FastMPK: Unleashing Cache Reuse for Sparse Matrix Power Kernels on Many-Core Architectures

Haozhong Qiu*
College of Computer Science and Technology, National University of Defense Technology
Changsha, China
qiuhaozhong16@nudt.edu.cn

Chuanfu Xu*
College of Computer Science and Technology, National University of Defense Technology
Changsha, China
xuchuanfu@nudt.edu.cn

Jianbin Fang
College of Computer Science and Technology, National University of Defense Technology
Changsha, China
j.fang@nudt.edu.cn

Shengguo Li, Chunye Gong
College of Computer Science and Technology, National University of Defense Technology
Changsha, China
{nudtlsg, gongchunye}@nudt.edu.cn

Tiaojie Xiao, Xiang Gao
College of Computer Science and Technology, National University of Defense Technology
Changsha, China
{xiaotiaojie, gaoxiang12}@nudt.edu.cn

Liang Deng, Jian Zhang, Zhe Dai
China Aerodynamics Research and Development Center
Mianyang, Sichuan
dengliang11@nudt.edu.cn, zhangjian@cardc.cn
daizhe_cardc@163.com

Yue Ding, Yue Wang
College of Computer Science and Technology, National University of Defense Technology
Changsha, China
{dingyue, wangyue}@nudt.edu.cn

Zhimeng Han, Qingsong Wang, Wenjing Zhang
College of Computer Science and Technology National University of Defense Technology
Changsha, China
{hanzhimeng24, zhangwenjing24}@nudt.edu.cn
wwwangqs17@163.com

Yonggang Che, Jie Liu
College of Computer Science and Technology, National University of Defense Technology
Changsha, China
{ygche, liujie}@nudt.edu.cn

***Abstract*—Many applications rely on the sparse matrix power kernel (MPK), which evaluates a sequence of sparse matrix-vector multiplications (SpMVs) with the same matrix. MPK is therefore dominated by the memory traffic of repeatedly streaming the sparse matrix, and cache reuse across successive SpMVs offers a major opportunity to reduce data movement. However, exploiting such reuse introduces inter-SpMV dependencies that can severely restrict parallelism and induce frequent global synchronization, especially on GPUs.**

This paper presents *FastMPK*, a cache-aware MPK designed to run efficiently on both CPUs and GPUs. FastMPK models the matrix as a dependency graph, recursively partitions it into cache-resident subgraphs, and constructs a *separator* subgraph to decouple cross-subgraph dependencies and expose parallelism. A dependency-preserving runtime scheduling maximizes immediate cache reuse while a multi-granularity parallelization enables scalability on many-core processors. Evaluated on two x86 CPUs, one ARM CPU, and two NVIDIA GPUs, FastMPK delivers average speedups of up to 1.84× over state-of-the-art cache-reuse MPK implementations and up to 1.43× over vendor-tuned oneMKL/cuSPARSE libraries. Integrated into Ginkgo, FastMPK yields up to 1.60× average end-to-end speedup for polynomial preconditioned GMRES.

***Index Terms*—SpMV, matrix power kernel, cache reuse, cache blocking, GPUs**

This work was supported by the National Natural Science Foundation of China under Grant No. 62272474.

*: Haozhong Qiu and Chuanfu Xu are corresponding and co-first authors. They contribute equally to this work.

I. Introduction

Sparse Matrix–Vector Multiplication (SpMV) is a fundamental kernel that appears across application domains ranging from high-performance computing to deep learning [1]. In many of these workloads, performance is determined not by a single SpMV but by repeatedly applying the same sparse operator. This motivates the sparse Matrix Power Kernel (MPK), which computes a sequence of SpMVs such as $[\mathbf{Ax}, \mathbf{A}^2\mathbf{x}, \ldots, \mathbf{A}^s\mathbf{x}]$ for a fixed sparse matrix $\mathbf{A}$ and a power s [2]–[4]. MPK is a building block in Krylov subspace methods (KSMs), algebraic multigrid (AMG), and eigenvalue solvers [5], [6].

In KSMs for solving $\mathbf{Ax} = \mathbf{b}$, conventional implementations often perform a global reduction after each SpMV (e.g., for orthogonalization of the Krylov basis), making synchronization and communication a first-order cost at scale. Communication-avoiding variants such as s-step [7], [8] reduce this overhead by generating s basis vectors per iteration using MPK. The resulting tradeoff is clear: fewer global synchronizations and communications, but a more performance-critical MPK.

MPK offers a significant opportunity to exploit temporal locality. Since successive SpMVs reuse the same matrix, a naïve implementation that calls an SpMV kernel s times needlessly streams the sparse matrix from memory repeat-

SC26, November 15-20, 2026, Chicago, Illinois, USA
979-8-3195-4789-7/26/$31.00 ©2026 IEEE

edly, even though parts of it could be reused from cache. Exploiting this locality is challenging, however: cache reuse across successive SpMVs induces inter-SpMV dependencies. Enforcing these dependencies naively can serialize the computation, reduce available parallelism, and introduce frequent global synchronization—precisely the bottlenecks that limit scalability on many-core processors, especially GPUs.

Prior cache-reuse MPK approaches for general sparse matrices, such as RACE MPK [3] and FBMPK [2] demonstrate that reuse is possible on CPUs but typically rely on level/color-based execution to maintain dependencies, which restricts parallelism and requires many global synchronizations. These design choices make it difficult to efficiently exploit cache reuse on GPUs, where saturation demands abundant parallel work with minimal kernel-launch and synchronization overhead.

To address this gap, we introduce FastMPK, a cache-aware MPK that makes reuse practical on both CPUs and GPUs. FastMPK models the matrix as a dependency graph, recursively partitions it into cache-resident subgraphs, and constructs a *separator* subgraph to decouple cross-subgraph dependencies. Building on this decomposition, a runtime scheduler executes subgraphs in a dependency-preserving order that promotes immediate cache reuse while minimizing global synchronization. FastMPK then combines coarse-grained parallelism across subgraphs with fine-grained parallelism within each subgraph to achieve high utilization on modern many-core architectures.

We evaluate FastMPK on 110 large matrices from the SuiteSparse Matrix Collection [9] and on two x86 CPUs, one ARM CPU, and two NVIDIA GPUs (A100 and H100). FastMPK consistently outperforms state-of-the-art cache-reuse MPK implementations and vendor-tuned SpMV-based baselines, and it translates these kernel-level gains into end-to-end speedups when integrated into Ginkgo for polynomial preconditioned GMRES [6], [8], [10].

This paper makes the following main contributions:

- We present FastMPK, a high-performance sparse matrix power kernel designed to run efficiently on both CPUs and GPUs.
- We propose a cache-reuse strategy for MPK based on dependency-graph partitioning and runtime scheduling, which can reduce the matrix memory traffic of the naïve MPK by nearly half.
- We develop a dependency-aware parallelization scheme that combines coarse-grained and fine-grained parallelism to minimize synchronization overhead on many-core processors.
- We conduct a comprehensive evaluation across diverse architectures and a broad set of sparse matrices, demonstrating consistent speedups over state-of-the-art MPK implementations and vendor-tuned SpMV libraries.
- We demonstrate the practical impact of FastMPK by integrating it into a polynomial preconditioned GMRES solver, where it achieves significant end-to-end speedups with minimal preprocessing overhead.

Algorithm 1: Baseline SpMV and MPK

```
1 function SpMV(A, x, b, st, ed)
2     for i ← st to ed-1 do in parallel
3         sum ← 0
4         for j ← A.rowPtr[i] to A.rowPtr[i+1] do
5             sum ← sum + A.value[j] · x[A.colIdx[j]]
6         b[i] ← sum
7 function MPK(A, x, s, N)
8     for i ← 0 to s-1 do
9         SpMV(A, x[:,i], x[:,i+1], 0, N)
```

II. Background and Motivation

A. Baseline SpMV and MPK

The SpMV kernel computes the product of a sparse matrix $\mathbf{A}$ and a dense vector $\mathbf{x}$, yielding a dense output vector $\mathbf{b}$. We adopt the prevalent Compressed Sparse Row (CSR) format [11], [12] in our baseline SpMV kernel. As shown in Algorithm 1 (lines 1-6), the CSR format employs three arrays: `value` and `colIdx` store the nonzero elements and their column indices, respectively, while `rowPtr` records the start and end offsets of each row within the previous arrays. A key feature of this SpMV kernel is its support for operating on a contiguous row block (row indices in $[st, ed-1]$) of the matrix, a functionality we leverage to implement FastMPK. This differs from most SpMV kernels in existing libraries, which only operate on the entire matrix.

Algorithm 1 (lines 7-9) shows a naïve MPK implementation, which simply invokes the SpMV kernel s times. This approach overlooks the opportunity for explicit cache reuse; consequently, the matrix $\mathbf{A}$ may be loaded from main memory up to s times if it exceeds the cache capacity.

Figure 1(a) and (b) illustrate the cache reuse opportunity between two successive SpMVs. Suppose a submatrix (yellow rows) of $\mathbf{A}$ has been loaded into cache to calculate $\mathbf{Ax}$, it is possible to reuse a subset (red rows) of this submatrix to update part of $\mathbf{A}^2\mathbf{x}$. However, this cache reuse must fulfill the data dependencies between successive SpMV invocations. For example, to obtain $\mathbf{A}^2\mathbf{x}[21]$, the calculation of $\mathbf{Ax}[14]$, $\mathbf{Ax}[21]$, $\mathbf{Ax}[22]$, and $\mathbf{Ax}[28]$ must have been completed. Each nonzero element of a matrix row corresponds to a dependency among rows/columns. These dependencies significantly complicate efficient implementation of MPK.

B. Existing MPK implementations

RACE MPK [3]. This approach is based on the Recursive Algebraic Coloring Engine (RACE) originally designed for shared-memory symmetric SpMV [13]. It first derives an undirected graph for a sparse matrix, where each vertex represents a matrix row/column, and each edge represents a nonzero element. Then the graph is divided into many RACE levels, and SpMV computations can be scheduled for different levels and powers according to the level neighborhood to achieve cache reuse and maintain dependencies.

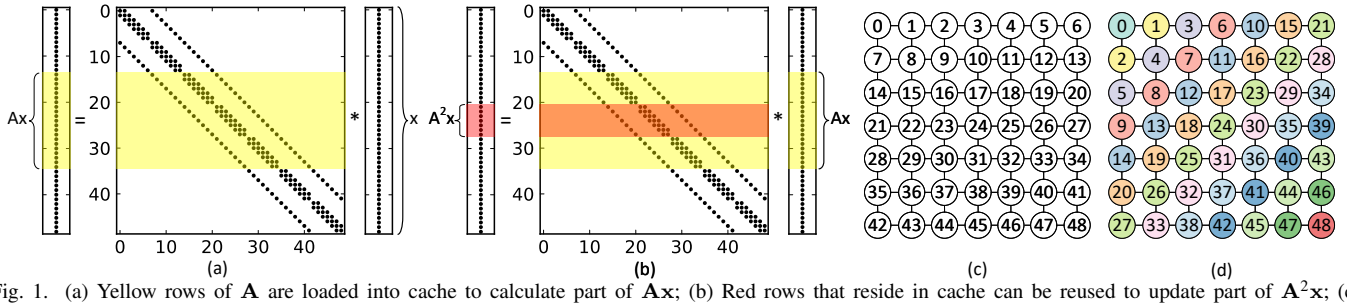


Fig. 1. (a) Yellow rows of $\mathbf{A}$ are loaded into cache to calculate part of $\mathbf{Ax}$; (b) Red rows that reside in cache can be reused to update part of $\mathbf{A}^2\mathbf{x}$; (c) Graph of $\mathbf{A}$; (d) 13 RACE levels (indicated by different colors) after reordering.

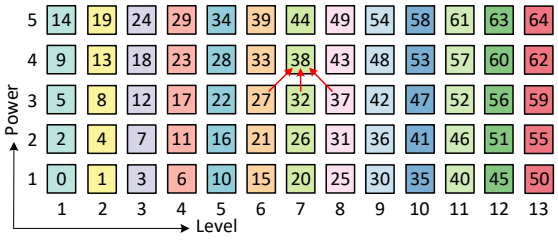


Fig. 2. Execution order of RACE MPK for Figure 1 (d) with power s=5. Boxes represent RACE levels (same colors as Figure 1 (d)). Numbers in boxes indicate the execution order. Red arrows indicate dependencies.

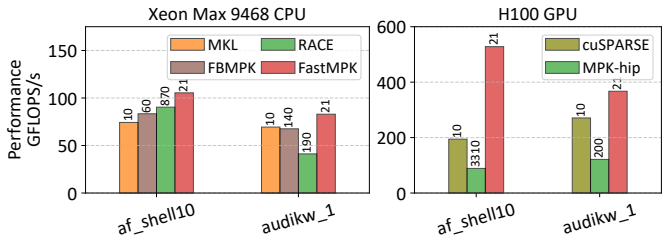


Fig. 3. Comparison of different MPK versions. Numbers on bars indicate the frequency of global synchronizations (or kernel launches on GPUs).

Figure 1(a), (c) and (d) show an example matrix $\mathbf{A}$, its graph, and the 13 RACE levels. Figure 2 shows the corresponding execution order in RACE MPK. As shown in Figure 2, a level can be reused after $s+1$ steps if it still resides in cache (i.e., the reuse distance is $s+1$); to advance all vertices of a level to $\mathbf{A}^{i}\mathbf{x}$, its neighboring levels must have finished the calculation of $\mathbf{A}^{i-1}\mathbf{x}$.

As parallelization can only be implemented within each RACE level, this approach typically exposes limited parallelism with frequent global synchronizations among levels, and heavily relies on level coarsening and point-to-point synchronization to improve scalability on CPUs. Its total number of synchronizations for $\mathbf{A}^{s}\mathbf{x}$ is $s \times numRL$, where $numRL$ is the number of coarse RACE levels. [14] evaluated a GPU version of RACE MPK called *MPK-hip*, and reported severe performance degradation compared to a naïve MPK, due to a large number of synchronizations (kernel launches).

FBMPK [2]. This approach first partitions a sparse matrix into the lower and upper triangular submatrices. Then it follows a forward-backward computation pipeline to process the two submatrices, and merges the calculation of $\mathbf{A}^{i}\mathbf{x}$ and $\mathbf{A}^{i+1}\mathbf{x}$ for each matrix row in the same iteration. Therefore, this approach ensures very fine-grained data reuse, with only one matrix row reused each time.

FBMPK leverages the algebraic block multi-color (ABMC) ordering [15] to identify independent rows for parallelization, thereby often leading to high synchronization cost, load imbalance, and poor spatial locality [16]. Its total number of synchronizations for $\mathbf{A}^{s}\mathbf{x}$ is $2s \times numC$, where $numC$ is the color number in ABMC. Currently, FBMPK does not provide a GPU version and only has been evaluated on CPUs.

C. Motivation

Despite their advantages, existing cache-reuse approaches for MPK often impose substantial parallelization overhead. As summarized in Table I, state-of-the-art techniques like FBMPK and RACE MPK require a large number of global synchronizations—proportional to both the power s and the number of colors or levels in the matrix partitioning ($numC$ or $numRL$). These values are typically much larger than 2, leading to significant synchronization costs, especially as s increases. Furthermore, these methods were primarily designed for multi-core CPUs and do not scale to many-core accelerators.

The performance implications of these limitations are evident in Figure 3, which compares different MPK implementations on two matrices with varying sparsity patterns (see Section IV-A). While FBMPK and RACE MPK improve performance for the well-structured matrix `af_shell10` on the CPU, they perform poorly on the more irregular matrix `audikw_1`, primarily due to excessive synchronization. Moreover, the performance of RACE MPK degrades severely on the GPU, making it slower than even the naïve standard MPK. FastMPK is the only method that delivers robust performance improvements across both CPU and GPU platforms for the two matrices.

In summary, while state-of-the-art MPK implementations leverage cache reuse, they do so at the cost of restricted parallelism and frequent synchronization. This hinders their efficiency on many-core processors like GPUs and degrades their performance on CPUs for irregular matrices. Thus, we propose FastMPK, which introduces a new cache blocking strategy that minimizes synchronization overhead while maintaining efficient cache reuse. FastMPK reduces the memory traffic of standard MPK by nearly half and requires no more

TABLE I
SUMMARY OF DIFFERENT MPK IMPLEMENTATIONS. BOTH *numC* AND *numRL* ARE TYPICALLY SIGNIFICANTLY LARGER THAN 2.

	numSync	CPU	GPU	Reuse	Distance	T
naïve MPK	s	yes	yes	no	/	1
FBMPK	$2s \times numC$	yes	no	L1	1	1/2
RACE MPK	$s \times numRL$	yes	no	L3	s+1	1/s
FastMPK	$\leq 2s$+1	yes	yes	L2	1	1/2

numSync: number of synchronizations; **Distance**: data reuse distance.
Reuse: main cache level of explicit data reuse.
T: relative memory traffic over a naïve MPK in theory.

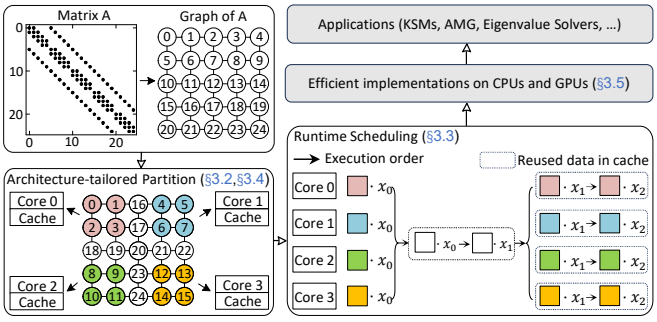


Fig. 4. The design overview of FastMPK

than $(2s + 1)$ global synchronizations for $\mathbf{A}^s\mathbf{x}$.

III. METHODOLOGY

A. Design overview

FastMPK is a library to efficiently calculate a sequence of SpMVs (MPK) by exploiting cache reuse on both CPUs and GPUs. As shown in Figure 4, FastMPK comprises several key components in its workflow. First, an architecture-tailored partitioning approach transforms an original matrix into cache-friendly independent submatrices and a separator according to core/SM counts and cache hierarchies of a CPU or GPU. Second, a novel runtime scheduling strategy executes SpMV operations on these submatrices/separator in a specified order to achieve cache reuse while fulfilling data dependencies among successive SpMV invocations. Third, efficient MPK kernels on CPUs and GPUs are implemented to exploit multi-granularity parallelization and minimize synchronizations in FastMPK. Finally, FastMPK can be seamlessly integrated into applications such as iterative solvers to improve the end-to-end performance. We provide detailed explanation in the following sections.

B. Architecture-tailored Matrix Partition

For a given matrix, we derive its undirected graph as described in Section II, and perform a *two-stage* architecture-tailored matrix partition. Each stage includes the following three steps:

Graph Decomposition. We first decompose the graph into *numPart* subgraphs (row blocks), where each subgraph contains no more than *partSize* nonzero elements. For a CSR-based sparse matrix with 8-byte double values and 4-byte int indices, *numPart* and *partSize* are determined as follows:

$$\begin{cases} partSize = C * \dfrac{cacheSize}{12} \\ numPart = \begin{cases} \lceil \dfrac{NNZ/partSize}{numCore} \rceil * numCore, & on\ CPU \\ \lceil NNZ/partSize \rceil, & on\ GPU \end{cases} \end{cases}$$

where NNZ is the total number of nonzero elements, *numCore* is the number of CPU cores, and C is a tuning parameter to account for other data (e.g. vectors) in cache. Note that *numPart* is a multiple of *numCore* on CPUs to ensure a balanced distribution of subgraphs among cores. *cacheSize* is a parameter that depends on a specific cache level of a processor to ensure that each subgraph can reside in this cache level for reuse, as discussed later.

This step can be implemented using existing graph partitioners like METIS [17]. However, we observe relatively high partitioning costs in METIS. Therefore, we develop a lightweight preprocessing tool for FastMPK. To obtain subgraphs, we perform breadth-first search (BFS) [18] to traverse the graph, and add some newly arrived vertices to current vertex subsets, if the total number of nonzero elements for each subset (including newly added vertices) is no more than *partSize*. Although METIS offers marginally better partition quality, our BFS approach significantly reduces preprocessing overhead with negligible impact on end-to-end MPK performance.

Separator formulation. Next, we iterate on each subgraph, and select some vertices that have neighbors in other subgraphs to formulate a special subgraph called *separator*. The separator disconnects the *numPart* subgraphs, similar to the concept commonly used in some Divide-and-Conquer (DC) algorithms [16], [19] such as the nested dissection [20]. In this paper, we always move the vertex with the most neighbors in its neighboring subgraph into the separator. This tends to minimize the number of vertices in the separator, which can benefit the performance of FastMPK.

Subgraph reordering. Finally, we reorder the vertex indices for all subgraphs including the separator to ensure that each subgraph contains vertices with consecutive indices, and the separator is indexed after the other subgraphs. This can improve the data locality when accessing the input/output vectors in SpMV.

After the first stage, we stop if the separator can also fit in cache (i.e. its number of nonzero elements is less than *partSize*); otherwise we take the separator as input and perform the above steps again as the second stage.

Discussions about cache hierarchy. Modern processors typically contain multi-level caches. Figure 5 (a) shows a simplified cache hierarchy for our tested CPUs and GPUs (see Table II). On CPUs, L3 cache is shared among all cores, while both L1 cache and L2 cache are core-private. Since the capacity of L1 cache is one to two orders of magnitude smaller than that of L2 cache, targeting L1 cache reuse will generate too many fine-grained subgraphs. Meanwhile, the

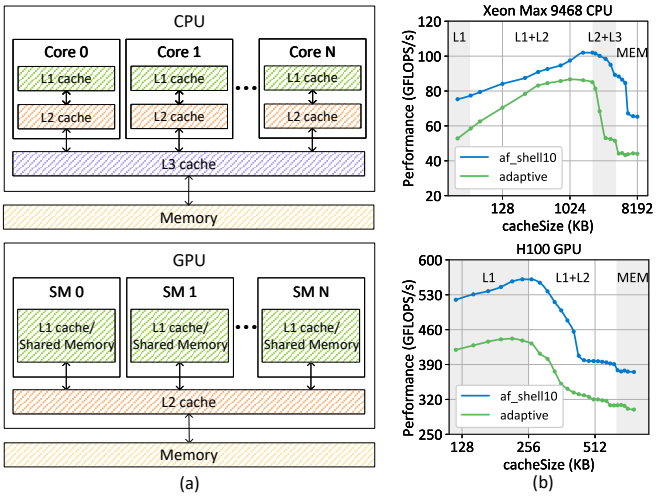


Fig. 5. (a) Simplified cache hierarchy for CPUs and GPUs. (b) Performance of FastMPK with varying $cacheSize$ (s=10).

aggregate capacity of L2 cache is roughly on par with that of L3 cache; however, L2 cache operates much faster than L3 cache. Therefore, we mainly target L2 cache reuse on CPUs, and *cacheSize* is set to the L2 cache size per core on CPUs.

On GPUs, L1 cache is shared by a GPU streaming multiprocessor (SM), while L2 cache is shared by all SMs. We prefer to reuse data in L1 cache on a GPU because the aggregate capacity of L1 cache is close to that of L2 cache. Unlike CPUs, we schedule multiple thread blocks to each SM to saturate a GPU. Therefore, *cacheSize* is set to $L1CacheSize/numActiveTB$, where *L1CacheSize* is the L1 cache size per SM, and *numActiveTB* is the number of active thread blocks per SM. Specifically, in this work *numActiveTB* is set to 2 (i.e. $cacheSize = L1CacheSize/2$), as our tested GPUs support a maximum of 2048 active threads per SM, and our thread block size is 1024.

Figure 5(b) shows that the performance of FastMPK changes significantly with varying *cacheSize* on the Xeon Max 9468 CPU and the H100 GPU. The result indicates that by customizing *cacheSize* FastMPK can successfully target data reuse at different cache levels. It also confirms the above discussion that targeting L2 cache reuse on CPUs and L1 cache reuse on GPUs can maximize the performance of FastMPK.

Example. Figure 6 shows an example of our two-stage partition for the matrix/graph in Figure 1 (*numPart=4*). We use a tree data structure to save the starting index for each subgraph. Figure 7(a) shows the tree data structure for Figure 6(e). This will be used in the following parallel implementation.

C. Runtime Scheduling

After matrix partition, we obtain many independent subgraphs and one separator. Each subgraph contains no more than *partSize* nonzero elements that can fit in cache. FastMPK schedules SpMV invocations by alternately advancing the odd and even powers on the subgraphs and the separator. To simply explain our scheduling strategy, consider a matrix $\mathbf{A}$ partitioned into two subgraphs ($\mathbf{S_1}$ and $\mathbf{S_2}$) and one separator $\mathbf{S_3}$ (i.e. $\mathbf{A} = \mathbf{S_1} + \mathbf{S_2} + \mathbf{S_3}$), the execution order on two CPU cores is as follows[1]:

$$(\mathbf{S_1}, \mathbf{S_2})\mathbf{x} \rightarrow |\mathbf{S_3}\mathbf{x} \rightarrow \overset{Ax}{|}\ \mathbf{S_3^2}\mathbf{x} \rightarrow |(\mathbf{S_1^2}, \mathbf{S_2^2})\mathbf{x} \rightarrow \overset{A^2x}{|}\ (\mathbf{S_1^3}, \mathbf{S_2^3})\mathbf{x} \rightarrow |\mathbf{S_3^3}\mathbf{x} \rightarrow \overset{A^3x}{|}\ \cdots \rightarrow \overset{A^{s-2}x}{|}\ \mathbf{S_3^{s-1}}\mathbf{x} \rightarrow |(\mathbf{S_1^{s-1}}, \mathbf{S_2^{s-1}})\mathbf{x} \rightarrow \overset{A^{s-1}x}{|}\ (\mathbf{S_1^s}, \mathbf{S_2^s})\mathbf{x} \rightarrow |\mathbf{S_3^s}\mathbf{x} \overset{A^sx}{|}$$

Note that $\mathbf{S_1}$ and $\mathbf{S_2}$ are assigned to different cores for concurrent execution of SpMVs, while $\mathbf{S_3}$ can be directly parallelized by vertices/rows. Meanwhile, subgraphs are immediately reused in cache between two neighboring powers (e.g., $\mathbf{S_1}$ and $\mathbf{S_2}$ are reused between $\mathbf{A}^2\mathbf{x}$ and $\mathbf{A}^3\mathbf{x}$).

This scheduling procedure can be applied to different numbers of subgraphs and CPU cores or GPU SMs. Figure 7(b) shows the scheduling diagram for Figure 6(c) on two CPU cores with a maximum power of 5. Each box represents a subgraph or separator calculated at different powers, with the numbers indicating the execution order. Each core is assigned two subgraphs and boxes with the same numbers are executed in parallel. The dotted rectangles indicate subgraphs that can be reused in cache at different powers.

Following the matrix partition and scheduling, we ensure that subgraphs can reside in cache and be reused with a reuse distance of 1. In Figure 7(b), we only load the subgraphs from the main memory 3 times to obtain $\mathbf{A}^5\mathbf{x}$, due to cache reuse. Suppose $\mathbf{Mem_{sub}}$ and $\mathbf{Mem_{sep}}$ represent the memory data traffic of the subgraphs and separator for each SpMV invocation, FastMPK consumes a total memory data traffic of $\lceil\frac{(s+1)}{2}\rceil * \mathbf{Mem_{sub}} + \lfloor\frac{(s+1)}{2}\rfloor * \mathbf{Mem_{sep}}$ to calculate $\mathbf{A}^s\mathbf{x}$, while a standard MPK implementation requires $s * (\mathbf{Mem_{sub}} + \mathbf{Mem_{sep}})$. Therefore, FastMPK tends to reduce the memory traffic of a standard MPK by a factor of $\frac{(s+1)}{2s}$ (i.e. roughly 1/2). We will present the measured data traffic of memory and cache for different MPK implementations in Section IV-B3. In addition, given a matrix and architecture we can define the reuse ratio R in FastMPK as $R = 1 - \frac{nnz_{sep}}{nnz_A}$, where nnz_{sep} and nnz_A are the numbers of nonzero elements for the separator and the whole matrix $\mathbf{A}$, respectively (see R in Table III for the representative matrices).

D. Further Optimizing Cache Reuse on CPUs

According to our scheduling strategy, subgraphs of the first stage (e.g., boxes marked with 0 and 1 in Figure 7(b)) cannot be reused immediately after calculating $\mathbf{Ax}$. This problem can be ameliorated by dividing these subgraphs into fine-grained parts to further improve cache reuse. Since each subgraph is formulated using the BFS algorithm (see Section III-B), we simply use the BFS levels to further divide a subgraph and implement a *level-based optimization* in FastMPK.

Figure 6(e) shows the result after dividing each subgraph of the first stage into three BFS levels (i.e., L_0, L_1 and L_2, indicated by different colors). Figure 7(c) shows the scheduling

[1]Suppose s is an odd integer and a vertical bar | represents a synchronization.

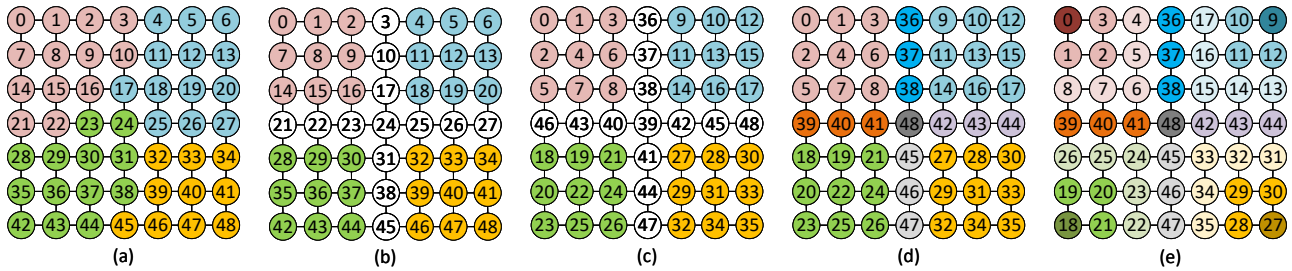


Fig. 6. (a) Partitioning a graph into four subgraphs (shown in different colors); (b) Formulating a separator (indicated with uncolored vertices); (c) After reordering vertices; (d) Second-stage partition for the separator; (e) Each first-stage subgraph is further divided into three levels indicated with different colors (see Section III-D).

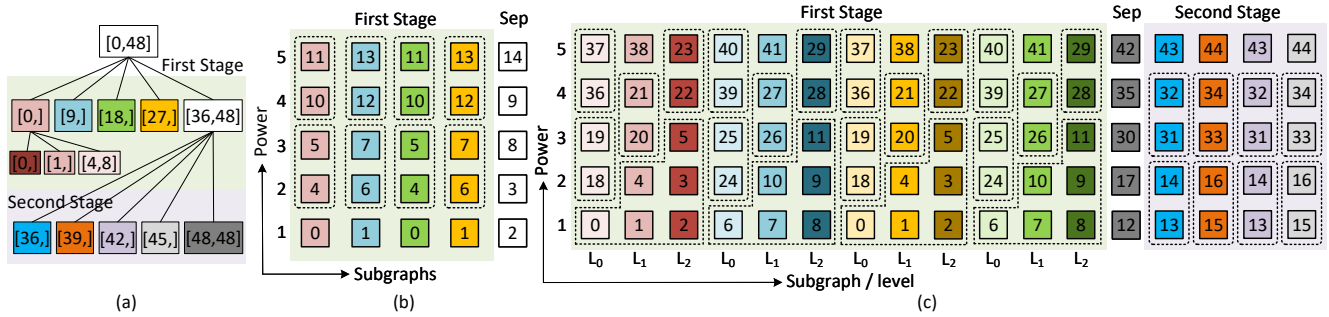


Fig. 7. (a) Tree data structure for Figure 6 (e); (b) Execution order of Figure 6(c) on 2 cores (s=5); (c) Execution order of Figure 6(e) on 2 cores (s=5). Numbers in boxes indicate the execution order. Subgraphs/levels with the same numbers are executed in parallel. Boxes in the same dotted rectangles indicate that subgraphs/levels are reused in cache.

diagram corresponding to Figure 6(e). Levels in each subgraph are scheduled on a CPU core in the following order to achieve cache reuse: $L_0x \rightarrow L_1x \rightarrow L_2x \rightarrow L_2^2x \rightarrow L_1^2x \rightarrow L_2^3x$. As we target L2 cache reuse on CPUs, all levels of a subgraph can reside in L2 cache; thus, L_2 is reused twice while L_1 is reused once. Furthermore, BFS levels are much smaller than subgraphs, which means that some of them may even fit in L1 cache. Therefore, we further improve cache reuse on CPUs for subgraphs at the first stage.

We note that the number of BFS levels ($numLevel$) should not be greater than the maximum power. A larger $numLevel$ will increase cache reuse, but harm the data locality when accessing the input vector in SpMVs. We will evaluate the impact of $numLevel$ in Section IV-B4. Additionally, we do not apply this optimization on GPUs, because it further limits parallelism in a thread block.

E. Implementation

FastMPK relies on a SpMV kernel that can operate on a subset of matrix rows and facilitate multi-granularity parallelization. Direct employment of a SpMV from vendor libraries may complicate the parallelization of FastMPK. Therefore, we implement our own SpMV for FastMPK.

SpMV implementation. On CPUs, we use the CSR-based SpMV in Algorithm 1, which operates on a row block of $[st, ed-1]$ for a sparse matrix. Parallelization is implemented at the row loop (line 2 in Algorithm 1) using OpenMP, and rows are statically assigned to cores to balance the nonzero elements.

On GPUs, we implement a CSR-based SpMV, which employs a sub-warp strategy similar to [21] and [22]. We dynamically assign 2^i threads within a warp[2] to calculate each row, where i depends on the average number of nonzero elements for all first-stage subgraphs or second-stage subgraphs.

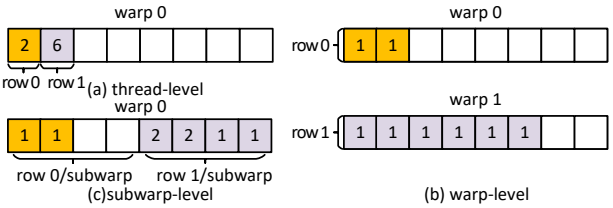


Fig. 8. Workload distribution for different SpMV implementations on GPUs (assume the warp size is 8). Each box represents a thread, with workload indicated by the number of assigned nonzero elements. For the subwarp-level strategy, we assume that two sub-warps, each containing $2^2 = 4$ threads, are scheduled to calculate the two rows.

Figure 8 illustrates different implementations on GPUs for two rows. Compared to the thread-level strategy, the sub-warp strategy avoids uncoalesced memory accesses and improves load balance. Compared to the warp-level strategy, the sub-warp strategy requires fewer GPU resources (only one warp instead of two). In addition, it can leverage the sub-warp shuffling operations in CUDA to efficiently accumulate the partial sum for each row. For the example in Figure 8, the sub-warp strategy performs two sub-warp reductions (6 additions in total), whereas the warp-level strategy requires two warp reductions (14 additions in total).

FastMPK implementation. Algorithm 2 presents a unified pseudo-code for parallel implementation of FastMPK on both CPUs and GPUs. Given the matrix $\mathbf{A}$, the maximum power s and the number of levels *numLevel*, we traverse subgraph-

[2] $i = 0, 1, ..., 5$ for a CUDA warp size of 32.

Algorithm 2: Pseudo-code of FastMPK

```
1  function FastMPK(A, x, s, numLevel, treeRoot)
       // numLevel: number of levels for first-stage subgraphs;
       // s: the maximum power;
       // treeRoot: the pointer for the root of the tree data structure
       // 1. Initial computation on first-stage subgraphs/levels
2      for j ← 0 to treeRoot->numPart-1 do in parallel
3          lev ← treeRoot->son[j]->lev
4          for i ← 0 to min(s, numLevel)-1 do
5              if i is odd then
6                  for k ← numLevel-1 to i do
7                      SpMV(A, x[:,i], x[:,i+1], lev[k], lev[k+1])
8              else
9                  for k ← i to numLevel-1 do
10                     SpMV(A, x[:,i], x[:,i+1], lev[k], lev[k+1])
       // Go to the separator at the first stage
11     sep1 ← treeRoot->sep
12     i ← 0
       // 2. Iteration for A^i x
13     while i + 1 < s do
           // 2.1: Run two SpMVs on second-stage subgraphs
14         if sep1->sep ≠ nullptr then
15             sep2 ← sep1->sep
16             SpMV(A, x[:,i], x[:,i+1], sep2->lev[0], sep2->lev[1])
17             for j ← 0 to sep1->numPart-1 do in parallel
18                 lev ← sep1->son[j]->lev
19                 SpMV(A, x[:,i], x[:,i+1], lev[0], lev[1])
20                 SpMV(A, x[:,i+1], x[:,i+2], lev[0], lev[1])
21             SpMV(A, x[:,i+1], x[:,i+2], sep2->lev[0],
                  sep2->lev[1])
22         else
23             SpMV(A, x[:,i], x[:,i+1], sep1->lev[0], sep1->lev[1])
24             SpMV(A, x[:,i+1], x[:,i+2], sep1->lev[0],
                  sep1->lev[1])
           // 2.2: Run two SpMVs on first-stage subgraphs/levels
25         for j ← 0 to treeRoot->numPart-1 do in parallel
26             lev ← treeRoot->son[j]->lev
27             for k ← 0 to numLevel-1 do
28                 SpMV(A,x[:,i+k+1],x[:,i+k+2],lev[k],lev[k+1])
29                 if i + k + 2 == s then break
30                 SpMV(A,x[:,i+k+2],x[:,i+k+3],lev[k],lev[k+1])
31         i ← i + 2
32     if s is odd then
33         SpMV(A, x[:,i], x[:,i+1], sep1->lev[0], sep1->lev[1])
```

s/levels from the root of the tree data structure (*treeRoot*) and execute SpMV operations according to our runtime scheduling strategy. We first perform initial computation to obtain part of $\mathbf{A}x$ to $\mathbf{A}^{\text{numLevel}}x$ on the first-stage subgraphs/levels (lines 2-10). Then we iterate for the remaining $\mathbf{A}^{\text{i}}x$ from i to s (lines 13-31). If the matrix contains second-stage subgraphs/levels (lines 14-21), we first execute a SpMV on the separator at the second stage (lines 15-16), and then execute two successive SpMVs on each second-stage subgraph/level (lines 17-20), which is followed by a SpMV on the second-stage separator (line 21). Otherwise, we execute two successive SpMVs on the separator at the first stage (lines 22-24). Next, we proceed to the first-stage subgraphs/levels and execute two successive SpMVs (lines 25-30). For each `while` iteration, we complete two powers for $\mathbf{A}^{\text{i}}x$, and i is advanced by 2 (line 31) for the next iteration. If s is odd, we require a final step (lines 32-33) to handle the separator at the first stage for the last power.

TABLE II
HARDWARE CONFIGURATIONS.

Metric	CPU			GPU	
	Xeon 9468	EPYC 7742	KP 920	A100	H100
Cores / SMs	48	64	32	108	132
L1 (KB)	48	32	64	192	256
L2 (MB)	2	0.5	0.5	40	50
L3 (MB)	105	256	32	N/A	N/A
Memory (GB)	256	128	128	40	80

On CPUs, FastMPK either concurrently executes SpMV operations on independent subgraphs/levels to exploit coarse-grained parallelism (lines 2-10, 17-20 and 25-30 in Algorithm 2), or calls SpMV on the separator to exploit fine-grained row-based parallelism (e.g., line 16). We disable nested OpenMP parallelism to avoid oversubscription of CPU cores. On GPUs, lines 2-10, lines 17-20 and lines 25-30 are encapsulated in three global CUDA kernels, where each subgraph is calculated by a thread block.

We can derive the total number of global synchronizations (or kernel launches on GPUs) *numSync* in FastMPK according to Algorithm 2. Suppose we perform a two-stage partition for $\mathbf{A}$ and s is odd, both the initial computation (lines 2-10) and the final step (lines 32-33) require one synchronization. In each `while` iteration, we require four synchronizations (line 16, lines 17-20, line 21 and lines 25-30). Therefore, *numSync* is $2 + 4\frac{s-1}{2} = 2s$. For a one-stage partition, each `while` iteration only requires two synchronizations (lines 22-24 and lines 25-30), and *numSync* is $1 + 2\frac{s-1}{2} + 1 = s + 1$ for odd s. We summarize *numSync* for a matrix with a one-stage or two-stage partition as follows: two-stage $\begin{cases} 2s, & s \text{ is odd} \\ 2s+1, & s \text{ is even} \end{cases}$ or one-stage $s + 1$ Compared to RACE and FBMPK, FastMPK significantly reduces the total number of synchronizations (see Table I).

IV. Performance Evaluation

A. Experiment Setup

Hardware and software. We perform experiments on two x86 CPUs (Intel Xeon Max 9468@2.1GHz and AMD EPYC 7742@2.2GHz), one ARMv8 CPU (Kunpeng 920@2.6GHz [23]) and two NVIDIA GPUs (A100 PCIe 40GB and H100 SXM5 80GB). Table II shows the detailed hardware configurations. For CPUs, L1 and L2 denote the L1 data cache size per core and the L2 cache size per core, respectively. For GPUs, L1 and L2 denote the L1 cache size per SM and the L2 cache size of the whole GPU, respectively. Intel Xeon Max 9468 has 64 GB of high-bandwidth memory (HBM), and was configured in HBM-only mode [24], [25]. On GPUs, the L1 cache and shared memory use the same hardware resources. We call `cudaFuncSetCacheConfig()` to prefer a larger L1 cache for FastMPK. On CPUs, we set thread affinity by binding each software thread to an individual physical core (without hyperthreading). All codes are compiled in double precision with the '-O3' optimization using Intel

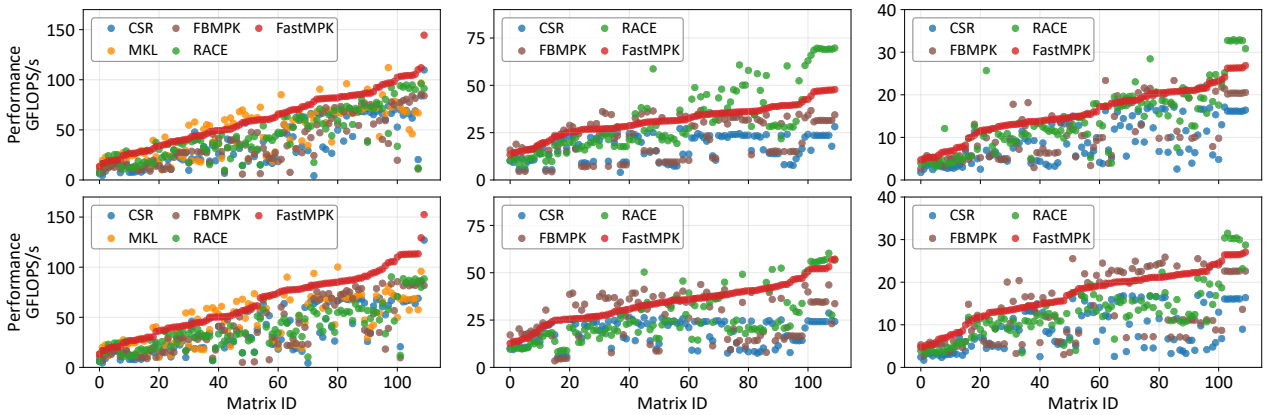


Fig. 9. Performance of different MPK implementations for 110 sparse matrices on CPUs (from left to right: Xeon Max 9468, EPYC 7742 and Kunpeng 920) with the maximum power s=5 (top) and s=10 (bottom).

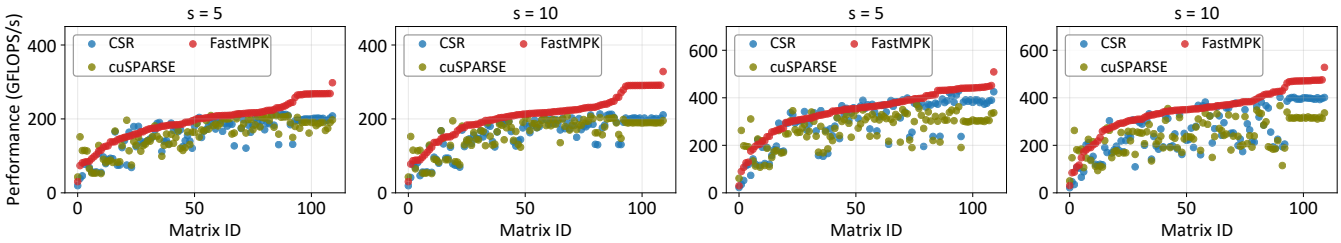


Fig. 10. Performance of different MPK implementations for 110 sparse matrices on the A100 GPU (left) and H100 GPU (right).

oneAPI 2025.1 on the Intel CPUs, GCC 11.2.1 on the AMD and HUAWEI CPUs, CUDA 12.1 on H100, and CUDA 11.7 on A100, respectively. We run each test 100 times and measure the average GFLOPS/s.

Baselines. We compare FastMPK to the following baselines: ❶ **CSR**, an MPK that directly calls our CSR-based SpMV. ❷ **MKL**, an MPK that directly calls the CSR-based SpMV in the Intel oneMKL library. ❸ **FBMPK**, an MPK implemented in [2]. ❹ **RACE** [3], an MPK based on the Recursive Algebraic Coloring Engine (RACE) approach [13]. ❺ **cuSPARSE**, an MPK that directly calls the CSR-based SpMV in the cuSPARSE library. Note that for RACE and FBMPK, we take the best measured performance by tuning their parameters on a given matrix and architecture according to their papers [2], [3].

Sparse matrices. We select 110 large matrices from the SuiteSparse Matrix Collection [9] for our tests. Each matrix contains more than 8,000,000 nonzero elements to ensure that its storage exceeds the cache capacity of our processors. Furthermore, we choose 7 representative matrices (see Table III) with different irregularities for a detailed analysis.

TABLE III
THE 7 REPRESENTATIVE MATRICES AND THEIR PROPERTIES. nnz_{std} IS THE STANDARD DEVIATION OF THE NONZERO ELEMENT NUMBER PER ROW. R_{cpu} AND R_{gpu} ARE THE REUSE RATIO OF NON-ZEROS ON THE XEON 9468 CPU AND THE H100 GPU RESPECTIVELY.

ID	Matrix	rows	nnz	nnz_{std}	R_{cpu}(%)	R_{gpu}(%)
1	af_shell10[1,2]	1,508,065	52,672,325	1.3	93.4	97.8
2	audikw_1[1,2]	943,695	77,651,847	42.4	84.6	39.3
3	Geo_1438[2]	1,437,960	63,156,690	40.8	95.8	72.9
4	gsm_106857[2]	589,445	21,758,924	15.7	100.0	80.4
5	Serena[1,2]	1,391,349	64,531,701	9.3	94.1	70.7
6	Long_Coup_dt0	1,470,152	87,088,992	6.7	93.0	64.4
7	SiO2	155,331	11,283,503	294.2	18.8	11.4

[1] Matrices evaluated by FBMPK [2].
[2] Matrices evaluated by RACE MPK [3].

B. Performance Analysis

1) Overall Performance: Figure 9 and 10 present the performance for different MPK versions with the maximum power s=5 and 10 on CPUs and GPUs. Table IV summarizes the speedup distributions of FastMPK over various baselines.

FastMPK achieves notable speedups over all baselines for most sparse matrices on both CPUs and GPUs. Compared to CSR, FastMPK improves performance for more than 99.1% and 90.9% of the matrices on CPUs and GPUs, with a geometric mean speedup of about 1.88× and 1.98× (Xeon 9468), 1.98× and 2.11× (EPYC 7742), 1.94× and 2.07× (Kunpeng 920), 1.26× and 1.35× (A100), 1.20× and 1.40× (H100), for s=5 and s=10. Given that CSR and FastMPK share the same SpMV implementation, this performance improvement is mainly due to effective cache reuse across multiple SpMV invocations in FastMPK.

Compared to MKL and cuSPARSE, FastMPK enhances the performance by 1.30× and 1.38× (Xeon 9468), 1.24× and 1.29× (A100), 1.26× and 1.43× (H100), for s=5 and s=10 using the geometric mean. We note that these two baselines employ highly vendor-tuned SpMVs, but they ignore explicit cache reuse. In contrast, FastMPK is based on a relatively simple SpMV implementation, so the speedups come primarily from cache reuse among multiple SpMV invocations. We

TABLE IV
SPEEDUP DISTRIBUTIONS OF FASTMPK OVER VARIOUS BASELINES FOR TWO POWERS ($s = 5$ AND $s = 10$). BLUE VALUES INDICATE THE LARGER GEOMETRIC MEAN SPEEDUPS OF THE TWO POWERS.

s	Speedup	Xeon Max 9468				EPYC 7742			Kunpeng 920			A100		H100	
		CSR	MKL	FBMPK	RACE	CSR	FBMPK	RACE	CSR	FBMPK	RACE	CSR	cuSPARSE	CSR	cuSPARSE
$= 5$	< 1	0.91%	26.4%	0.00%	4.55%	0.00%	17.3%	33.6%	0.91%	15.5%	20.0%	7.28%	9.09%	9.09%	8.18%
	1-1.5	39.1%	47.3%	41.8%	61.8%	36.4%	42.7%	50.9%	30.9%	46.4%	61.8%	76.4%	72.7%	71.8%	71.8%
	1.5-2	22.7%	8.18%	26.5%	20.0%	26.4%	8.18%	13.6%	24.5%	10.9%	13.6%	13.6%	16.4%	12.7%	16.4%
	≥ 2	37.3%	18.2%	31.8%	13.6%	37.3%	31.8%	1.82%	43.6%	27.3%	4.55%	2.73%	1.82%	6.36%	3.63%
	Mean	1.88×	1.30×	1.84×	1.43×	1.98×	1.60×	1.16×	1.94×	1.46×	1.20×	1.26×	1.24×	1.20×	1.26×
	Max	18.6×	3.44×	9.37×	9.21×	4.34×	3.53×	2.21×	8.08×	5.73×	4.55×	2.26×	2.21×	2.58×	2.31×
$= 10$	< 1	0.91%	22.7%	0.00%	3.64%	0.00%	20.0%	16.4%	0.91%	20.9%	9.09%	1.81%	7.27%	4.55%	8.18%
	1-1.5	30.9%	42.7%	51.8%	31.8%	37.3%	47.3%	36.4%	30.9%	41.8%	55.5%	77.3%	59.1%	59.1%	47.3%
	1.5-2	29.1%	16.4%	15.5%	37.3%	22.7%	4.55%	31.8%	25.5%	10.0%	27.3%	16.4%	30.0%	25.5%	31.8%
	≥ 2	39.1%	18.2%	32.7%	27.3%	40.0%	28.2%	15.5%	42.7%	27.3%	8.18%	4.54%	3.64%	11.0%	12.7%
	Mean	1.98×	1.38×	1.80×	1.71×	2.11×	1.59×	1.42×	2.07×	1.42×	1.41×	1.35×	1.29×	1.40×	1.43×
	Max	18.8×	4.17×	11.7×	9.37×	4.55×	3.91×	3.04×	7.77×	5.57×	4.45×	2.49×	2.37×	2.81×	3.74×

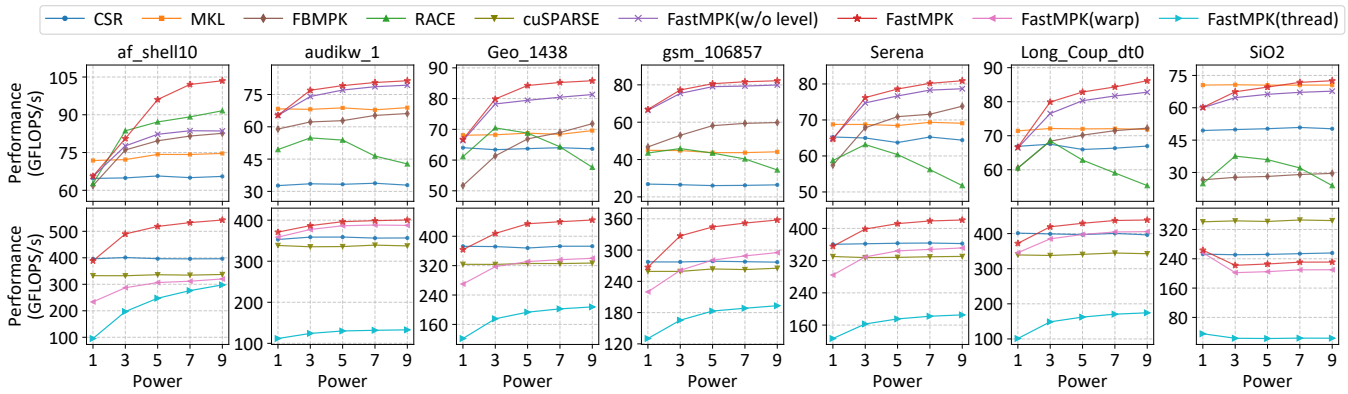


Fig. 11. Performance for different MPK versions with the power $s \in [1, 9]$ on the Xeon 9468 CPU (top) and the H100 GPU (bottom). FastMPK (w/o level) denotes the version that disables the level-based optimization in Section III-D.

expect further performance enhancement if a vendor SpMV is integrated into FastMPK, because our approach is complementary to vendor-specific optimizations that target a single SpMV invocation.

FastMPK yields significant speedups over the two state-of-the-art approaches. Compared to FBMPK, it achieves a geometric mean speedup of 1.84× and 1.80× (Xeon 9468), 1.60× and 1.59× (EPYC 7742), 1.46× and 1.42× (Kunpeng 920), for s=5 and s=10. Compared to RACE, the corresponding speedups are 1.43× and 1.71× (Xeon 9468), 1.16× and 1.42× (EPYC 7742), 1.20× and 1.41× (Kunpeng 920) using the geometric mean.

FastMPK generally achieves higher geometric mean speedups with larger s on both CPUs and GPUs (see blue values in Table IV). RACE shows a slightly higher performance than FBMPK. Since RACE mainly targets L3 cache reuse with a reuse distance of $s + 1$, it performs better on CPUs featuring large L3 cache (e.g., AMD EPYC 7742 in our tests) [3]. However, its performance may drop significantly with larger powers, as analyzed in the next section. In addition, both FBMPK and RACE fail to run efficiently on GPUs.

2) Analysis on Representative Matrices: Figure 11 compares different MPK versions with s ranging from 1 to 9 on Xeon 9468 and H100 for matrices in Table III. FastMPK achieves the maximum performance among all versions for almost all matrices and powers. Compared to FastMPK (w/o level), which disables the *level-based optimization* in Section III-D, FastMPK improves performance by 1.07× using the geometric mean, due to increased cache reuse on the Xeon 9468 CPU. Compared to the warp-level and thread-level strategies, our subwarp-level implementation significantly improves the performance of FastMPK on the H100 GPU, with a geometric mean speedup of 1.22× and 2.77×, respectively.

The performance of FastMPK and FBMPK gradually increases with larger powers. In contrast, RACE shows advantages for small powers (often ≤ 3 or 5), and its performance degrades severely for larger powers, confirming the report in [3]. This is mainly because RACE has a reuse distance of $s+1$, and it may swap matrix data out of cache for a large s, thereby failing cache reuse.

We typically observe poor performance for FBMPK and RACE on *irregular* matrices (e.g., *SiO2*, refer to nnz_{std} in Table III), because the two approaches often incur more synchronizations on these matrices. For example, on Xeon 9468 *SiO2* produces 15 colors in FBMPK and 15 coarse levels in RACE, which corresponds to $30 \times s$ and $15 \times s$ synchronizations to calculate $\mathbf{A}^s\mathbf{x}$ in FBMPK and RACE, respectively. In contrast, FastMPK performs no more than $2s + 1$ synchronizations to calculate $\mathbf{A}^s\mathbf{x}$ for an arbitrary matrix. The performance of FastMPK also deteriorates on

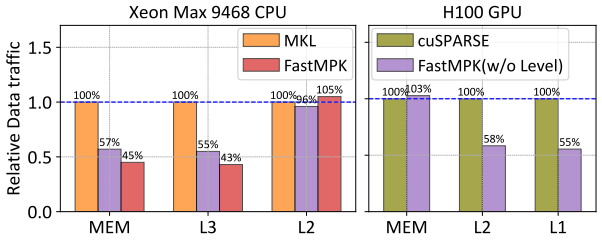


Fig. 12. Relative data traffic for memory and cache in different MPK versions ($s = 10$) using the geometric mean across the 7 representative matrices.

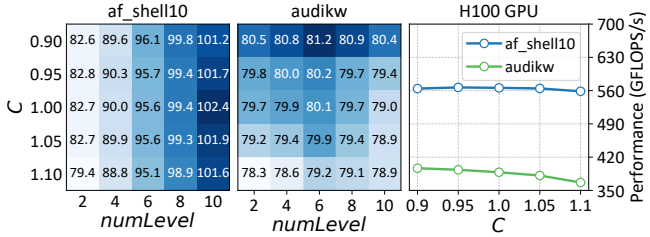


Fig. 13. Performance of FastMPK with different parameter values on Xeon 9468 (left two heat maps) and H100 (s=10).

SiO2, due to low reuse ratio of nonzero elements (only 18.8% for CPU and 11.4% for GPU, see Table III). However, FastMPK shows significantly better performance than FBMPK and RACE on Xeon 9468 for *SiO2*. The result demonstrates that FastMPK offers a superior trade-off between the benefits and overheads of data reuse.

3) Data Traffic for Memory and Cache: We use the LIKWID tool [26] on CPUs and Nsight Compute [27] on GPUs to measure data traffic of memory and cache for different MPK versions. Figure 12 shows the relative data traffic (using the geometric mean across the 7 representative matrices) for FastMPK, compared to MKL and cuSPARSE.

On Intel Xeon 9468, FastMPK (w/o level) reduces the data traffic of L3 cache and the main memory to about 57% and 55%, while FastMPK further reduces the data traffic to about 43% and 45% with our level-based optimization. On H100, FastMPK reduces the data traffic of memory and L2 cache to about 58% and 55%. The measured result is basically in line with the analysis in Section III-C that FastMPK can roughly halve the memory traffic of a standard MPK. We note that efficient cache reuse in MPK depends not only on the reduction of memory traffic but also the extra overheads (e.g., synchronization costs) in parallelization.

4) Parameter Impacts: We introduce two parameters in FastMPK: C accounts for other data in cache (e.g. vectors) and $numLevel$ indicates the number of levels for first-stage subgraphs (see Section III-D). We suggest that C is chosen from [0.90, 1.10] and $numLevel$ is chosen from [2, s]. Figure 13 shows the parameter impacts for two matrices with $C \in \{0.90, 0.95, 1.00, 1.05, 1.10\}$ and $numLevel \in \{2, 4, 6, 8, 10\}$. We typically achieve optimal performance with C= 0.90 or 0.95, and $numLevel$=s.

C. Case Study: the Preconditioned GMRES Solver

Previous cache-reuse MPK implementations have been shown to accelerate end-to-end applications such as sparse iterative solvers [28]. In this section, we showcase that FastMPK can be used directly to optimize the Generalized Minimal Residual (GMRES) solver—one of the most popular Krylov subspace methods for solving non-symmetric linear systems [6]. A preconditioner is often employed to accelerate the convergence of a Krylov subspace solver such as GMRES. In this work, we employ a polynomial preconditioner with a Leja-ordered Newton basis formulation [8], [29], which can enhance numerical stability.

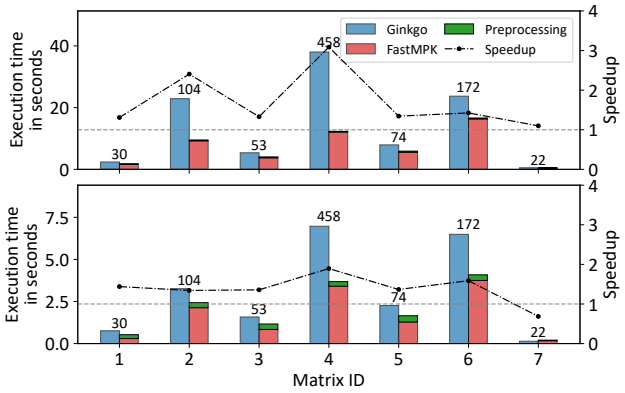


Fig. 14. End-to-end performance for the baseline and FastMPK-optimized versions of GMRES on Xeon 9468 (top) and H100 (bottom). Both versions converge with the same numbers of iterations as shown on bars.

Our implementations are developed on top of the high-performance numerical library Ginkgo [10]. Ginkgo includes a parallel GMRES for CPUs and GPUs, with an interface where users can provide a preconditioner. For a linear system $\mathbf{Ax} = \mathbf{b}$, our polynomial preconditioner with a degree of d requires the calculation of $\mathbf{A^d b}$. Our baseline version consecutively calls the CSR-based SpMV kernel from Ginkgo for d times in the polynomial preconditioner. This procedure is replaced with FastMPK for cache reuse in our optimized version.

Figure 14 compares the execution times for the two versions on Xeon 9468 and H100. We solve $\mathbf{Ax} = \mathbf{b}$ for the representative matrices and set $d = 40$ to ensure solver robustness. The GMRES restart interval is set to 100, and we stop when we reach a relative residual below 10^{-6} or a maximum of 1,000 iterations. On the CPU, FastMPK achieves an end-to-end speedup between 1.10× and 3.09× (with a geometric mean of 1.60×). On the GPU, the geometric mean speedup is 1.33×. Both versions converge with the same numbers of iterations, as FastMPK is an *implementation* optimization, without changing the numerical behavior of the solver. Figure 14 also includes the preprocessing time for each matrix[3], which is a one-time cost before entering the GMRES iterations. The preprocessing cost accounts for less than 10% of the total solver time across all matrices, equivalent to at most 10 MPK iterations. The case study shows that FastMPK can be easily integrated to accelerate existing applications.

[3]All matrix preprocessing is executed on the Xeon 9468 CPU.

V. Related Work

Optimizing a single SpMV. Sparse matrix–vector multiplication is one of the most extensively studied kernels in HPC. Prior work improves single-invocation performance mainly by reducing irregular memory traffic via storage formats, blocking, reordering, and architecture-aware kernels [30]–[33]. Because each nonzero is consumed only once per SpMV, these techniques primarily target reuse and locality of the input/output vectors (e.g., through register and cache blocking). Such single-SpMV optimizations are largely complementary to FastMPK and can be applied on top of our MPK scheduling.

MPK and communication avoidance. MPK has long been studied in the context of communication-avoiding and s-step Krylov subspace methods, where reusing the same matrix across multiple SpMVs reduces distributed communication and synchronization frequency [29], [34]–[36]. On shared-memory machines, attempts to exploit temporal locality across successive SpMVs often draw inspiration from temporal blocking in stencil computations [37]–[40]. For general sparse matrices, however, directly applying stencil-style blocking can introduce redundant work and/or irregular accesses to matrix elements, which limits performance [3].

Cache reuse for general matrices on CPUs. FBMPK [2] and RACE MPK [3] are two representative state-of-the-art approaches that enable cache reuse for MPK on unstructured matrices. Both maintain correctness by enforcing inter-SpMV dependencies through structured execution (colors or levels), but this comes at the cost of reduced available parallelism and frequent global synchronization. As a result, these methods can be sensitive to irregular sparsity patterns on CPUs and generally do not scale to many-core accelerators, where high parallelism is required to saturate the device.

MPK on GPUs. Existing GPU-oriented MPK work has largely targeted communication or data-movement issues across devices rather than cache reuse within a GPU. MehriDehnavi *et al.* [41] propose a cache-blocked MPK to avoid repeatedly transferring a matrix between CPUs and GPUs, focusing on cases where the matrix does not fit in GPU memory. Yamazaki *et al.* [42] study MPK to reduce communication among multiple GPUs. Panzlaff [14] evaluates a GPU adaptation of RACE MPK and reports significant slowdowns on modern GPUs due to the large number of kernel launches implied by level-by-level execution.

In contrast, FastMPK is designed to make cache reuse practical for MPK on *both* CPUs and GPUs by preserving locality while explicitly recovering parallelism and minimizing global synchronization.

VI. Conclusion

We presented FastMPK, a high-performance sparse matrix power kernel (MPK) that makes cache reuse effective across successive SpMV invocations on both CPUs and GPUs. FastMPK combines a cache-aware matrix decomposition with a dependency-preserving schedule that recovers parallel work and minimizes global synchronization. Across diverse many-core architectures, FastMPK consistently outperforms state-of-the-art MPK implementations and vendor-tuned SpMV-based baselines.

Looking forward, we plan to integrate FastMPK with distributed-memory MPK frameworks to enable hybrid parallelization at scale. Beyond MPK, the same decomposition and scheduling principles may benefit other iterative sparse workloads that repeatedly apply a fixed sparse operator, such as sequences of SpMM. Finally, we will further evaluate FastMPK inside end-to-end solvers (e.g., communication-avoiding Krylov methods), where reducing synchronization and memory bandwidth are critical for performance.

References

[1] T. Hoefler, D. Alistarh, T. Ben-Nun, N. Dryden, and A. Peste, "Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks," *Journal of Machine Learning Research*, vol. 22, no. 241, pp. 1–124, 2021. [Online]. Available: https://doi.org/10.48550/arXiv.2102.00554

[2] Y. Zhang, S. Li, F. Yuan, D. Dong, X. Yang, T. Li, and Z. Wang, "Memory-aware optimization for sequences of sparse matrix-vector multiplications," in *International Parallel and Distributed Processing Symposium*. Piscataway: IEEE, 2023, pp. 379–389. [Online]. Available: https://doi.org/10.1109/IPDPS54959.2023.00046

[3] C. Alappat, G. Hager, O. Schenk, and G. Wellein, "Level-based blocking for sparse matrices: Sparse matrix-power-vector multiplication," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 2, pp. 581–597, 2022. [Online]. Available: https://doi.org/10.1109/TPDS.2022.3223512

[4] J. Gurhem, M. Vandromme, M. Tsuji, S. G. Petiton, and M. Sato, "Sequences of sparse matrix-vector multiplication on fugaku's a64fx processors," in *International Conference on Cluster Computing*. Piscataway: IEEE, 2021, pp. 751–758. [Online]. Available: https://doi.org/10.1109/Cluster48925.2021.00111

[5] Y. Saad, *Numerical Methods for Large Eigenvalue Problems: Revised Edition*. Philly: SIAM, 2011. [Online]. Available: https://doi.org/10.1137/1.9781611970739

[6] ——, *Iterative methods for sparse linear systems*. Philly: SIAM, 2003. [Online]. Available: https://doi.org/10.1137/1.9780898718003

[7] M. Naumov, "S-step and communication-avoiding iterative methods," Technical Report NVR-2016-003, NVIDIA, Tech. Rep., 2016. [Online]. Available: https://research.nvidia.com/sites/default/files/pubs/2016-04_S-Step-and-Communication-Avoiding/nvr-2016-003.pdf

[8] Z. Xu, J. J. Alonso, and E. Darve, "A numerically stable communication-avoiding s-step GMRES algorithm," *SIAM J. Matrix Anal. Appl.*, vol. 45, no. 4, pp. 2039–2074, 2024. [Online]. Available: https://doi.org/10.1137/23m1577109

[9] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, 2011. [Online]. Available: https://doi.org/10.1145/2049662.2049663

[10] H. Anzt, T. Cojean, G. Flegar, F. Göbel, T. Grützmacher, P. Nayak, T. Ribizel, Y. M. Tsai, and E. S. Quintana-Ortí, "Ginkgo: A Modern Linear Operator Algebra Framework for High Performance Computing," *ACM Transactions on Mathematical Software*, vol. 48, no. 1, pp. 2:1–2:33, Feb. 2022. [Online]. Available: https://doi.org/10.1145/3480935

[11] I. S. Duff, A. M. Erisman, and J. K. Reid, *Direct methods for sparse matrices*. New York: Oxford University Press, 2017. [Online]. Available: https://doi.org/10.1007/3-540-12676-7

[12] G. H. Golub and C. F. Van Loan, *Matrix computations - 4th Edition*. Philadelphia: JHU press, 2013. [Online]. Available: https://doi.org/10.1137/1.9781421407944

[13] C. Alappat, A. Basermann, A. R. Bishop, H. Fehske, G. Hager, O. Schenk, J. Thies, and G. Wellein, "A recursive algebraic coloring technique for hardware-efficient symmetric sparse matrix-vector multiplication," *ACM Trans. Parallel Comput.*, vol. 7, no. 3, jun 2020. [Online]. Available: https://doi.org/10.1145/3399732

[14] M. Panzlaff, "Matrix power kernels on gpus," Master's thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg, 2024. [Online]. Available: https://hpc.fau.de/files/2024/10/MA-Panzlaff.pdf

[15] T. Iwashita, H. Nakashima, and Y. Takahashi, "Algebraic block multi-color ordering method for parallel multi-threaded sparse triangular solver in iccg method," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, 2012, pp. 474–483. [Online]. Available: https://doi.org/10.1109/IPDPS.2012.51

[16] H. Qiu, C. Xu, J. Fang, L. Deng, J. Zhang, Q. Wang, Y. Ding, Z. Dai, Y. Che, S. Chen, and J. Liu, "Towards scalable unstructured mesh computations on shared memory many-cores," in *Proceedings of ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. New York, NY, USA: Association for Computing Machinery, 2024, p. 109–119. [Online]. Available: https://doi.org/10.1145/3627535.3638473

[17] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM J. Sci. Comput.*, vol. 20, pp. 359–392, 1998. [Online]. Available: https://doi.org/10.5555/305219.305248

[18] C. Y. Lee, "An algorithm for path connections and its applications," *IRE transactions on electronic computers*, vol. EC-10, no. 3, pp. 346–365, 2009. [Online]. Available: https://doi.org/10.1109/TEC.1961.5219222

[19] H. Qiu, C. Xu, J. Fang, J. Zhang, L. Deng, Y. Ding, Q. Wang, S. Chen, Y. Che, and J. Liu, "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*. Piscataway: IEEE Press, 2024. [Online]. Available: https://doi.org/10.1109/SC41406.2024.00054

[20] A. George, "Nested dissection of a regular finite element mesh," *Siam Journal on Numerical Analysis*, vol. 10, no. 2, pp. 345–363, 1973. [Online]. Available: https://doi.org/10.1137/0710032

[21] T. Gale, M. A. Zaharia, C. Young, and E. Elsen, "Sparse gpu kernels for deep learning," *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–14, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:219955899

[22] H. Qiu, C. Xu, J. Fang, J. Zhang, L. Deng, Z. Dai, Y. Ding, Y. Wang, Z. Han, Y. Che, and J. Liu, "Dcsolver: Accelerating sparse iterative solvers via divide-and-conquer on gpus," *ACM Trans. Archit. Code Optim.*, Jun. 2025, just Accepted. [Online]. Available: https://doi.org/10.1145/3746233

[23] J. Xia, C. Cheng, X. Zhou, Y. Hu, and P. Chun, "Kunpeng 920: The first 7nm chiplet-based 64-core arm soc for cloud services," *IEEE Micro*, vol. 41, no. 5, pp. 67–75, 2021. [Online]. Available: https://doi.org/10.1109/MM.2021.3085578

[24] Intel Corporation, *Intel® Xeon® CPU Max Series Configuration and Tuning Guide*, August 2023. [Online]. Available: https://www.intel.com/content/www/us/en/content-details/787743/intel-xeon-cpu-max-series-configuration-and-tuning-guide.html

[25] Y. Chen, N. Finamore, F. Han, and R. E. Spangler, "Implementing high bandwidth memory and intel xeon processors max series on lenovo thinksystem servers," Lenovo Press, Technical Paper, June 2023. [Online]. Available: https://lenovopress.lenovo.com/lp1738-implementing-intel-high-bandwidth-memory

[26] J. Treibig, G. Hager, and G. Wellein, "Likwid: A lightweight performance-oriented tool suite for x86 multicore environments," in *international conference on parallel processing workshops*. Piscataway: IEEE, 2010, pp. 207–216. [Online]. Available: https://doi.org/10.1109/ICPPW.2010.38

[27] NVIDIA Corporation, *NVIDIA Nsight Compute*, NVIDIA Corporation, 2023, available at: https://developer.nvidia.com/nsight-compute.

[28] C. L. Alappat, J. Thies, G. Hager, H. Fehske, and G. Wellein, "Algebraic temporal blocking for sparse iterative solvers on multi-core CPUs," *The International Journal of High Performance Computing Applications*, vol. 39, no. 2, pp. 230–250, 2025.

[29] M. Hoemmen, *Communication-avoiding Krylov subspace methods*. Berkeley: University of California, Berkeley, 2010. [Online]. Available: https://dl.acm.org/doi/book/10.5555/1970638

[30] S. Filippone, V. Cardellini, D. Barbieri, and A. Fanfarillo, "Sparse matrix-vector multiplication on gpgpus," *ACM Transactions on Mathematical Software*, vol. 43, no. 4, pp. 1–49, 2017. [Online]. Available: https://doi.org/10.1145/3017994

[31] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst, *Templates for the solution of linear systems: building blocks for iterative methods*. Philly: SIAM, 1994. [Online]. Available: https://doi.org/10.1137/1.9781611971538

[32] R. W. Vuduc, *Automatic performance tuning of sparse matrix kernels*. Berkeley: University of California, Berkeley, 2003. [Online]. Available: https://doi.org/10.5555/1023242

[33] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks," in *Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures*. New York, NY, USA: Association for Computing Machinery, 2009, p. 233–244. [Online]. Available: https://doi.org/10.1145/1583991.1584053

[34] J. Demmel, M. F. Hoemmen, M. Mohiyuddin, and K. A. Yelick, "Avoiding communication in computing krylov subspaces," University of California, Berkeley, Tech. Rep. UCB/EECS-2007-123, 2007. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-123.html

[35] E. C. Carson, *Communication-avoiding Krylov subspace methods in theory and practice*. Berkeley: University of California, Berkeley, 2015. [Online]. Available: https://doi.org/10.1093/acprof:oso/9780198508380.001.0001

[36] J. Dongarra, S. Tomov, P. Luszczek, J. Kurzak, M. Gates, I. Yamazaki, H. Anzt, A. Haidar, and A. Abdelfattah, "With extreme computing, the rules have changed," *Computing in Science & Engineering*, vol. 19, no. 3, pp. 52–62, 2017. [Online]. Available: https://doi.org/10.1109/MCSE.2017.48

[37] M. Mohiyuddin, M. Hoemmen, J. Demmel, and K. Yelick, "Minimizing communication in sparse matrix solvers," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. New York, NY, USA: Association for Computing Machinery, 2009, pp. 1–12. [Online]. Available: https://doi.org/10.1145/1654059.1654096

[38] T. Muranushi and J. Makino, "Optimal temporal blocking for stencil computation," *Procedia Computer Science*, vol. 51, pp. 1303–1312, 2015. [Online]. Available: https://doi.org/10.1016/j.procs.2015.05.315

[39] J. Morlan, S. Kamil, and A. Fox, "Auto-tuning the matrix powers kernel with sejits," in *High Performance Computing for Computational Science*. Berlin: Springer, 2013, pp. 391–403. [Online]. Available: https://doi.org/10.1007/978-3-642-38718-0_36

[40] D. Huber, M. Schreiber, and M. Schulz, "Graph-based multi-core higher-order time integration of linear autonomous partial differential equations," *Journal of Computational Science*, vol. 53, pp. 101–349, 2021. [Online]. Available: https://doi.org/10.1016/j.jocs.2021.101349

[41] M. MehriDehnavi, Y. El-Kurdi, J. Demmel, and D. Giannacopoulos, "Communication-avoiding krylov techniques on graphic processing units," *IEEE Transactions on Magnetics*, vol. 49, no. 5, pp. 1749–1752, 2013. [Online]. Available: https://doi.org/10.1109/TMAG.2013.2244861

[42] I. Yamazaki, H. Anzt, S. Tomov, M. Hoemmen, and J. Dongarra, "Improving the performance of ca-gmres on multicores with multiple gpus," in *International Parallel and Distributed Processing Symposium*, 2014, pp. 382–391. [Online]. Available: https://doi.org/10.1109/IPDPS.2014.48