Memory-aware Optimization for Sequences of Sparse Matrix-Vector Multiplications

Yichen Zhang*, Shengguo Li*, Fan Yuan†, Dezun Dong*, Xiaojian Yang*, Tiejun Li*, Zheng Wang‡
**National University of Defense Technology,* China
{zhangyichen, nudtlsg, dong, yangxj, tjli}@nudt.edu.cn
†*Xiangtan University,* China
fyuan@smail.xtu.edu.cn
‡*University of Leeds,* United Kingdom
z.wang5@leeds.ac.uk

***Abstract*—This paper presents a novel approach to optimize multiple invocations of a sparse matrix-vector multiplication (SpMV) kernel performed on the same sparse matrix A and dense vector x, like $Ax, A^2x, \cdots, A^kx$, and their linear combinations such as $Ax + A^2x$. Such computations are frequently used in scientific applications for solving linear equations and in multi-grid methods. Existing SpMV optimization techniques typically focus on a single SpMV invocation and do not consider opportunities for optimization across a sequence of SpMV operations (SSpMV), leaving much room for performance improvement. Our work aims to bridge this performance gap. It achieve this by partitioning the sparse matrix into submatrices and devising a new computation pipeline that reduces memory access to the sparse matrix and exploits the data locality of the dense vector of SpMV. Additionally, we demonstrate how our approach can be integrated with parallelization schemes to further improve performance. We evaluate our approach on four distinct multi-core systems, including three ARM and one Intel platform. Experimental results show that our techniques improve the standard implementation and the highly-optimized Intel math kernel library (MKL) by a large margin.**

***Index Terms*—Sparse matrix-vector computation, color re-ordering parallelizaiton, multi-cores**

I. Introduction

The sparse matrix-vector multiplication (SpMV) operation is one of the most common operations in scientific and high-performance (HPC) workloads. An SpMV operation, $y = Ax$, multiplies a sparse matrix A of size $n \times n$ by a dense vector x of size n, and then produces a dense vector y of size n. Despite its prevalence and importance, SpMV is is often responsible for the performance bottleneck [1]–[3]. Unfortunately, optimizing SpMV for modern CPU architectures presents significant challenges [4]–[6]. In particular, SpMV is characterized by low arithmetic intensity and irregular memory access patterns [7], [8], making memory optimization for SpMV kernels highly important.

Efforts have been made to optimize SpMV by optimizing the sparse matrix storage format [5], [9]–[11] and the computation kernel [12], [13]. These prior optimizations have primarily focused on the computation of a single, isolated SpMV invocation. While important, they ignore the optimization opportunities for a sequence of SpMV (`SSpMV`) operations [14], [15]. An SSpMV pattern involves performing a series of SpMV operations, such as $Ax, A^2x, A^3x, \cdots, A^kx$, and their linear combinations, such as $A^2x + Ax$, where the same sparse matrix A is reused across invocations. This computing pattern is commonly seen in solving eigenvalue problems [16]–[19], linear equations [20], [21], and multigrid methods [22] that underpin various scientific applications. Given the ubiquity and importance of SSpMV in scientific computing, there is a critical need to identify optimization opportunities for SSpMV.

In this paper, we introduce a new approach for optimizing sequence sparse matrix-vector multiplication (SSpMV) by leveraging the sparse matrix data locality across consecutive SpMV invocations. Our approach involves revising the classical SSpMV computation pipeline used by mainstream algorithms to reduce the number of data accesses to the sparse matrix A. Specifically, we divide the sparse matrix into submatrices and reorganize the elements of dense vectors x to merge computation across multiple SpMV operations and reduce the number of data accesses to submatrices, without altering the computation semantics or outcome. As we will show later in the paper, our approach can half the number of memory accesses to the submatrices in certain scenarios, significantly reducing the memory overhead. To expose more parallelism, we then employ the algebraic block multi-color ordering (ABMC) algorithm [23] to rearrange the elements of matrix A. Crucially, this preprocessing step is a one-off cost and hence its overhead can be amortized if matrix A is reused multiple times in SSpMV.

Our techniques have been implemented in the open-source FBMPK library[1]. FBMPK is designed to support generic sequence sparse matrix-vector multiplication (SSpMV) of the form $y = \sum_{i=0}^{k} \alpha_i A^i x$, where α_i are real or complex constants. We evaluate our approach by applying it to the sparse matrix-power-vector kernel (MPK) [14], [15], which represents an instance of SSpMV. We test the performance of our system on four multi-core systems: three ARM systems, including Phytium (FT) 2000+ [24], Kunpeng (KP) 920 [25], and Thunder X2 [26], as well as an Intel platform. Experimental results show that FBMPK improves the standard SSpMV implementation by an average of 1.50x (up to 2.27x) on ARM

[1]Code and data are available at https://github.com/Fliange/FBMPK

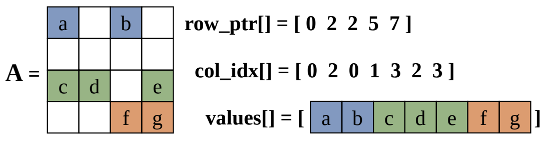

A =
a
b
c
d
e
f
g
row_ptr[] = [0 2 2 5 7]
col_idx[] = [0 2 0 1 3 2 3]
values[] = [a b c d e f g]

Fig. 1. Sparse matrix A and its corresponding CSR storage format.

Algorithm 1: The standard MPK of $A^k x$

```
Input: Matrix $A$, vector $x$ and $k$.
Output: Vector $y$.
for $power = 0$ to $k-1$ do
    SpMV($A$, $x$, $y$)
    $x = y$
end
function SpMV($A$, $x$, $y$):
    for $i = 0$ to $A.nrow - 1$ do
        $sum = 0$
        for $j$=$A.row\_ptr[i]$ to $A.row\_ptr[i+1]-1$ do
            $sum += A.val[j] * x.val[A.col\_ind[j]]$
        end
        $y.val[i] = sum$
    end
end
```

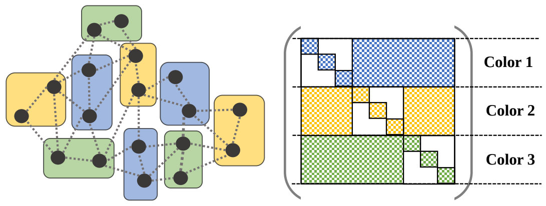


Fig. 2. An ABMC method with a block size of 2 and 3 colors (a), and a potential outcome for applying ABMC to a sparse matrix (b). Each dot in (a) corresponds to a row of the matrix in (b).

platforms. On the heavily optimized Intel Math Kernel Library (MKL) [27], FBMPK improves the performance by an average of 1.73x (up to 2.32x).

This paper makes the following contributions:

- It proposes a new, better algorithm to compute SSpMV (Section III-B);
- It shows how the dense vector storage format can be tailored to improve the memory access latency of SSpMV (Section III-C);
- It demonstrates how the multi-color reordering algorithm can be employed to expose parallelism for SSpMV (Sections III-D and III-E).

II. Background

A. The CSR Sparse Matrix Storage Format

In this paper, we describe and evaluate the proposed techniques using the *compressed sparse row* (CSR) format [28]. This format is widely used to store sparse matrices. However, our techniques can be equally applied to other mainstreamed row-major sparse matrix storage formats.

As shown in Fig 1, the CSR format explicitly stores column indices and nonzeros in arrays `col_idx` and `values`, respectively. Both arrays `col_idx` and `values` are of size `nnz`, where `nnz` is the number of nonzero elements of the input matrix A. CSR uses a vector `row_ptr`, which points to row starts in indices and data, to query matrix values. The length of `row_ptr` is $n+1$, where n is the number of rows of matrix A and the last item of `row_ptr` is the total number of the nonzero elements of the matrix.

B. The Sparse Matrix-Power-vector Kernel

Unlike prior work that focuses on optimizing a single SpMV innovation, our work targets SSpMV, where the sparse matrix A is used multiple times in more than one SpMV invocation. One representative example of SSpMV is the *sparse matrix-power-vector kernel* (`MPK`) . The MPK kernel performs a series of SpMV operations on sparse matrix A and vector x, as Ax, A^2x, A^3x, $\cdots$, A^kx, where the power, k, is a constant number. Compared to a single SpMV operation where A and x are used only once, the matrix and vector are reused multiple times across SpMV innovations in MPK. Note that the MPK is widely seen in multigrid methods [22] and solutions for eigenvalue problems [16]–[19] and linear equations [20], [21]. As such, it represents an important application class.

Algorithm 1 outlines the standard implementation of the MPK. This is achieved by performing a sequence of SpMV operations, i.e., $x_i = Ax_{i-1}$ with $i = 1, \cdots, k$ using a standard SpMV kernel for each invocation. In this paper, we use the standard MPK implementation as the *baseline* to evaluate our approach. Our goal is to reduce the memory access latency of MPK to improve its performance.

C. Matrix Reordering Techniques

Matrix reordering is a commonly used technique in SpMV and direct methods for solving sparse linear equations. The widely used reordering methods include reverse Cuthill–McKee algorithm (RCM) [29] to improve data locality, level scheduling [30], [31] to introduce parallelism in solving sparse triangular linear systems, and so on. Our work utilizes the Algebraic Block Multi-Color Ordering Method (ABMC) [32] to partition and reorder the sparse matrix to expose computation parallelism. The ABMC method extends the block multi-color ordering that is widely used to solve unstructured problems [33]. Fig 2(a) illustrates the ABMC method with nine blocks and three colors, where each block contains two nodes (or rows) and blocks with the same color can be processed in parallel. Fig 2(b) shows a potential outcome of applying the ABMC method to partition and reorder elements of a sparse matrix.

III. Our Approach

In this section, we use the MPK method (Section II-B) as a working example to explain our approach. However, our techniques can be equally applied to other SSpMV kernels in the form of $y = \sum_{i=0}^{k} \alpha_i A^i x$.

A. Submatrices for Sparse Matrix A

Our approach partitions and stores sparse matrix A as submatrices. Specifically, we split A into three submatrices, $A = U + L + D$, where U and L are the upper and lower triangular matrices, respectively, and D is the diagonal matrix. Note that we store D as a vector to reduce storage and computation overhead. With our matrix partitioning strategy, the SpMV operation of Ax can then be computed as $Ax = (D + L + U)x = Lx + (D + U)x$ or $(L + D)x + Ux$. As we will describe in the next subsection, this strategy allows us to merge the computation across SpMV invocations to reduce the number of memory accesses to submatrices L and U to improve performance. We note that a similar matrix partitioning strategy was used in a different context to optimize the symmetric Gauss-Seidel (SYMGS) method [34].

B. A New SSpMV Processing Pipeline

With our sparse matrix partitioning strategy in place, we can now develop a new and better SSpMV processing pipeline across SpMV invocations. We describe our approach in this subsection, using the standard MPK implementation as a reference baseline.

Standard MPK. Fig 3(a) outlines the processing pipeline of the standard MPK, which computes the result by multiplying sparse matrix A with the outcome of the previous SpMV invocation. This strategy requires k memory accesses to matrix A for an MPK kernel of $A^k x_0$, which is computed as $x_i = Ax_{i-1}; i = 1, \cdots, k$.

Forward-backward MPK. As depicted in Fig 3(b), our proposed processing pipeline, namely *forward-backward MPK* (FBMPK), consists of two stages: a forward and a backward stage. By splitting A into $A = D + L + U$, an SpMV operation $y = Ax$ is translated into computing the sum of three terms, $y = Dx + Lx + Ux \equiv x_d + x_L + x_U$. Here, $x_L = Lx$ denotes the contribution of L and x to vector y, and so do $x_d = Dx$, and $x_U = Ux$. With our implementation, we compute x_L and x_U in the forward and the backward stages respectively.

Forward stage. Fig 4 shows the process of computing $x_1 = Ax_0$, focusing on the forward stage. First, we compute the product of U and x_0 in parallel and temporarily store the result in $x_{U_1} (= Ux_0)$; see Fig 4(a). In the forward stage, we access rows of L from top to bottom. Crucially, we compute Lx_0 and Lx_1 in a pipelined manner. This is done by merging the computation process of x_{L_1} and x_{L_2}. This is illustrated in Fig 4(b), where $x_{L_1} = Lx_0$ and $x_{L_2} = Lx_1$. Let's consider the particular row highlighted in Fig 4(b). This row is used to compute the fourth element of vector x_{L_1}. Since the submatrix L stores the lower triangular elements, only the first three elements of x_1 are required to compute the fourth entry of x_{L_2}. Since the elements of x_1 have already been computed, we can start computing the fourth entry of x_{L_2}. In essence, our optimization exploits pipeline parallelism across two SpMV invocations to improve computation parallelism. We note that the computed entry of x_{L_1} will be immediately combined with x_{U_1} and x_{d_1} to compute the element of x_1, as shown in Fig 4(c). The computed elements of x_{L_2} will be temporarily stored in x_{L_2}, which will be used in the backward stage.

Backward stage. Like the forward stage, the backward stage computes Ux_i to obtain $x_{U_{i+1}}$ (with $i = 1$ in Fig 4). The result will then be added with x_{L_2} produced in the forward stage and $x_{d_2} = Dx_1$. Finally, we obtain MPK $x_2 = Ax_1$. As shown in Fig 4(d), submatrix U in the backward stage is accessed from bottom to top, and entries of x_2 are also computed bottom-up. Therefore, the multiplication of U and x_2 can also be computed in the backward stage of Fig 4(d), from bottom to top. To this end, we merge the compute processes of x_{U_i} and $x_{U_{i+1}}$ in the backward stage, for $i = 1, 2, \ldots, k-1$.

Memory optimization. In FBMPK, two consecutive vectors are computed together. In the forward stage of Fig 4(b), the contributions of submatrix L to vectors x_i and x_{i+1}, x_{L_i} and $x_{L_{i+1}}$, are computed at the same time. As a result, we only visit the submatrix L once in the forward stage. Similarly, in the backward stage, submatrix U is only accessed once and used to update both x_{U_i} and $x_{U_{i+1}}$. As shown in Fig 3(b), to compute the MPK $A^k x$, FBMPK needs to access U for $2 + \lfloor \frac{k-1}{2} \rfloor = \frac{k}{2} + 1$ times and L for $\frac{k}{2}$ times when k is even, and both U and L for $1 + \frac{k-1}{2}$ times when k is odd. Therefore, FBMPK needs to access matrix A for roughly $\frac{k+1}{2}$ times. Comparing with the standard MPK which accesses k times of A, the memory access of FBMPK is reduced nearly by half.

C. Optimize Memory-Access to Vector: Back to Back

In MPK computation, dense vectors are typically stored in a separate array of length n. FBMPK visits the same position of two vectors in a row to update the latter vector. However, the two vectors are physically stored independently, hindering data locality. Fig5 demonstrates the access of two vectors x_i and x_{i+1} in the iterative process. To enhance locality, we merge these two vectors into a single array of length $2n$, while logically separating them. The two vectors are stored interleaved, as depicted as vector xy in Fig5.

D. Matrix Reordering for Parallelization

So far, we have described our techniques under the assumption of a single-threaded MPK kernel. When parallelizing the kernel, data dependence must be addressed. For instance, Fig 6 illustrates the potential dependence that can occur when multiple threads access submatrix L and vector x simultaneously. The figure shows how each parallel thread processes a row in a 5×5 matrix L. If thread 4 completes the computation of $x_{L_1}[4]$ (the bottom element of vector x_{L_1}) first, it will proceed to compute x_{L_2} (the bottom element of vector x_{L_2}). However, as threads 0 and 1 may not have finished calculating $x_{L_1}[0]$ and $x_{L_1}[1]$ (the top two elements in x_{L_1}), a synchronization point must be inserted to pause thread 4 until the other parallel threads complete their work. Unfortunately, synchronization barriers can incur substantial overhead for fine-grained parallelism and should be avoided if possible.

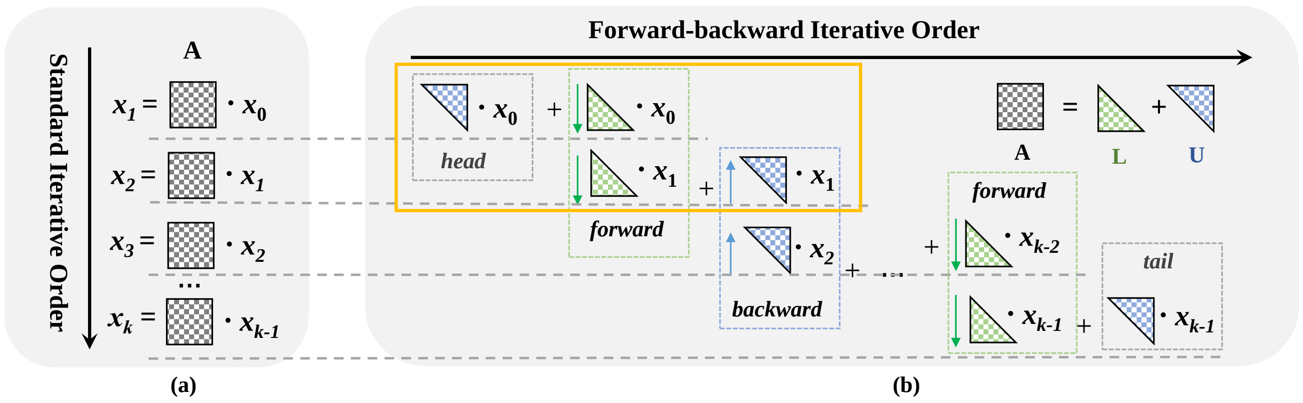


Fig. 3. Overview of two MPK methods. The baseline method, shown in (a), uses the standard iterative order. Our new forward-backward method, shown in (b), performs MPK calculations in forward-backward iterative order. The baseline needs to read the matrix k times from memory. By comparison, our forward-backward method only needs to read the full matrix from memory $\frac{k+1}{2}$ times. Note that, we ignore the representation of the diagonal matrix.

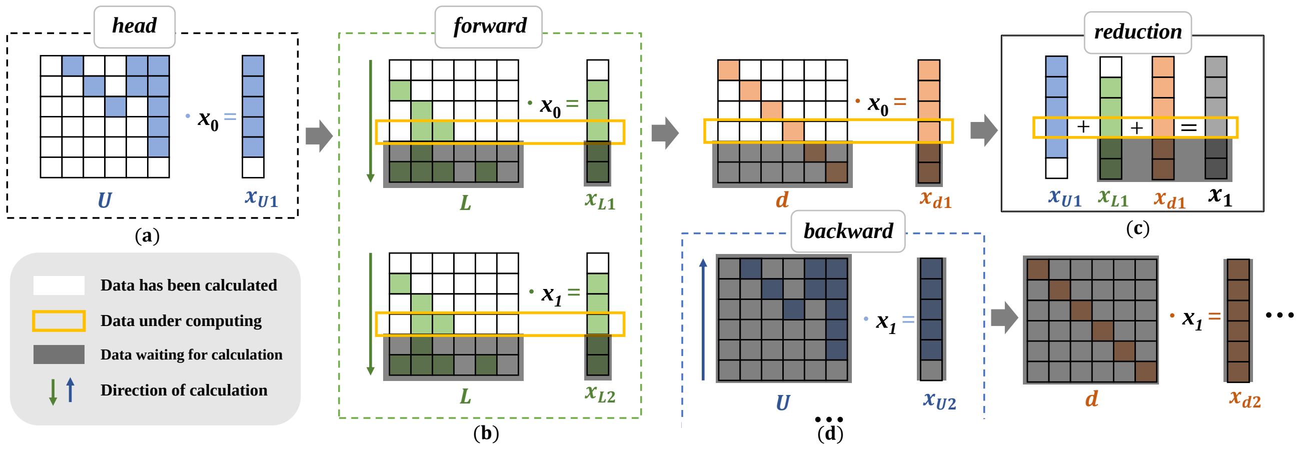


Fig. 4. How FBMPK performs computation for $x_2 = A^2x_0$ (i.e, $x_1 = Ax_0$, $x_2 = Ax_1$) for yellow rectangles in Fig 3. FBMPK requires 1.5 memory accesses to matrix A by exploiting data reuse. By comparison, the standard MPK needs to access A twice.

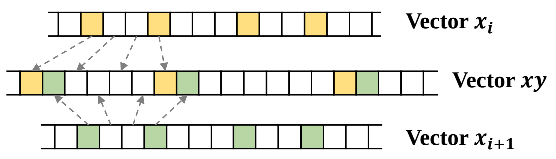


Fig. 5. Different ways of memory access of dense vector.

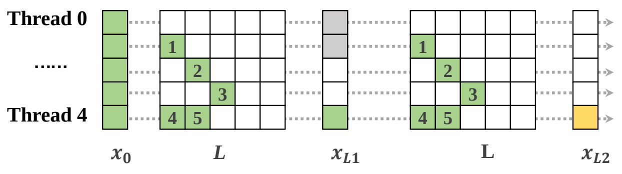


Fig. 6. Data dependencies under multithreading.

Our approach to parallelizing FBMPK involves utilizing the ABMC algorithm [23] to rearrange the matrix elements. The algorithm organizes the matrix elements into blocks and uses coloring to indicate dependencies, so blocks with the same color can be processed simultaneously. The maximum number of elements in each block can be set, with a trade-off between performance and parallelism, and our implementation allows the user to specify the number of blocks, with a default of either 512 or 1024. In this work, we use the Colpack [35] library to assign colors to blocks.

E. Putting Together

Algorithm 2 describes our FBMPK algorithm with the amendments to the matrix and vector storage formats our parallelization strategy. This algorithm computes $A^s x, s \leq k$ where s is an odd number. A similar algorithm can also be devised to compute $A^s x$ when s is an even number.

The operations of head and tail are parallel SpMV (Lines 3 and 31). During the forward-backward process, computations are performed in parallel in the same color (Line 6) after applying the ABMC algorithm to partition the matrix. The number of blocks for each thread task are allocated in advance. At computation time, each thread gets the start and end of the row respectively (Lines 7 and 19). In the forward stage, Lines 11 and 12 are the core of the algorithm, which completes the calculation of x_L twice. Vector xy is the interleaved storage

Algorithm 2: Parallel FBMPK for computing $A^k x$

```
Input: matrix $L$, matrix $U$, vector $dia$, vector $x$, and $k$.
Output: vector $y$.
1  Function FBMPK_odd($L$, $U$, $dia$, $y$, $x$, $k$)
2      // Computing submatrix $U$ in head.
3      do $SpMV(U, x, tempvec)$ in parallel.
4      for $power = 0$ to $k - 1$ do
5          // Computing submatrix $L$ in forward order.
6          Blocks of each color are calculated in parallel.
7          for $row_i = start[t]$ to $end[t] - 1$ in thread t do
8              $sum0 = tmpvec[row_i] + dia[row_i] * xy[row_i * 2]$
9              $sum1 = 0.0$
10             for $j = L.row\_ptr[row_i]$ to
                 $L.row\_ptr[row_i + 1] - 1$ do
11                 $sum0 += L.val[j] * xy[L.col\_ind[j] * 2]$
12                 $sum1 += L.val[j] * xy[L.col\_ind[j] * 2 + 1]$
13             end
14             $xy[row_i * 2 + 1] = sum0$
15             $tmpvec[row_i] = sum1 + dia[row_i] * xy[row_i * 2 + 1]$
16         end
17         // Computing submatrix $U$ in backward order.
18         Blocks of each color are calculated in parallel.
19         for $row_i = end[t] - 1$ to $start[t]$ in thread t do
20             $sum0 = tmpvec[row_i]$
21             $sum1 = 0.0$
22             for $j = U.row\_ptr[row_i + 1] - 1$ to
                 $U.row\_ptr[row_i]$ do
23                 $sum0 += U.val[j] * xy[U.col\_ind[j] * 2 + 1]$
24                 $sum1 += U.val[j] * xy[U.col\_ind[j] * 2]$
25             end
26             $xy[row_i * 2] = sum0$
27             $tmpvec[row_i] = sum1$
28         end
29     end
30     // Computing submatrix $L$ in tail.
31     do $SpMV(L, xy, lvec)$ in parallel.
32     $y = lvec + tmpvec + diavec$
33 end
```

TABLE I
HARDWARE PLATFORMS USED IN EVALUATION.

	FT 2000+	Thunder X2	KP 920	Xeon
#Cores	64	32	64	26
Sockets	1	2	2	2
#NUMAs	8	1	1	2
CPU Freq.	2.2GHz	2.5GHz	2.6GHz	2.1GHz
L1 cache	32KB	32KB	64KB	64KB
L2 cache	2MB	256KB	512KB	1MB
L3 cache	None	32MB	64MB	35.75MB

vector introduced in section III-C. We always initialize x_0 at the even position of the vector xy. The merged result x_i is stored in the vector xy (Lines 14 and 26), and vector $tmpvec$ is used to temporarily store x_L or x_U (Lines 15 and 27).

IV. EXPERIMENTAL SETUP

A. Evaluation Platforms

We evaluate our approach on Intel and ARM platforms. Table I lists the hardware platforms used, including three ARMv8 multi-cores and an Intel Xeon Gold 6230R CPU. Our main evaluation platform is FT 2000+, which implements 8 NUMA nodes with 64 cores. Since the data access across NUMA nodes can cause significant performance degradation, it is more challenging to optimize SSpMV on FT 2000+ than on other platforms. All hardware platforms run Linux kernel version 4.19.46. Codes on ARM platforms are compiled with GCC version 12.1 using the "-O3 -fopenmp" options, and codes on Xeon are compiled with Intel compiler ICC version 2022.1.2-146. We use $numactl$ to interleave memory allocation on all NUMA nodes.

TABLE II
INPUT MATRICES USED IN OUR EVALUATION

ID	Input	Rows(N)	#nnz	#nnz/N
1	afshell10	1.51M	52.67M	34.93
2	audikw_1	0.94M	77.65M	82.28
3	cage14*	1.51M	27.13M	18.02
4	cant	0.06M	4.01M	64.17
5	Flan_1565	1.56M	117.41M	75.03
6	G3_circuit	1.59M	7.66M	4.83
7	Hook_1498	1.50M	60.92M	40.67
8	inline_1	0.50M	36.82M	73.09
9	ldoor	0.95M	46.52M	48.86
10	ML_Geer*	1.50M	110.88M	73.72
11	nlpkkt120	3.54M	96.85M	27.34
12	pwtk	0.22M	11.63M	53.39
13	Serena	1.39M	64.53M	46.38
14	shipsec1	0.14M	7.81M	55.46

*These two matrices are unsymmetric. Others are symmetric.

B. Workloads

Table II lists the matrices used in our evaluation. The matrices used in our evaluation are of different sizes and sparsities. They cover various domains, including circuit simulation, optimization, structural and 2D/3D problems, and directed weighted graphs. Therefore, the dataset can be useful in testing the generalization ability of our approach. We also include some of the same matrix inputs used in prior work for optimizing SpMV [14], [36] and MPK [37] for a fair comparison.

C. Evaluation Methodology

We run each test case 50 times on unloaded machines and report the geometric mean of the runtime. Unless stated otherwise, execution time always refers to the computation time of SpMV or MPK. We also exclude the overhead for splitting and reordering elements of the sparse matrix, as this preprocessing step can often be performed offline when storing the matrix data.

On the Intel platform, we compare to the baseline MPK which uses the SpMV kernel from MKL. On the ARM platform, we compare our approach and the baseline MPK using the same optimized SpMV kernel. This SpMV kernel is heavily optimized and can even outperform the MKL implementation by 13% on our Intel platform.

V. EXPERIMENTAL RESULTS

In this section, we first show that our proposed SSpMV processing pipeline, FBMPK, outperforms the baseline implementation in the majority of the matrix inputs, with a speedup of up to 2.32x (Section V-A). We then provide a detailed analysis showing that our optimization scale well to a larger number of SpMV invocations by reducing the memory accesses (Sections V-B and V-C. We then quantify the impact of individual optimizations (Sections V-D - V-F) before showing the scalability of our approach (Section V-G).

A. Overall Performance

Figure 7 reports the speedup achieved by our proposed FBMPK approach over the baseline MPK on four platforms with power $k = 5$. FBMPK delivers notable speedups for most input matrices, with an average performance improvement of 1.50x, 1.54x, 1.47x, and 1.73x on FT 2000+, Thunder X2, KP 920, and Intel Xeon, respectively. Additionally, we observed a maximum speedup of up to 2.32x.

Recall that FBMPK roughly halves the number of memory accesses matrix A. As the locality of SpMV is different when accessing submatrices L and U compared to accessing the original sparse matrix A, the reduction in memory accesses translates into various speedup improvements across matrices. There are some matrices with a speedup of more than 2x. $inline\_1$ and $audikw\_1$ achieve a speedup of more than 2x, because these two matrices gain locality improvement after ABMC reordering. Later in Section V-E we will give a breakdown of our individual optimizations.

Across the various input matrices evaluated in our study, matrix $cant$ exhibits noticable performance variance between ARM and x86 platforms. Specifically, on ARM platforms, $cant$ has the smallest dimension among all input data, with only $62,451$ rows and the smallest number of elements. After reordering, it is divided into 512 blocks, with only 77 blocks (or even fewer) in one color to be run in parallel. Moreover, each block contains an average of only 120 rows of sparse matrix data. As a result, the computation workload of this matrix is too small to cover the thread overhead when using all physical cores. However, we found that using only 24 threads on the FT 2000+ platform yielded a speedup of 1.3x. In Section V-G, we discuss how the number of threads can affect performance. By comparison, the Xeon platform has only 26 hardware threads, yet our proposed FBMPK approach achieves a speedup of 1.72x on $cant$. Overall, despite the variability in performance across different matrices and platforms, FBMPK consistently outperforms the baseline on most of the input data, showing the advantages of our techniques.

B. Impact of SpMV Invocations

Figure 8 quantifies the performance impact of varying the MPK power factor, k, from 3 to 9 on different platforms. Generally, the benefits of our approach become increasingly pronounced as k increases with more SpMV invocations. For instance, when $k = 3$, our approach achieves an average speedup of 1.29x, 1.34x, 1.31x, and 1.42x on FT 2000+, Thunder X2, KP 920, and Intel Xeon, respectively. However, as we increase k to 9, the average speedup improves to 1.64x, 1.70x, 1.65x, and 1.85x on FT 2000+, Thunder X2, KP 920, and Intel Xeon, respectively.

An important reason for this improvement trend is that our approach computes half of the matrix elements in the head and tail parts, as depicted in Fig 3, and a larger number of k can lead to more reduction in the memory accesses. Specifically, when $k = 3$, there are 2 matrix memory accesses for A. By contrast, the baseline method requires 3 matrix memory accesses. When $k = 9$, our approach needs to access the memory of matrix A 5 times, while the baseline method needs to access the matrix in full for 9 times. In this case, our approach reduces the number of memory accesses by nearly half. Section V-C provides a more detailed analysis of memory accesses using runtime profiling. Overall, our approach is effective in reducing the memory accesses when computing SSpMV through multiple SpMV invocations.

C. Memory Accesses Profiling

In this experiment, we use the LIKWID performance monitoring tool [38] to further analyze memory accesses for MPK. We measure the total amount of data read and write from DRAM. We conduct this experiment on the Intel Xeon platform and compare our approach with the MKL library. Fig 9 gives the ratio of memory accesses of FBMPK over the MKL baseline. It shows the reduction in memory access for MPKs with power $k = 3$, 6 and 9, respectively.

In theory, the memory access ratio of our approach to the MKL baseline is $\frac{k+1}{2k}$. This translates to a reduction of 67%, 58% and 56% over the baseline when $k = 3$, 6 and 9, respectively. In reality, the memory accesses can be higher than theoretical number due to other overhead from optimizations, such as increased vector accesses. For example, in this experiment, when k is increased from 3 to 6 and 9, the average ratio of matrix memory accesses decreased from 74% to 65%, and then to 62%, respectively.

We also found that the decrease in memory access is directly related to the sparsity of the matrix. The most sparse matrix in our dataset is $G3\_circuit$, which has only, on average, 4.83 elements per row. It means that one access to the vector once is equivalent to one fifth visits of this matrix. The memory access ratio for this matrix is 77% when $k = 9$, as vector accesses dominate the total memory accesses. By contrast, our approach gives the largest reduction on the measured memory accesses of ML_Geer, with a ratio of 58% when $k = 9$. Overall, our approach can effectively reduce the number of memory accesses compared to the baseline but reduction changes depending on the sparsity of the matrix.

D. Vector Optimization Analysis

To demonstrate the impact of individual optimizations on the performance of our approach, we created an implementation variant that only incorporates our forward-backward (FB) method (Section III-B), which we compared with a version featuring the back-to-back (BtB) vector optimization (Section III-C). BtB involves storing the two vectors to be computed interleaved, with the non-zero elements of the vectors at the corresponding positions always adjacent (e.g., the highlighted cells in Fig 5). We refer to the latter version as FB+BtB.

Figure 10 compares FB and BtB on FT 2000+ and Xeon platforms. While both approaches can contribute to performance improvement, the benefit of BtB is more evident on

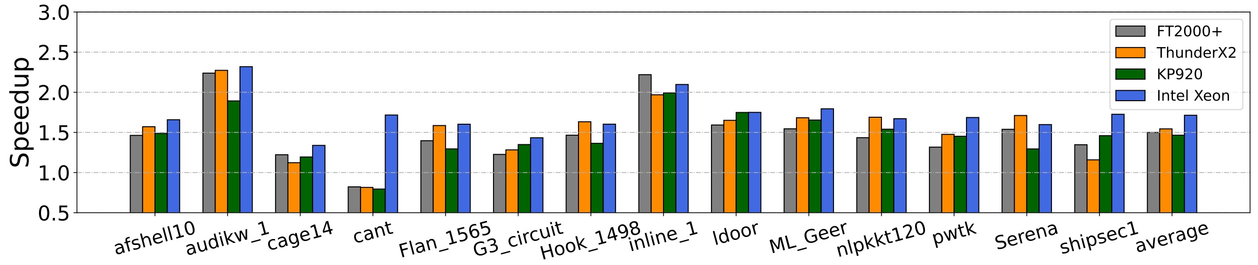


Fig. 7. Speedups of FBMPK over the baseline MPK with power k=5 on four different platforms.

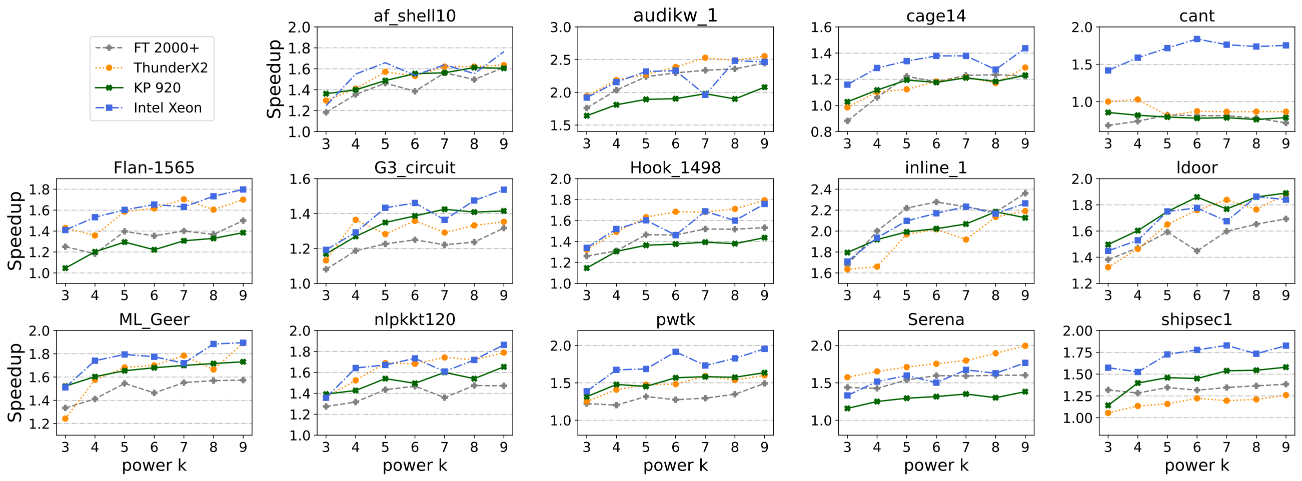


Fig. 8. Performance with different MPK k (power) values on our evaluation platforms.

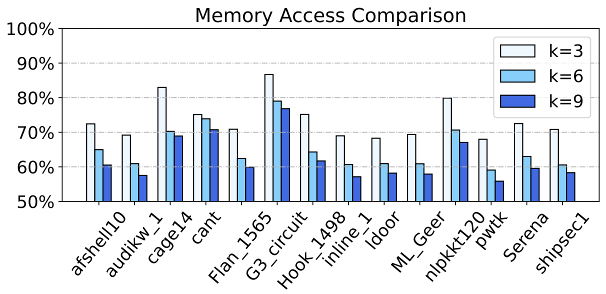


Fig. 9. DRAM read/write volume ratio between MKL and our approach under different MPK k settings on Xeon.

TABLE III
SPEEDUP NORMALIZATION TO THE NATIVE SPMV IMPLEMENTATION WHEN APPLYING ABMC

ID	Input	Slowdown	ID	Input	Slowdown
1	afshell10	1.01	2	audikw_1	**1.80**
3	cage14	1.00	4	cant	0.97
5	Flan_1565	1.00	6	G3_circuit	1.08
7	Hook_1498	1.01	8	inline_1	**1.44**
9	ldoor	1.06	10	ML_Geer	0.98
11	nlpkkt120	0.98	12	pwtk	1.02
13	Serena	1.04	14	shipsec1	1.04

FT 2000+. Specifically, BtB provides additional performance improvement on 13 input matrices on FT 2000+, with an average speedup of 1.50x, compared to a speedup of 1.41x with the FB method alone. Moreover, the performance benefit of BtB is transferable to the remaining two ARM platforms, yielding a 10% additional speedup. In contrast, BtB vector optimization offers only modest improvement on the Intel platform.

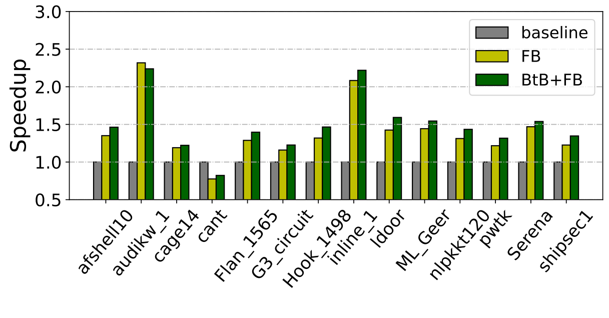
E. Impacts of Matrix Reordering

Recall that we employ the ABMC algorithm to reorder the sparse matrix to expose parallelism (Section III-D). While reordering is useful for parallelizing SSpMV, it may affect the performance of a single SpMV invocation due to the restructured sparse matrix. To understand the impact of ABMC on a single SpMV invocation, we measure slowdown by normalizing the time used when performing baseline SpMV on the original matrix and the one on the ABMC-transformed matrix.

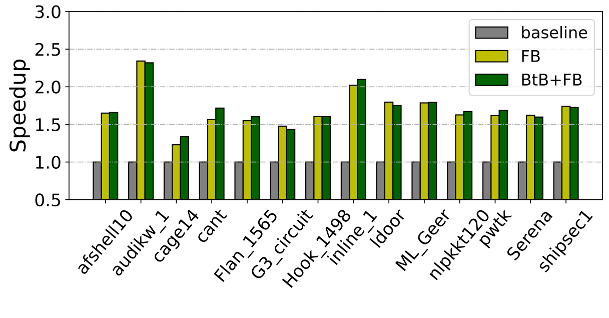
Table III shows the result on FT 2000+, where a ratio great than 1 means a performance improvement on a single SpMV invocation. For most of the matrices, we observe little impact or event performance improvement. On three matrices, there is a less than 3% slowdown, which can be amortized by the benefit of parallelization.

F. Preprocessing Overhead

Central to our idea is partitioning sparse matrix A into three submatrices: an upper triangular matrix U, a lower triangular matrix L, and a diagonal matrix D. In this work, L and U are stored in the CSR format, and D is stored in a array d



(a) FT 2000+



(b) Intel Xeon

Fig. 10. The impact of forward-backward (FB) MPK and back-to-back (BtB) vector optimizaiton on FT 2000+ (a) and Xeon (b) when power $k = 5$.

TABLE IV
STORAGE OVERHEAD

Format	*col_ind*	*row_ptr*	*values*	*d*
CSR	nnz	$n + 1$	nnz	0
L+U+d	$(nnz - n)$	$2 * (n + 1)$	$(nnz - n)$	n

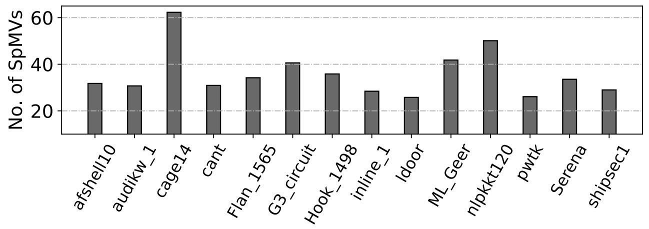
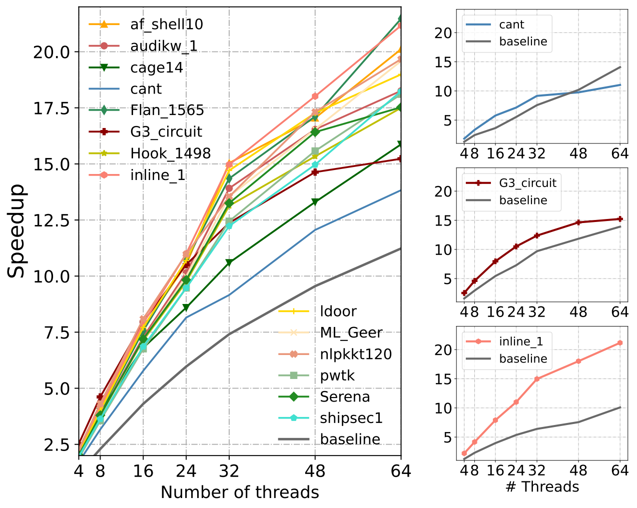


Fig. 11. The time required to reorder the matrix with ABMC, which is normalized to number of single thread SpMV invocations. Note that matrix preprocessing is a one-off cost, which can often be performed offline without impacting the runtime performance.

of length n which is the dimension of matrix A. Table IV compares our storage format with the CSR format, assuming the total number of non-zero elements of A is nnz, and it shows that the memory storage costs of these two formats are nearly the same.

We stress that the matrix partition can be performed offline. When using parallelization, our approach incurs extra overhead for using the ABMC to reorder the matrix elements. To quantify the overhead of matrix reordering, we measure the overhead by applying ABMC to each of our test matrices. In Fig 11, we normalize the preprocessing overhead to the execution time of single thread SpMV invocation. On average, the overhead of the ABMC algorithm is equivalent to 36 SpMV invocations. While this overhead seems to be significant, this reordering process can be performed offline without affecting the runtime. In many scientific workloads, like HPCG [34], a similar blocking algorithm is used to reorder the matrix elements to expose parallelism for other computation kernels. Furthermore, an MPK-like kernel is often invoked many times. As such, this *one-off* preprocessing overhead is usually negligible at runtime.



(a) (b)

Fig. 12. Scalability of our approach on FT 2000+ when $k = 5$. The speedup is normalized to the single-threaded execution of the baseline MPK implementation. Our approach exhibits good scalability.

G. Scalability Analysis

Fig 12(a) shows the scalability of our approach on FT 2000+ as we increase the number of parallel threads with $k = 5$ for the MPK kernel. The speedup is normalized to the single-threaded execution of the baseline MPK implementation. Our approach exhibits good scalability, where the average speedup is improved from 2.08x with 4 threads to 18.05x with 64 threads. We note that we also observe similar scalability on our other evaluation platforms.

Fig 12(b) provides a zoom-in analysis on three selected matrices with interesting behavior. As we have seen in the previous discussion, *cant* is the only matrix that our approach does not always outperform the baseline. Our approach outperforms the baseline when using less than 48 threads. Once again, this small matrix does not benefit from our optimization when using a large number of parallel threads where each thread works on a small portion of the data. This is because the overhead of multi-threading outweighs the improvement in computation. Our approach outperforms the baseline implementation on *G3_circuit*, although the scalability on this matrix is less strong than other matrices. Like *cant*, *G3_circuit* is also a small matrix that does not benefit from a high number of parallel threads. Finally, our approach gives the best scalability on *inline_*1. Although this is not the largest matrix, it benefits from ABMC reordering, which improves the cache utilization in a multi-threaded environment. Note that our approach also gives the fastest speedup on this matrix in matrix reordering (Section V-E).

Overall, our approach demonstrates good scalability and outperforms the baseline implementation on the majority of the evaluation scenarios.

VI. Related Work

There is an extensive body of work on optimizing SpMV. These include efforts on designing new storage formats for various processor architectures, including CPUs [4], [39] and GPUs [10], [40]–[42]. In this paper, we demonstrate the effectiveness of our approach on the commonly used CSR format. However, our optimization is equally applicable to many other sparse storage formats. Other approaches look at how to optimize a single SpMV kernel through vectorization [5], [9], [11] and parallelization [43]–[45]. Many of these techniques are orthogonal to our approach by improving a single SpMV kernel.

Our work specifically targets optimizing a sequence of SpMV invocations (SSpMV). This is an area largely overlook by prior work, which mainly focuses on optimizing a single, isolated SpMV kernel invocation. There are some domain-specific methods attempting to optimize the communications among parallel processes when computing formulas like $A^k x$ on distributed and shared memory platforms. These include the Krylov iterative methods [46], [47], and s-step Krylov methods [48]. These approaches typically use a block strategy developed for stencil computation [46] to partition the data for parallelization. Unlike these strategies, our work exploits the opportunities across SpMV invocations to optimize for memory and computation, but we also use a similar block partitioning strategy for parallelization.

LB-MPK, presented in a recent technical report [15], is most closely related to our approach (but we were unable to build their code on our platforms to provide a direct comparison). LB-MPK uses the recursive algebraic coloring engine (RACE) [37] to parallelize MPK. However, our approach is fundamentally different from LB-MPK in several ways. Firstly, we propose to split the sparse matrix into submatrices, while LB-MPK still uses the classical CSR format to store the entire matrix. Secondly, LB-MPK performance drops significantly with a larger k ($\approx 6-8$). This is because LB-MPK requires the multiple computation outcomes across SpMV invocations to be kept in the cache to gain good performance. However, doing so becomes increasingly more difficult as the number of SpMV invocations grows. By contrast, our approach only requires keeping the results of two consecutive SpMV invocations and can scale well to a large number of SpMV invocations. Thirdly, LB-MPK requires significantly higher preprocessing overhead than our approach. Finally, [14] reports the performance of SSpMV vector addition, $A(Ax + x) + x$, on the Fugaku's A64FX processor using a standard SSpMV implementation. However, it does not provide memory optimization like our work.

VII. Discussions

Naturally, there is room for improvement and further work. We discuss a few points here.

Compiler-based approach. One of our ongoing works is to use compiler-based techniques to translate the standard SpMV kernel and its inputs to use our optimizing library. It would be interesting to see if the compiler-based code translation can adapt to a wide range of kernels other than MPK to reduce developer involvement.

Sparse matrix storage formats. Our implementation splits matrix A to submatrices L and U, which are then stored in the CSR format. While this strategy already gives significant improvement, we are aware that CSR may prevent vectorization. Our future work will exploit other storage formats like ELLPACK [49] and Sliced ELL [5], and investigate if a better storage format can be designed for SSpMV.

Distributed and NUMA optimization. This work focuses on performance optimization on shared-memory multi-cores. Since our approach does not introduce extra synchronization and communication overhead, a distributed implmentation can directly benefit from the improved performance of a single CPU. Our current implementation does not model the NUMA impact. Therefore, NUMA-aware work distribution techniques [50] are complementary to our solution.

Other parallelization strategies. Since the computation pattern of FBMPK is similar to symmetric Gauss-Seidel (SYMGS) [34], it may benefit from similar parallelization strategies designed for SYMGS, such as the level scheduling strategies [51], besides of the multi-coloring strategy used in this work.

Heterogeneous devices. This work focuses on homogeneous multi-core CPUs because they remain the most widely used computing devices in HPC. Our future work will look into extending our techniques to the GPU space. Given the expensive memory access overhead on GPUs, memory optimization would become even more important. It would also be interesting to see if our techniques can be integrated with optimization designed for GPU-based SpMV kernels [41] and storage formats [10].

VIII. Conclusion

We have presented a new way to optimize multiple invocations of sparse matrix-vector multiplication (SpMV). This computation pattern can be widely found in scientific workloads but is largely ignored in prior work for SpMV optimization. Our approach reduces the memory access latency of the workload by exploiting the data locality of the sparse matrix and dense vector involved in the computation. To this end, we first partition the sparse matrix into submatrices, around which we then design a new processing pipeline across SpMV invocations. Our approach also employs the algebraic block multi-color ordering (ABMC) algorithm to preprocess the sparse matrix to expose parallelism. We apply our techniques to the matrix-power-vector kernel (MPK), $Ax, \cdots, A^k x$, and evaluate it on three ARM and one Intel systems. Experimental results show that our approach consistently improves over the baseline implementation, significantly outperforming the heavily-optimized SpMV implementation from the Intel MKL library, delivering a speedup of up to 2.32x.

Acknowledgement

This work was supported in part by the National Key R&D Program of China under grant agreement 2021YFB0300101, the National Science Foundation of China (NSFC) under grant agreements 61902411, 62032023, 12002382, 11275269, and 42104078, the Excellent Youth Foundation of Hunan Province under grant agreement 2021JJ10050, and a UK Royal Society International Collaboration grant. Yichen Zhang and Shengguo Li contributed equally to this work, and Dezun Dong and Tiejun Li are the corresponding authors.

References

[1] A. D. K. Kaushik, B. D. E. Keyes, and B. F. S. D, "Toward realistic performance bounds for implicit CFD codes," in *Proceedings of Parallel CFD'99*. Elsevier, 1999, pp. 233–240.

[2] J. R. Gilbert, S. Reinhardt, and V. B. Shah, "High-performance graph algorithms from parallel sparse matrices," in *In PARA'06: Proceedings of the 8th international conference on Applied parallel computing*, 2007, pp. 260–269.

[3] E. jin Im, E. jin Im, K. Yelick, and K. Yelick, "Optimization of sparse matrix kernels for data mining," in *First SIAM Conf. on Data Mining*, 2000.

[4] S. Williams, L. Oliker, R. Vudec, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," in *ACM/IEEE SC'07*, New York, 2007, pp. 38:1–38:12.

[5] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop, "A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units," *SIAM Journal on Scientific Computing*, vol. 36, no. 5, pp. C401–C423, 2014.

[6] R. Vuduc, J. W. Demmel, and K. A. Yelick, "OSKI: A library of automatically tuned sparse matrix kernels," in *Institute of Physics Publishing*, 2005.

[7] E.-J. Im, *Optimizing the performance of sparse matrix-vector multiplication*. University of California, Berkeley, 2000.

[8] J. Willcock and A. Lumsdaine, "Accelerating sparse matrix computations via data compression," in *Proceedings of the 20th annual international conference on Supercomputing*, 2006, pp. 307–316.

[9] L. Chen, P. Jiang, and G. Agrawal, "Exploiting recent SIMD architectural advances for irregular applications," in *IEEE/ACM CGO*. IEEE, 2016, pp. 47–58.

[10] W. Liu and B. Vinter, "CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication," in *29th ACM ICS'15*. New York: ACM, 2015, pp. 339–350.

[11] B. Xie, J. Zhan, X. Liu, W. Gao, Z. Jia, X. He, and L. Zhang, "CVR: Efficient vectorization of SPMV on x86 processors," in *IEEE CGO*, 2018.

[12] J. Park, M. Smelyanskiy, K. Vaidyanathan, A. Heinecke, D. D. Kalamkar, M. M. A. Patwary, V. Pirogov, P. Dubey, X. Liu, C. Rosales *et al.*, "Optimizations in a high-performance conjugate gradient benchmark for IA-based multi-and many-core processors," *IJHPCA*, pp. 11–27, 2016.

[13] Q. Zhu, H. Luo, C. Yang, M. Ding, W. Yin, and X. Yuan, "Enabling and scaling the HPCG benchmark on the newest generation Sunway supercomputer with 42 million heterogeneous cores," in *SC'21*, 2021, pp. 1–13.

[14] J. Gurhem, M. Vandromme, M. Tsuji, S. G. Petiton, and M. Sato, "Sequences of Sparse Matrix-Vector Multiplication on Fugaku's A64FX processors," in *CLUSTER*. IEEE, 2021, pp. 751–758.

[15] C. Alappat, G. Hager, O. Schenk, and G. Wellein, "Level-based blocking for sparse matrices: Sparse matrix-power-vector multiplication," *arXiv preprint arXiv:2205.01598*, 2022.

[16] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and e. H. Van der Vost, *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide*. Philadelphia: SIAM, 2000.

[17] G. H. Golub and C. F. V. Loan, *Matrix Computations*, 3rd ed. The Johns Hopkins University Press, Baltimore, MD, 1996.

[18] R. Li, Y. Xi, L. Erlandson, and Y. Saad, "The eigenvalues slicing library (EVSL): Algorithms, implementation, and software," *SIAM Journal on Scientific Computing*, vol. 41, no. 4, pp. C393–C415, 2019.

[19] J. Winkelmann, P. Springer, and E. D. Napoli, "Chase: Chebyshev accelerated subspace iteration eigensolver for sequences of Hermitian eigenvalue problems," *ACM Transactions on Mathematical Software (TOMS)*, pp. 1–34, 2019.

[20] Y. Saad, *Iterative methods for sparse linear systems*. Boston, MA: PWS publishing Company, 1996.

[21] D. S. Watkins, *The matrix eigenvalue problem: GR and Krylov subspace methods*. SIAM, 2007.

[22] E. Chow, A. J. Cleary, and R. D. Falgout, "Design of the hypre preconditioner library," *Object Oriented Methods for Inter-operable Scientific and Engineering Computing*, pp. 106–116, 1999.

[23] T. Iwashita, H. Nakashima, and Y. Takahashi, "Algebraic block multicolor ordering method for parallel multi-threaded sparse triangular solver in ICCG method," in *26th IPDPS*. IEEE, 2012, pp. 474–483.

[24] Phytium, "FT-2000+/64," 2019. [Online]. Available: https://en.wikichip.org/wiki/phytium/feiteng/ft-2000\%2B-64

[25] Huawei, "Kunpeng 920," 2019. [Online]. Available: https://www.hisilicon.com/cn/products/Kunpeng/Huawei-Kunpeng/Huawei-Kunpeng-920

[26] F. Mantovani, M. Garcia-Gasulla, J. Gracia, E. Stafford, F. Banchelli, M. Josep-Fabrego, J. Criado-Ledesma, and M. Nachtmann, "Performance and energy consumption of HPC workloads on a cluster based on ARM ThunderX2 CPU," *Future generation computer systems*, vol. 112, pp. 800–818, 2020.

[27] Intel, "Intel oneAPI Math Kernel Library," 2022. [Online]. Available: https://www.intel.com/content/www/us/en/develop/documentation/get-started-with-mkl-for-dpcpp/top.html

[28] J. L. Greathouse and M. Daga, "Efficient sparse matrix-vector multiplication on GPUs using the CSR storage format," in *SC'14*. IEEE, 2014, pp. 769–780.

[29] A. George, J. W. Liu *et al.*, *Computer solution of large sparse positive definite systems*. Prentice-Hall Englewood Cliffs, NJ, 1981, vol. 134.

[30] J. Park, M. Smelyanskiy, N. Sundaram, and P. Dubey, "Sparsifying synchronization for high-performance shared-memory sparse triangular solver," in *International Supercomputing Conference*. Springer, 2014, pp. 124–140.

[31] M. Naumov, "Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the gpu," *NVIDIA Corp., Westford, MA, USA, Tech. Rep. NVR-2011*, vol. 1, 2011.

[32] T. Iwashita, H. Nakashima, and Y. Takahashi, "Algebraic block multicolor ordering method for parallel multi-threaded sparse triangular solver in ICCG method," in *26th IPDPS*, 2012, pp. 474–483.

[33] T. Iwashita and M. Shimasaki, "Algebraic multicolor ordering for parallelized ICCG solver in finite-element analyses," *IEEE transactions on magnetics*, vol. 38, no. 2, pp. 429–432, 2002.

[34] J. Dongarra and M. A. Heroux, "Toward a new metric for ranking high performance computing systems," Sandia, Tech. Rep. SAND2013-4744, 2013.

[35] A. H. Gebremedhin, D. Nguyen, M. M. A. Patwary, and A. Pothen, "Colpack: Software for graph coloring and related problems in scientific computing," *ACM Transactions on Mathematical Software (TOMS)*, vol. 40, no. 1, pp. 1–31, 2013.

[36] C. Gómez, F. Mantovani, E. Focht, and M. Casas, "Efficiently running SPMV on long vector architectures," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2021, pp. 292–303.

[37] 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 Transactions on Parallel Computing (TOPC)*, vol. 7, no. 3, pp. 1–37, 2020.

[38] T. Gruber and et. al., "LIKWID performance tools," 2019. [Online]. Available: https://github.com/RRZE-HPC/likwid

[39] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, "Efficient sparse matrix-vector multiplication on X86-based many-core processors," in *ICS'13*. New York: ACM, 2013, pp. 273–282.

[40] Y. Niu, L. Zhengyang, M. Dong, Z. Jin, W. Liu, and G. Tan, "TileSpMV: A tiled algorithm for sparse matrix-vector multiplication on GPUs," in *35th IPDPS*. IEEE, 2021, pp. 68–78.

[41] P. Mironowicz, A. Dziekonski, and M. Mrozowski, "A Task-Scheduling Approach for Efficient Sparse Symmetric Matrix-Vector Multiplication on a GPU," *SIAM Journal on Scientific Computing*, 2015.

[42] W. Liu and B. Vinter, "Speculative segmented sum for sparse matrix-vector multiplication on heterogeneous processors," *Parallel Computing*, vol. 49, pp. 179–193, 2015.

[43] D. Merrill and M. Garland, "Merge-based parallel sparse matrix-vector multiplication," in *SC'16*, Salt Lake, 2016.

[44] C. Alappat, J. Laukemann, T. Gruber, G. Hager, G. Wellein, N. Meyer, and T. Wettig, "Performance Modeling of Streaming Kernels and Sparse Matrix-Vector Multiplication on A64FX," in *IEEE/ACM PMBS*. IEEE, 2020, pp. 1–7.

[45] N. Namashivayam, S. Mehta, and P.-C. Yew, "Variable-sized blocks for locality-aware SpMV," in *IEEE/ACM CGO*. IEEE, 2021.

[46] J. Demmel, M. F. Hoemmen, M. Mohiyuddin, K. A. Yelick, J. Demmel, M. Hoemmen, M. Mohiyuddin, and K. Yelick, "Avoiding communication in computing krylov subspaces," UC, Berkeley, Tech. Rep. UCB/EECS-2007-123, 2007.

[47] E. Carson, "Communication-avoiding krylov subspace methods in theory and practice," UC, Berkeley, Tech. Rep. UCB/EECS-2015-179, 2015.

[48] E. Carson, T. Gergelits, and I. Yamazaki, "Mixed precision s-step lanczos and conjugate gradient algorithms," *Numer. Linear Algebra Appl.*, vol. 23, pp. 656–673, 2016.

[49] D. R. Kincaid, J. R. Respess, D. M. Young, and R. R. Grimes, "Algorithm 586: Itpack 2c: A fortran package for solving large sparse linear systems by adaptive accelerated iterative methods," *ACM Transactions on Mathematical Software (TOMS)*, pp. 302–322, 1982.

[50] X. Yu, H. Ma, Z. Qu, J. Fang, and W. Liu, "Numa-aware optimization of sparse matrix-vector multiplication on armv8-based many-core architectures," in *IFIP NPC*. Springer, 2020, pp. 231–242.

[51] J. Park, M. Smelyanskiy, K. Vaidyanathan, A. Heinecke, D. D. Kalamkar, X. Liu, M. M. A. Patwary, Y. Lu, and P. Dubey, "Efficient shared-memory implementation of high-performance conjugate gradient benchmark and its application to unstructured matrices," in *SC'14:*. IEEE, 2014, pp. 945–955.