
21: **end for**
22: **end for**
23: **end for**

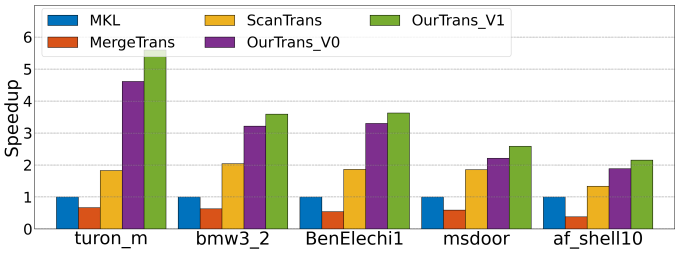


Fig. 4: Performance comparison of five transpose methods on the Intel platform using five matrices. The x-axis denotes the matrix index, and the y-axis shows the speedup.

In Algorithm 3, multiplication and transposition are executed sequentially in lines 5-–6. To reduce memory traffic and redundant operations, we fuse these two stages into a single kernel. Algorithm 5 presents the fused procedure. Here, for

a sparse matrix X, ptr_X, indx_X, and val_X denote the CSR row pointer, column index, and value arrays, respectively. For example, ptr_S_L denotes the row pointer array of S_L. The variable pos_X denotes the current write position in the corresponding output array. Line 4 constructs the auxiliary array cnt, which will store per-thread information for the transposed output. Line 5 scales the diagonal entries of L by $1/2$ to ensure $A = L + L^T$. After computing RL in line 6, Lines 7—39 generate S_L and S_U^T by combining the multiplication with transpose generation. Line 8 performs the symbolic phase, similar to lines 5-–13 of Algorithm 4. This phase estimates the nonzero distributions of S_L and S_U^T, storing the corresponding structural information in ptr_S_L and cnt, respectively. Lines 10-–39 correspond to the numeric phase. During computation, a shared marker array is reused for both triangular parts. For S_L (lines 17-–25), marker records inserted positions for fast lookup and duplicate detection. For S_U^T (lines 26-–34), marker uses a dedicated negative tag to indicate visited entries, while cnt provides direct access to the target positions in S_U^T. Together, the reuse of marker and the introduction of cnt enable efficient multiply-transpose fusion.

By fusing multiplication with transposition and exploiting symmetry, we eliminate the explicit construction of S_U. This further reduces memory traffic and auxiliary overhead, leading to improved overall performance.

IV. EXPERIMENT

A. Experimental Setup

Our experiments are conducted on both an x86 platform and an ARM platform. Table I summarizes the hardware configurations used in our evaluation, including an Intel Xeon Gold 6348 CPU and a Phytium (FT) 2000+ CPU [25]. All implementations are compiled in double precision with the -O3 optimization flag enabled. On the x86 platform, we use the Intel oneAPI Compiler (version 2023.0), while on the ARM platform we compile with GCC 9.3.0.

We primarily focus on single-node performance when comparing different algorithms. On the x86 platform, all experiments are conducted with 28 threads, whereas on the ARM platform we use 32 threads.

We evaluate our method on 125 symmetric sparse matrices from the SuiteSparse Matrix Collection [19]. All selected matrices have more than 5,000 rows, and their average number of nonzeros per row exceeds 5. In our experiments, the restriction and prolongation operators R and P are constructed from the first-level coarsening produced by an aggregation-based AMG strategy. We focus on the first level because it contains the largest fine-grid operator in the AMG hierarchy, providing the most demanding matrix-scale setting for evaluating the computational and memory-access efficiency of the RAP kernel. This also ensures a consistent kernel-level evaluation across all test matrices. The performance behavior across deeper AMG levels, where coarse operators become smaller but may exhibit increased density, will be investigated as part of our future end-to-end AMG study. In addition,

Algorithm 5 Symmetric RAP with Multiply-Transpose Fusion.

1: **Input:** $R \in \mathbb{R}^{M\times K}$, $L \in \mathbb{R}^{K\times K}$, $P \in \mathbb{R}^{K\times M}$
2: **Output:** $C_L \in \mathbb{R}^{M\times M}$
3: NumThread = omp_get_max_threads()
4: Initialize cnt[:]=0, $marker$[:]=-1
5: $L_{ii} \leftarrow \frac{1}{2}L_{ii}, \quad i = 1, \ldots, K$
6: RL = SPA_SpGEMM(R, L)
7: [RowStart[:], RowEnd[:]] = nnz_balanced(RL)
8: Do the symbolic phase and obtain the cnt array
9: // Numerical phase
10: **#pragma omp parallel for**
11: **for** $t = 0$ **to** NumThread $- 1$ **do**
12: **for** $i =$ RowStart[t] **to** RowEnd[t] **do**
13: pos_S_L = ptr_S_L[i]
14: **for all** nonzero RL_{ik} in $RL_{i,:}$ **do**
15: **for all** nonzero P_{kj} in $P_{k,:}$ **do**
16: temp = $RL_{ik} \times P_{kj}$
17: **if** $j \leq i$ **then**
18: **if** marker[$M \times t + j$] $<$ ptr_S_L[i] **then**
19: marker[$M \times t + j$] = pos_S_L
20: indx_S_L[*pos_*S_L] = j
21: val_S_L[*pos_*S_L] = temp
22: pos_S_L++
23: **else**
24: val_S_L[marker[$M \times t + j$]] += temp
25: **end if**
26: **else**
27: **if** marker[$M \times t + j$] != $-i - 2$ **then**
28: marker[$M \times t + j$] = $-i - 2$
29: pos_U^T = cnt[$M \times t + j$]++
30: indx_S_U^T[pos_U^T] = i
31: val_S_U^T[pos_U^T] = temp
32: **else**
33: val_S_U^T[cnt[$M \times t + j$] $- 1$] += temp
34: **end if**
35: **end if**
36: **end for**
37: **end for**
38: **end for**
39: **end for**
40: $C_L = S_L + S_U^T$ // Accumulate

we further select 15 representative matrices for an in-depth performance analysis.

To demonstrate the generality of the proposed approach, we incorporate it into both the two-stage RAP formulation and the HYPRE-style single-phase RAP procedure, and evaluate their performance through comparative experiments.

B. Overall Performance

We apply our optimizations to both the two-stage and single-phase RAP implementations, and evaluate them on 125 matrices on Intel and ARM platforms. The baseline is our straightforward SPA-based RAP implementation without symmetry exploitation or multiply-transpose fusion. In Fig. 5,

TABLE I: Detailed configurations of multicore CPUs.

	Xeon Gold 6348	FT 2000+
Core frequency	2.6 GHz	2.2 GHz
Core number	56	64
#NUMAs	2	8
L1 cache (I/D)	80 KB	32 KB
L2 cache (per core)	1.25 MB	2 MB
L3 cache	42 MB	None

Algorithm 3 represents the symmetry-exploiting method, while Algorithm 5 further incorporates multiply-transpose fusion. The y-axis reports the speedup over the baseline implementations, while the x-axis denotes the matrix index. The matrices are ordered by increasing number of rows.

Fig. 5a and 5b summarize the results on the Intel platform for the two-stage and single-phase RAP implementations. Relative to the baseline two-stage RAP, leveraging matrix symmetry achieves an average speedup of $1.41\times$, and multiply-transpose fusion further increases it to $1.52\times$. For the single-phase RAP, leveraging matrix symmetry delivers $1.43\times$ speedup on average, and fusion yields $1.56\times$. Overall, both optimizations provide consistent improvements for most matrices.

Fig. 5c and 5d present the corresponding results on the ARM platform. Compared with the baseline two-stage RAP, leveraging matrix symmetry yields an average speedup of $1.49\times$, which is further improved to $1.59\times$ with multiply-transpose fusion. For the single-phase RAP, leveraging matrix symmetry attains $1.25\times$ on average, and fusion improves it to $1.62\times$. These results indicate that our approach also performs well on ARM.

Across both platforms, leveraging matrix symmetry accelerates the vast majority of matrices. In contrast, multiply-transpose fusion is generally more effective for small to medium matrices, while its benefit becomes less significant on large matrices. The main advantage of fusion is that it avoids explicit materialization and separate traversal of S_U during transpose construction. However, the transpose metadata update is moved into the inner SpGEMM computation loop. When the output structure becomes highly irregular or the candidate columns are widely scattered, the associated `cnt` and marker updates exhibit increasingly random access patterns. For large matrices, the cost of these irregular accesses may outweigh the savings from eliminating the explicit transpose stage. In contrast, the standalone transpose operation processes memory in a more regular streaming manner, making it more cache-friendly and allowing the metadata construction overhead to be amortized more effectively.

Table II further summarizes the contribution of each optimization under the two-stage and single-phase RAP frameworks. The results show that exploiting matrix symmetry consistently provides the dominant performance improvement across both platforms. In contrast, multiply-transpose fusion offers additional acceleration, although its effectiveness is more sensitive to matrix characteristics and hardware architectures. Overall, the ablation study confirms that the proposed optimizations complement each other and together contribute to the overall performance improvement.

TABLE II: Ablation study of different optimization techniques in RAP computation. The reported values are the average speedups over the baseline implementation on each platform.

Framework	Method	Intel	FT2000+
Two-stage	Baseline RAP	1.00×	1.00×
	+ Symmetry Exploitation	1.41×	1.49×
	+ Multiply-Transpose	1.52×	1.59×
Single-Phase	Baseline RAP	1.00×	1.00×
	+ Symmetry Exploitation	1.43×	1.25×
	+ Multiply-Transpose	1.56×	1.62×

C. Performance Comparison with Other Methods

We further evaluate 125 matrices on the Intel platform, comparing our approach with Intel MKL, HYPRE and the row-fused RAP method [17]. We use MKL version 2023.0.0 as the optimized vendor baseline. For HYPRE (v3.1.0), we benchmark two RAP routines, `hypre_BoomerAMGBuildCoarseOperatorKT` and `hypre_ParCSRMatrixRAPKT`. For each test case, we report the better performance achieved by the two routines, denoted as `HYPRE-best`.We also include the row-fused method as a RAP-specific baseline.

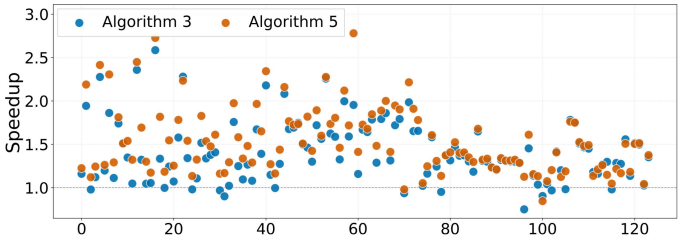
We compare MKL, `HYPRE-best` and the row-fused RAP method with our multiply-transpose fusion implementations in both the two-stage and single-phase RAP variants. Fig. 6 reports the speedups relative to MKL. Overall, the proposed methods outperform MKL, HYPRE-best, and the row-fused method on most of the tested matrices.

The optimized two-stage and fused variants achieve average speedups of $1.51\times$ and $2.41\times$ over MKL, and $1.31\times$ and $2.10\times$ over `HYPRE-best`, respectively. Compared with the row-fused method, the average speedups are $1.54\times$ for the two-stage variant and $2.66\times$ for the fused variant. These results demonstrate the effectiveness of the proposed optimizations.

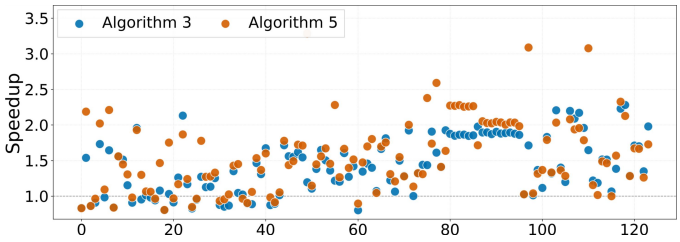
D. Performance Comparison on Representative Matrices

We further select 15 representative matrices for detailed analysis. Their characteristics are summarized in Table III. Using these matrices, we conduct experiments on both Intel and ARM platforms. Fig. 7 reports the speedups of all methods over two baselines: the baseline two-stage implementation and the baseline single-phase RAP implementation.

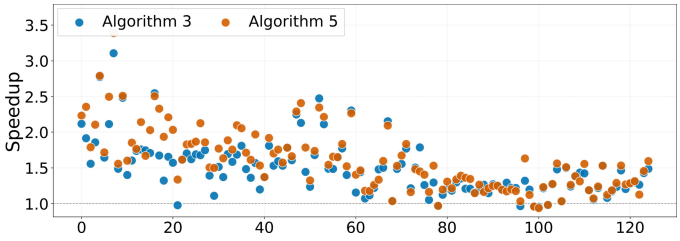
Overall, exploiting matrix symmetry together with multiply-transpose fusion yields substantial performance gains on most matrices. On the Intel platform, relative to the two-stage baseline, our optimizations achieve an average speedup of $1.60\times$ across the 15 matrices, with a maximum of $2.41\times$ on `s3dkt3m2`. Relative to the single-phase baseline, the average speedup is $1.50\times$, peaking at $2.33\times$ on `nasasrb`. On the ARM platform, the two-stage variant attains a mean speedup of $1.62\times$ and reaches $2.79\times$ on `crankseg_1`; for the single-phase variant, the mean speedup is $1.52\times$, with the best case of $2.25\times$ also on `crankseg_1`.



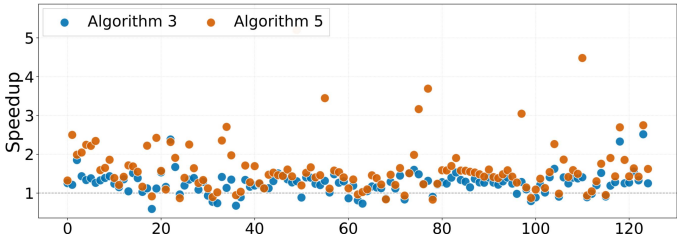
(a) Intel: Two-Stage



(b) Intel: Single-Phase



(c) ARM: Two-Stage



(d) ARM: Single-Phase

Fig. 5: Performance comparison on Intel and ARM platforms across 125 matrices. Fig. 5a and 5b report results on Intel, while Fig. 5c and 5d report results on ARM. Fig. 5a and 5c show speedups over the baseline two-stage method, whereas Fig. 5b and 5d show speedups over the baseline single-phase method. The x-axis denotes the matrix index and the y-axis denotes the speedup relative to the corresponding baseline.

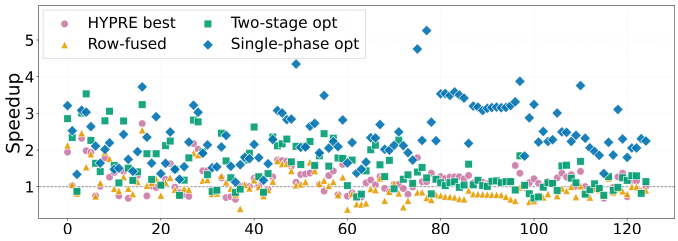


Fig. 6: Performance comparison with MKL, HYPRE and the row-fused method on the Intel platform using 125 matrices. The x-axis denotes the matrix index, and the y-axis reports the speedup over MKL.

However, for several matrices, multiply-transpose fusion provides limited improvement. This behavior is more pronounced for matrices with highly irregular sparsity patterns, where the fused metadata updates introduce substantial cache miss overheads during SpGEMM accumulation.

E. Memory-Access Overhead

We also measure memory-access overhead on the Intel platform using LIKWID [26] for the 15 representative matrices. Fig. 8 compares the memory overhead of the two-stage and single-phase RAP implementations against their corresponding baselines. Overall, our optimizations substantially reduce memory-access overhead across these matrices. For two-stage RAP, leveraging symmetry reduces the overhead to 69% of the baseline on average, and multiply-transpose fusion further lowers it to 67%. For single-phase RAP, leveraging symmetry reduces the overhead to 65% on average, and fusion further reduces it to 64%. In summary, our approach provides effective memory-access optimizations for both RAP formulations.

TABLE III: Input matrices used in our evaluation.

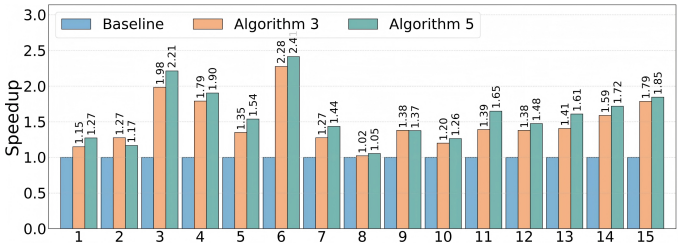
ID	Matrix	Rows(N)	#nnz	#nnz/N
1	crankseg_1	52,804	10,614,210	201.01
2	nasasrb	54,870	2,677,324	48.79
3	cant	62,451	4,007,383	64.17
4	t3dh_a	79,171	4,352,105	54.97
5	t3dh_e	79,171	4,352,105	54.97
6	s3dkt3m2	90,449	3,753,461	41.50
7	2cubes_sphere	101,492	1,647,264	16.23
8	filter3D	106,437	2,707,179	25.43
9	TEM152078	152,078	6,459,326	42.47
10	turon_m	189,924	1,690,876	8.90
11	bmw3_2	227,362	11,288,630	49.65
12	BenElechi1	245,874	13,150,496	53.48
13	Ga41As41H72	268,096	18,488,476	68.96
14	msdoor	415,863	20,240,935	48.67
15	af_shell10	1,508,065	52,672,325	34.93

V. CONCLUSION

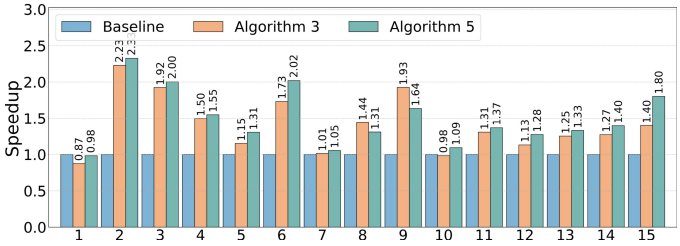
In this paper, we proposed a memory-efficient symmetry-exploiting RAP algorithm for algebraic multigrid. The method exploits matrix symmetry and the relation $R = P^T$ to reduce redundant computation and memory access, and further combines multiply-transpose fusion with nonzero-aware load-balanced transposition to lower memory traffic and improve parallel efficiency.

We implemented the proposed techniques in both two-stage and single-phase RAP formulations and evaluated them on Intel and ARM multicore platforms using 125 symmetric SuiteSparse matrices. Experimental results show consistent speedups and reduced memory-access overhead, achieving up to $1.62\times$ average speedup over the baseline implementations and $2.41\times$ over Intel MKL.

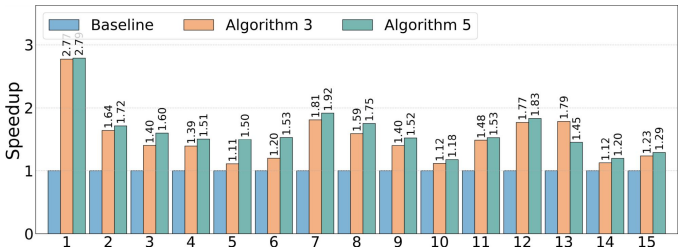
Future work will integrate the proposed symmetric RAP method into a complete symmetry-exploiting AMG frame-



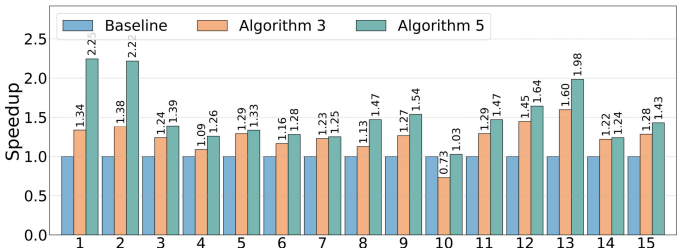
(a) Intel: Two-Stage



(b) Intel: Single-Phase

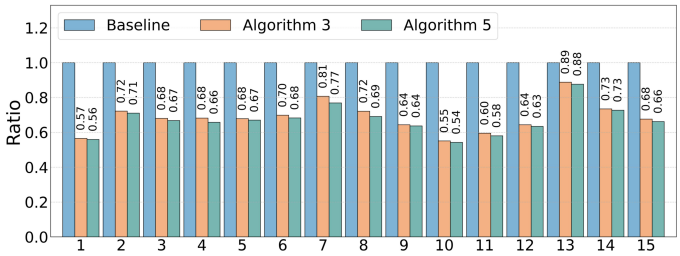


(c) ARM: Two-Stage

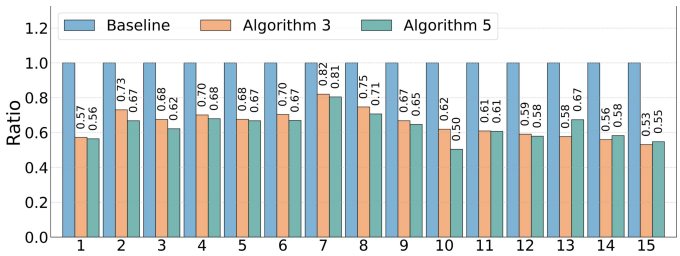


(d) ARM: Single-Phase

Fig. 7: Performance comparison on Intel and ARM platforms across 15 representative matrices. Fig. 7a and 7b report results on Intel, while Fig. 7c and 7d report results on ARM. Fig. 7a and 7c show speedups over the baseline two-stage method, whereas Fig. 7b and 7d show speedups over the baseline single-phase method. The x-axis denotes the matrix index and the y-axis denotes the speedup relative to the corresponding baseline.



(a) Intel: Two-Stage



(b) Intel: Single-Phase

Fig. 8: Memory consumption on the Intel platform for 15 representative matrices. Fig. 8a reports the memory-consumption ratio relative to the baseline two-stage method, whereas Fig. 8b reports the ratio relative to the baseline single-phase method. The x-axis denotes the matrix index, and the y-axis denotes the ratio with respect to the corresponding baseline.

work and systematically evaluate its impact on overall AMG performance, including the contribution of RAP to the setup time, per-stage timing breakdowns, and level-wise performance across the AMG hierarchy. We will also extend the method to distributed-memory and heterogeneous platforms.

VI. Acknowledgment

The authors would like to thank the referees for their valuable comments. This work is supported by the National Key R&D Program of China (2023YFB3001702), and NSF of China (No. 62502048, 62032023, 42104078, 61902411).

References

[1] M. Emans, "Efficient parallel AMG methods for approximate solutions of linear systems in CFD applications," *SIAM Journal on Scientific Computing*, vol. 32, no. 4, pp. 2235–2254, 2010.

[2] M. Brezina, C. Tong, and R. Becker, "Parallel algebraic multigrids for structural mechanics," *SIAM Journal on Scientific Computing*, vol. 27, no. 5, pp. 1534–1554, 2006.

[3] T. Mifune, T. Iwashita, and M. Shimasaki, "A fast solver for FEM analyses using the parallelized algebraic multigrid method," *IEEE transactions on magnetics*, vol. 38, no. 2, pp. 369–372, 2002.

[4] J. W. Ruge and K. Stüben, "Algebraic multigrid," in *Multigrid methods*. SIAM, 1987, pp. 73–130. [Online]. Available: https://doi.org/10.1137/1.9781611971057.ch4

[5] K. Stüben, "A review of algebraic multigrid," *Numerical Analysis: Historical Developments in the 20th Century*, pp. 331–359, 2001.

[6] A. Bienz, W. D. Gropp, and L. N. Olson, "Reducing communication in algebraic multigrid with multi-step node aware communication," *The International Journal of High Performance Computing Applications*, vol. 34, no. 5, pp. 547–561, 2020.

[7] P. W. Hemker, "A note on defect correction processes with an approximate inverse of deficient rank," *Journal of Computational and Applied Mathematics*, vol. 8, no. 2, pp. 137–139, 1982.

[8] N. Bell, S. Dalton, and L. N. Olson, "Exposing fine-grained parallelism in algebraic multigrid methods," *SIAM Journal on Scientific Computing*, vol. 34, no. 4, pp. C123–C152, 2012.

[9] W. Liu and B. Vinter, "A framework for general sparse matrix–matrix multiplication on GPUs and heterogeneous processors," *Journal of Parallel and Distributed Computing*, vol. 85, pp. 47–61, 2015.

[10] Y. Lu, L. Zeng, T. Wang, X. Fu, W. Li, H. Cheng, D. Yang, Z. Jin,

M. Casas, and W. Liu, "Amgt: Algebraic multigrid solver on tensor cores," in *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2024, pp. 1–16.
[11] J. Gao, W. Ji, F. Chang, S. Han, B. Wei, Z. Liu, and Y. Wang, "A systematic survey of general sparse matrix-matrix multiplication," *ACM Computing Surveys*, vol. 55, no. 12, pp. 1–36, 2023.
[12] A. Azad, G. Ballard, A. Buluc, J. Demmel, L. Grigori, O. Schwartz, S. Toledo, and S. Williams, "Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication," *SIAM Journal on Scientific Computing*, vol. 38, no. 6, pp. C624–C651, 2016.
[13] C. Hong, Q. Wang, R. Mao, Y. Liang, R. Xia, and J. Liu, "SaSpGEMM: Sorting-avoiding sparse general matrix-matrix multiplication on multi-core processors," in *Proceedings of the 53rd International Conference on Parallel Processing*, 2024, pp. 1166–1175.
[14] C. Hong, Q. Wang, X. Peng, G. Liu, Q. Zhang, X. Chen, and J. Liu, "VES: Vectorized sparse general matrix-matrix multiplication on multi-core DSPs," in *Proceedings of the 54th International Conference on Parallel Processing*, 2025, pp. 135–145.
[15] J. R. Gilbert, C. Moler, and R. Schreiber, "Sparse matrices in MATLAB: Design and implementation," *SIAM journal on matrix analysis and applications*, vol. 13, no. 1, pp. 333–356, 1992.
[16] R. D. Falgout and U. M. Yang, "hypre: A library of high performance preconditioners," in *International Conference on computational science*. Springer, 2002, pp. 632–641.
[17] J. Park, M. Smelyanskiy, U. M. Yang, D. Mudigere, and P. Dubey, "High-performance algebraic multigrid solver optimized for multi-core based distributed parallel systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015, pp. 1–12.
[18] H. Wang, W. Liu, K. Hou, and W. Feng, "Parallel transposition of sparse data structures," in *Proceedings of the 2016 international conference on supercomputing*, 2016, pp. 1–13.
[19] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, Dec. 2011. [Online]. Available: https://doi.org/10.1145/2049662.2049663
[20] U. M. Yang, "Parallel algebraic multigrid methods—high performance preconditioners," in *Numerical solution of partial differential equations on parallel computers*. Springer, 2006, pp. 209–236.
[21] R. D. Falgout and J. B. Schroder, "Non-Galerkin coarse grids for algebraic multigrid," *SIAM Journal on Scientific Computing*, vol. 36, no. 3, pp. C309–C334, 2014.
[22] F. G. Gustavson, "Two fast algorithms for sparse matrices: Multiplication and permuted transposition," *ACM Transactions on Mathematical Software (TOMS)*, vol. 4, no. 3, pp. 250–269, 1978.
[23] Y. Nagasaka, S. Matsuoka, A. Azad, and A. Buluç, "Performance optimization, modeling and analysis of sparse matrix-matrix products on multi-core and many-core processors," *Parallel Computing*, vol. 90, p. 102545, 2019.
[24] Z. Gu, J. Moreira, D. Edelsohn, and A. Azad, "Bandwidth optimized parallel algorithms for sparse matrix-matrix multiplication using propagation blocking," in *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, 2020, pp. 293–303.
[25] Phytium, "FT-2000+/64," https://www.phytium.com.cn/homepage/index/, accessed: May, 2026.
[26] J. Treibig, G. Hager, and G. Wellein, "Likwid: A lightweight performance-oriented tool suite for x86 multicore environments," in *2010 39th international conference on parallel processing workshops*. IEEE, 2010, pp. 207–216.