MAB-SpMV: Towards Cache Line Level Memory Access Balance of SpMV on CPU

Paper ID: 44

Abstract

Sparse Matrix-Vector Multiplication (SpMV) plays a crucial role in scientific computing. However, it often generates severe load imbalance among threads. Most previous load-balancing methods ignored the CPU's cache line-based access characteristics, resulting in limited load-balancing effectiveness.

This paper proposes MAB-SpMV, a load-balancing SpMV based on the Compressed Sparse Row (CSR) format, to realize better load balancing. MAB-SpMV evaluates the access and computing load of CSR-Based SpMV from the cache line level and uses two-dimensional partitioning of the workload based on CSR format to ensure load balancing among threads, which achieves better load balancing compared to previous work.

We compare MAB-SpMV with the CPU-based benchmark SpMV algorithm on 2661 matrices collected from SuiteSparse. The experimental results show that we achieve average speedup ratios of x2.06, x1.69, x5.60, and x3.14 relative to MKL, Merge-Based SpMV, CSR5, and CVR, respectively. In addition, our proposed load-balancing preprocessing method can be easily applied to other SpMVs.

CCS Concepts: • **Computing methodologies → Parallel algorithms**.

Keywords: CPU, SpMV, Workload Balance, Cacheline

ACM Reference Format:
Paper ID: 44. 2024. MAB-SpMV: Towards Cache Line Level Memory Access Balance of SpMV on CPU. In *Proceedings of The 33rd International Symposium on High-Performance Parallel and Distributed Computing (Conference acronym 'XX).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Sparse Matrix-Vector Multiplication (SpMV) plays a pivotal role in various scientific computing applications, including PageRank, graph partitioning and clustering, finite element analysis, and solving systems of linear equations [1, 9, 20, 29]. Unfortunately, SpMV often becomes a performance bottleneck due to its access-intensive and bandwidth-constrained characteristics, which severely restricts the efficiency of the overall computation [15, 19]. Therefore, efficient implementation and optimization of SpMV are crucial for enhancing the efficiency of scientific computing.

Utilizing modern CPUs with multithreaded parallelism presents an opportunity to accelerate SpMV and enhance the overall efficiency of computing. However, the inherently irregular distribution of non-zero elements in sparse matrices potentially leads to significant load imbalances in multithreaded SpMV computing. Sparse matrices are typically stored in specific formats such as Compressed Sparse Row (CSR), Compressed Sparse Column (CSC), and Coordinate (COO), with CSR being the most widely used format, as illustrated in Fig. 1.

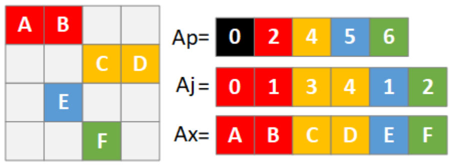

Figure 1. Brief introduction to CSR Format

The Ap array records the offset for the non-zero elements' column indexes in Aj and values in Ax. Specifically, $Ap[i]$ indicates the starting position of the non-zero elements for the i^{th} row in the Aj and Ax arrays. Here, $Aj[j]$ and $Ax[j]$ respectively represent the column index and value of the j^{th} non-zero element. The same with the CSR format, the CSR-Based SpMV is also the most commonly utilized SpMV algorithm supported by many scientific computing libraries such as MKL [33] and CuSparse [31].

However, the standard parallel CSR-Based SpMV allocates workload with row granularity, also known as Row-Splitting, resulting in significant load imbalances. The standard parallel CSR-Based SpMV is illustrated in Algorithm 1. The y in Algorithm 1 refers to the output vector, and the x refers to the dense vector in SpMV.

Previous approaches have sought to mitigate this issue by employing dynamic scheduling of threads, as seen in

Algorithm 1 Standard Parallel CSR-Based SpMV

```
Input: $Ap, Aj, Ax, x$
Output: $y$
1: #program omp parallel for
2: for $i = 0; i < rows; i++$ do
3:     $sum = 0$
4:     for $j = Ap[i]; j < Ap[i+1]; j++$ do
5:         $sum = sum + x[Aj[j]] \times Ax[j]$
6:     end for
7:     $y[i] = sum$
8: end for
```

SpV8 [21], NNZ-Splitting methods like CVR [35] and CSR5 [25], and Hybrid-Splitting methods like Merge-Based SpMV [28], which have demonstrated improved load balancing to a certain extent.

However, coarse-grained workload partitioning, such as Row-Splitting, leads to load imbalances, and overly fine-grained workload partitioning, such as NNZ-Splitting and Hybrid-Splitting, exacerbates the issue due to the CPU's cache line access granularity. Additionally, since data access load constitutes a substantial portion of the total overhead in SpMV, analyzing access at the cache line level presents an opportunity for better load balancing.

This paper analyzes access load for sparse matrices in CSR-Based SpMV, primarily focusing on cache line level accesses. Then, we propose MAB-SpMV, a cache line level load-balancing SpMV based on the CSR format. By traversing the Ap and Aj in CSR format, MAB-SpMV evaluates the access and computing load of CSR-Based SpMV from the cache line level. It counts the cache lines fetched for computing as a part of the SpMV workload. Furthermore, the cache lines with enough valid elements will be considered explicitly for computing load, also a part of the SpMV total workload. After workload evaluation, MAB-SpMV uses two-dimensional partitioning according to the workload based on the CSR format. The partitioning provides each thread's range of rows and non-zero elements to ensure load balancing among threads. Finally, MAB-SpMV achieves better load balancing and impressive performance improvement compared to previous work. Notably, the preprocessing of the MAB-SpMV algorithm requires only a simple workload evaluating step and a fast workload range computing step.

To evaluate the effectiveness of the MAB-SpMV algorithm, we conduct tests using the Intel Xeon Platinum 9242 processor on our dataset, which consists of 2661 matrices collected from the SuiteSparse Sparse Matrix Collection [14]. Experimental results reveal the notable performance of the MAB-SpMV algorithm, achieving a speedup ratio of 2.06x compared to the highly optimized scientific computing library MKL. Furthermore, relative to open-source Benchmark SpMV algorithms Merge-Based SpMV, CSR5, and CVR, the MAB-SpMV algorithm achieves speedup ratios of x1.69, x5.60, and x3.14, respectively.

The contributions of this paper can be summarized as follows:

- **Access Load Analysis:** We critically examine the widely employed load balancing method for SpMV concerning access load. Through experimental analysis, we assess the impact of access load to x on the performance of SpMV and propose a workload evaluation method based on the cache line.
- **MAB-SpMV Algorithm:** We introduce a novel SpMV algorithm, MAB-SpMV, designed to achieve better load balancing with minimal preprocessing time. Notably, MAB-SpMV demonstrates a substantial performance improvement over existing CPU-based SpMV benchmark methods.
- **Implementation and Testing:** MAB-SpMV is implemented on the prevalent Intel Xeon CPU, representing a mainstream computing platform in scientific computing. We conduct extensive testing and analysis on 2661 matrices from the SuiteSparse sparse matrix dataset, achieving x2.06, x1.69, x5.60, and x3.14 average speedup ratios to MKL, Merge-Based SpMV, CSR5, and CVR.

2 STATE-OF-THE-ART SPMV

2.1 Overview

Optimizing SpMV has been widely studied on various architectures [3, 4, 11, 17, 23–26, 35]. Modern CPUs are equipped with numerous cores, and hyper-threading technology effectively doubles the available threads for CPU scheduling [27]. The performance analysis of SpMV [16, 18, 34] points out that load imbalance severely affects the performance of SpMV. As the thread count rises, load balancing among threads in SpMV performance becomes increasingly crucial [28]. Predominantly, load balancing relies on NNZ-splitting [2, 7, 8, 11, 13, 25, 30, 35, 36], yet Merrill et al. [28] introduces Hybrid-splitting techniques for enhanced load distribution. Notably, existing load-balancing methods overlook the associated overhead of memory access. Li et al. [23] proposed a new sparse matrix format, which considers the overhead of accessing x to improve locality on asymmetric multicore processors (AMPs). However, the overhead of accessing other arrays is ignored. Despite extensive literature emphasizing the utilization of SIMD [2, 7, 21, 25, 35], prefetching [35], blocking [10, 12, 30, 36], format selection [5, 6, 22, 32, 38–40], and other strategies to optimize SpMV efficiency, it is essential to recognize that, for SpMV, memory access cost outweighs other factors, and achieving load balance in memory access stands as the pivotal element in enhancing overall SpMV performance.

2.2 Benchmark SpMV on CPU

MKL CSR-Based SpMV [33]: MKL, Intel's Math Kernel Library, employs the CSR format as its default sparse matrix storage format. MKL CSR-Based SpMV is widely acknowledged as the benchmark for SpMV due to its highly optimized performance.

MKL provides an Inspector-Executor preprocessing operation. This operation optimizes SpMV performance through a comprehensive set of actions, including internal format conversions, row-column reordering on sparse matrices, and additional enhancements. Throughout this paper, the term MKL refers to MKL CSR-Based SpMV. When explicitly discussing MKL CSR-Based SpMV with the Inspector-Executor preprocessing operation, we use the term $MKL - Inspector$.

CVR-Based SpMV [35]: CVR employs a sequential addition approach for sparse rows in the matrix, aggregating them into a two-dimensional array with a width equivalent to the SIMD lanes. This method optimizes the utilization of SIMD lanes by carefully timing and positioning write-backs, achieving a notably high SIMD channel efficiency.

CVR adopts the NNZ-Splitting method to achieve load balancing, evenly distributing non-zero elements among multiple threads. Additionally, CVR implements a $Steal$ policy, strategically allocating non-zero elements in SIMD lanes. This allocation prioritizes channels with fewer computational tasks, ensuring a balanced workload distribution and enhancing overall computational efficiency.

CSR5-Based SpMV [25]: CSR5 implements a reorganization of the non-zero elements within the matrix, organizing them into discrete units known as $Tiles$. Each thread is responsible for computing one $Tile$ at a time. A $Tile$ consists of $\omega \times \sigma$ non-zero elements, their corresponding column coordinates, and a descriptor. The descriptor contains essential information about the flags marking the first element in tile, empty-row, segment-sum, and write-back indexes.

By converting non-zero elements into these fine-grained $Tiles$, CSR5 achieves an effect akin to NNZ-Splitting. This strategy allows for more efficient computation and management of non-zero elements, enhancing the overall performance and load-balancing capabilities of SpMV.

Merge-Based SpMV [28]: The Hybrid-Splitting approach incorporates the write-back overhead to the output vector y into the load balancing considerations. Unlike the NNZ-Splitting method, which only considers the sum of non-zero elements (balancing access to Aj and Ax), the Hybrid-Splitting method broadens its scope. Besides Aj and Ax, it includes Ap and the output vector y in its load balancing calculations.

Specifically, the total load is determined by considering not only the non-zero elements (Aj and Ax) but also the row indices (Ap and y). This more comprehensive load-balancing strategy ensures a more equitable distribution of computational tasks among threads, contributing to improved load balancing compared to methods solely relying on non-zero elements.

3 MOTIVATION

3.1 SpMV Workload Imbalance Analysis

In standard CSR-Based SpMV, workloads are assigned to threads based on the granularity of the row, which is also known as Row-Splitting. However, this method of load allocation results in severe load imbalance. As indicated in Table 1, there is a substantial discrepancy in the number of non-zero elements allocated to each thread.

Table 1. NNZ assigned to threads with Row-Splitting

Name	Rows	NNZ			
		Avg	Min	Max	Max / Min
us04	163	3099	36	13277	368.81
lp†	10,280	14667	2459	175994	71.57
brain†	27,607	1868	1430	15228	10.65
vsp†	39,668	3956	2820	21220	7.52
dc1	116,835	7983	6015	120242	19.99
ins2	309,412	28661	21602	331012	15.32
ASIC†	682,862	40331	31849	427784	13.43
web†	1,000,005	32349	28828	37644	1.31

† means abbreviation of matrix name.

For example, in the lp_osa_60 matrix, each thread, on average, is tasked with computing 14,667 non-zero elements. However, the most burdened thread must handle 175,994 non-zero elements, while the least burdened thread must only take 2,459 non-zero elements—this discrepancy of 71.57 times results in severe load imbalance.

Dynamic scheduling of threads sometimes performs better load balancing concerning row granularity. However, the performance gains from this approach are limited. In Fig. 2, certain cases as $us04$, $brainpc2$, $vsp\_south31\_slptsk$, and $webbase-1M$ show that the dynamic allocation method lags behind the standard CSR-Based SpMV using the static allocation method. This is attributed to increased scheduling overhead with dynamic allocation [28]. Therefore, preprocessing-based thread workload division is more feasible for better load balancing in CSR-Based SpMV.

The NNZ-Splitting regards NNZ as the SpMV workload and is widely used in preprocessing, aiming to allocate an equal number of non-zero elements to each thread. However, it overlooks that CPU access granularity is a cache line. This oversight may lead to a significantly unbalanced load on memory accesses, diminishing the effectiveness of load balancing.

Hybrid-Splitting regards $NNZ + Rows$ as the SpMV workload to balance access to Ap, Aj, and Ax. But it also neglects the access load for x. Table 2 shows the number of cache lines

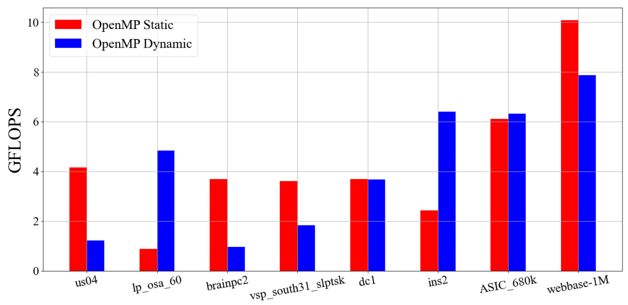

Figure 2. Parallel CSR-Based SpMV with Static/Dynamic workload distribution method

each thread needs to fetch from x when using NNZ-Splitting or Hybrid-Splitting for load balance. This table likely demonstrates the potential imbalance and inefficiencies in memory accesses of x.Notice that in the Hybrid-Splitting, we use $x+y$ to denote the workload assigned to each thread and *nnz* in NNZ-Splitting.

When using the NNZ-Splitting method, the total number of cache lines fetched from x varies tremendously even though each thread is assigned to the same number of non-zero elements. A maximum difference of 7.17 times for number of cache lines fetched from x among the threads in $TSOPF\_FS\_b300\_c1$ and 7.52 times in $nxp1$. When using the Hybrid-Splitting method, the number of cache lines fetched from x between threads decreases, and a better load balancing is achieved, but it still suffers from load imbalance from reading x, such as a maximum difference of 5.35 times between threads for the total number of cache lines fetched from x in $Raj1$ and even a more significant difference in $email-EuAll$. Although data reuse exists between different threads by sharing the cache when performing multithreaded parallel SpMV, it is not practical in reducing the imbalance of accesses to x in most cases due to the poor localization of sparse matrix non-zero elements. There are two reasons for the unbalanced load on x access: 1) overly fine-grained workload partitioning; 2) irregular accesses to x. Hence, the access load of all arrays from the cache line level needs to be evaluated to achieve better load balancing.

3.2 Cache Line Wise Memory Access

In contemporary CPUs, a cache line typically has a size of 64 bytes, accommodating 16 valid elements of type *int* or 8 of type *double*. Unless explicitly stated otherwise, the default floating-point type discussed in this paper is *double*.

Each access is consecutive for arrays Ap, Aj, and Ax. However, accesses are often discontinuous when accessing x. Figure 3 illustrates that $Cacheline0$ contains 3 valid elements, $Cacheline1$ has 5, but $Cacheline2$ only has 1, and $Cacheline3$ contains 7. The valid elements refer to the elements used for multiplication after reading from x.

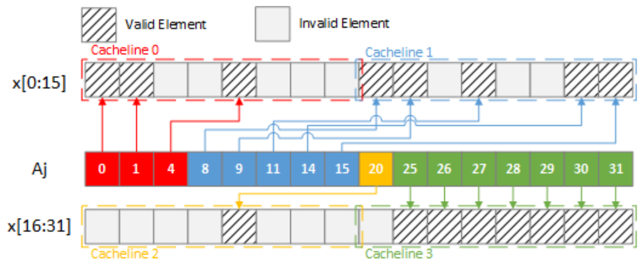

Figure 3. Data usage situation in cache line

This discrepancy leads to the fact that if the number of non-zero elements that need to be computed is NNZ, the total number of cache lines that need to be fetched from x varies from NNZ to $8 \times NNZ$ depending on the number of valid elements per cache line.

The distribution of non-zero elements in each matrix was analyzed, as depicted in Figure 4. The category "Non-zeros in Cacheline (nnz=1)" signifies cases where a cache line contains only one valid element. On average, this scenario covers 43.45% of the non-zero elements in each matrix. The remaining 56.55% of non-zero elements are distributed in the cache line with "Non-zeros in Cacheline (nnz>1)".

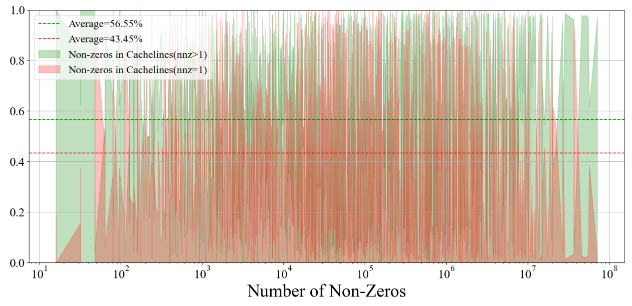

Figure 4. Distribution of non-zero elements in cache line

The results illustrate that cache lines encompass various numbers of elements that need to be accessed. Consequently, even if each thread is assigned an equal number of non-zero elements, each thread's total amount of data read from x exhibits significant variability. This highlights the impact of diverse data requirements within cache lines on the total amount of data accessed by each thread during SpMV computations.

3.3 Impact of Memory Access to $\underline{x}$ on SpMV

SpMV is an access-intensive, bandwidth-constrained algorithm; accesses occupy the main part of the total overhead, and unbalanced accesses to the x seriously affect the performance.

To assess the impact of the number of visits to x on performance, we conduct simulations with a single thread computing a row in a sparse matrix. This is motivated by the

Table 2. Fetched cache lines in x with NNZ-Splitting and Hybrid-Splitting

Name	NNZ-Splitting					Hybrid-Splitting				
	nnz	Fetched cache lines in x				$x + y$	Fetched cache lines in x			
		Avg	Min	Max	Max / Min		Avg	Min	Max	Max / Min
bloweybq	730	273	91	306	3.36	834	273	104	299	2.88
ABACUS_shell_md	2276	1803	750	2078	2.77	2520	1803	768	2078	2.71
psmigr_2	5626	3094	1875	3746	2.00	5659	3094	1880	3718	1.98
email-EuAll	4376	4213	3859	4347	1.13	7139	4213	2920	6604	2.26
Raj1	13568	8588	1725	11317	6.56	16315	8588	2067	11056	5.35
nxp1	27669	21486	3465	26040	7.52	31988	21486	4005	25953	6.48
TSOPF_FS_b300_c1	45835	6976	5743	41574	7.24	46139	6976	5783	36012	6.23

unpredictable behavior that arises with multiple threads, where random sharing of cache data among threads can introduce significant random performance. Simulating a single thread enables a more stable performance measurement and facilitates a focused analysis of performance factors.

We generate random Aj values and sort them in ascending order to emulate the irregular access pattern of x in CSR format. The randomly generated x length is sufficiently long, and the sparsity is maintained at least 1/128 in random. Although sparsity varies significantly across different sparse matrices, ensuring a reasonably small sparsity mitigates the impact of random accesses to varying sparsities on the final efficiency. This simulation framework enables a controlled assessment of the influence of x access patterns on performance.

The performance of sparse row computation with varying access to x for different NNZ is shown in Table 3.

Table 3. Time overhead of accessing x

NNZ	$MA(x)$	$MA(Total)$	$Time$	$Reduction$
32768	2097152	2490368	0.094665	-
32768	1048576	1441792	0.049678	47.52%
32768	524288	917504	0.0476069	49.71%
32768	262144	655360	0.0412405	56.44%
131072	8388608	9961472	0.387939	-
131072	4194304	5767168	0.224713	42.08%
131072	2097152	3670016	0.206519	46.77%
131072	1048576	2621440	0.172722	55.48%
262144	33554432	39845888	6.18924	-
262144	16777216	23068672	3.93646	36.40%
262144	8388608	14680064	3.28026	47.00%
262144	4194304	10485760	2.61579	57.74%

Table 3 illustrates the time overhead while computing a fixed NNZ. When NNZ is fixed, $FLOPs$, accesses to Aj and Ax remain constant. However, the bytes read from x (denoted as $MA(x)$) vary based on the average number of valid elements in each cache line.

The smaller the number of valid elements in each cache line on average, the larger the number of accesses that need to be performed to compute the same number of NNZ, and the larger the number of accesses to x. For instance, when average number of valid elements in each cache line is 1, 84.21% of the total memory accesses are to x, as shown in Table Table 3 line 1, 5, 9. Furthermore, when the average number is 2, this percentage reduces to 72.72%, as shown in Table 3 line 2, 6, 10. The time overhead reductions are 47.52%, 42.08%, 36.40%, as shown in the last column.

Simulations for different NNZ values demonstrate a consistent reduction in time overhead as $MA(x)$ decreases. Similarly, when there are 2, 4, and 8 valid elements per cache line, the time reduces by an average of 42.00%, 47.83%, and 56.55% compared with one valid element per cache line. These results underscore the substantial impact of accessing x on the efficiency of SpMV, mainly when the average number of valid elements in each cache line is 1.

3.4 Real World Matrices

Figure 5 displays the average number of valid elements contained in cache lines of matrices in the dataset. The metric **Average NNZ per Cacheline** represents the average number of valid elements in each cache line.

From Fig. 5, we can see that 66.27% of the matrices exhibit **Average NNZ per Cacheline** values between 1-2, 82.67% between 1-3, and 90.39% between 1-4. This indicates that, for most matrices, accessing x constitutes a significant portion of the overhead when performing SpMV. Specifically, for 90.39% of matrices, access to x accounts for at least 57.14% of the total access load, with 66.27% of the matrices having an even higher percentage (at least 72.72%).

These findings underscore the critical importance of load balancing for memory accesses, emphasizing the need for efficient strategies to distribute the access load evenly across threads to optimize SpMV performance.

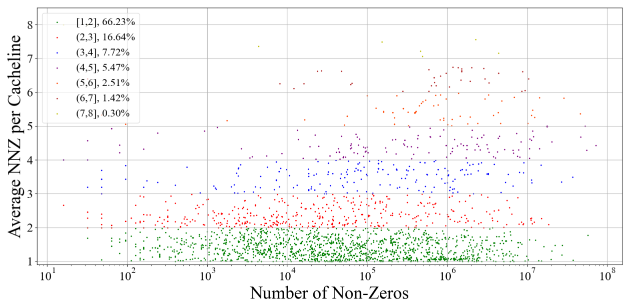

Figure 5. Average valid elements per cache line

4 METHOD

4.1 Memory Access Balance-Splitting

Memory Access Balance-Splitting is illustrated in Fig. 6

4.2 MAB-SpMV Workload Evaluation

The SpMV workload can be described as:

$$Workload = \alpha C(nnz) + \beta MA(Aj + Ax + Ap + y + x)$$

$C(nnz)$ corresponds to one floating-point multiply-add operation on nnz non-zero elements, while $MA(Aj+Ax+Ap+y+x)$ indicates the number of accesses required to Aj, Ax, Ap, y, and x. The overheads of the floating-point operation and the access operation are denoted by α and β, respectively. In our approach, we consolidate the overhead of writing back to y with the overhead of accessing other arrays, aligning with the principles of Merge-Based SpMV [28]. It's essential to note that Merge-Based SpMV assumes equality between computing overhead and access overhead (i.e., $\alpha = \beta$), but in practical scenarios, computing overhead is often less than access overhead (i.e., $\alpha < \beta$).

We quantify the access workload by using the total number of cache lines accessed in Ap, Aj, Ax, y, and x, converting the load evaluation to cache line units. Algorithm 2 shows the calculation of the total number of cache lines.

We use the number of accessed cache lines to represent the access workload. *workload* is for storing evaluated workload while traversing the sparse matrix in a CSR-Based SpMV manner. The cache line count is tallied per row as threads read data in row order. In lines 14, 15, and 16, the variables recording the addresses are set to -1 at the beginning of each row. Given that a cache line can accommodate 16 *int* or 8 *double* elements, the divisors in lines 6, 10, 18, 22, and 26 of the algorithm are set to 16, 8, 8, 8, 8, and 16, respectively. If the computation of the current element necessitates access to a new cache line, one is added to the *cache_lines*. In line 30, the current generated load is recorded in $workload[j+1]$. Finally, the total load generated from computing the i^th non-zero element to the j^th non-zero element is $workload[i] - workload[j]$. If NNZ non-zero elements are in the sparse

Algorithm 2 Computing Workload

```
Input: $Ap, Aj$
Output: $workload$
 1: $workload[0] = 0$
 2: $cache\_lines = 0$
 3: for $i = 0; i < rows; i++$ do
 4:     if $last\_Ap\_cacheline\_addr \neq i/16$ then
 5:         $last\_Ap\_cacheline\_addr = i/16$
 6:         $cache\_lines++;$
 7:     end if
 8:     if $last\_y\_cacheline\_addr \neq i/8$ then
 9:         $last\_y\_cacheline\_addr = i/8$
10:         $cache\_lines++;$
11:     end if
12:     for $j = Ap[i]; j < Ap[i+1]; j++$ do
13:         if $last\_x\_cache\_addr \neq Aj[j]/8$ then
14:             $last\_x\_cache\_addr = Aj[j]/8$
15:             $cache\_lines++;$
16:         end if
17:         if $last\_Ax\_cacheline\_addr \neq j/8$ then
18:             $last\_Ax\_cacheline\_addr = j/8$
19:             $cache\_lines++;$
20:         end if
21:         if $last\_Aj\_cacheline\_addr \neq j/16$ then
22:             $last\_Aj\_cache\_addr = j/16$
23:             $cache\_lines++;$
24:         end if
25:         $workload[j+1] = cache\_lines$
26:     end for
27: end for
```

matrix, then $workload[nnz-1]$ captures the total load for performing SpMV.

4.3 MAB-SpMV Workload Partitioning

We use $workload[nnz-1]$ as the total workload for threads. The workload range of each thread is calculated as shown in Algorithm 3.

Lines 1-3 of Algorithm 3 calculate the workload of each thread (wl_per_thread), so for thread tid, the range of its workload is $[wl\_per\_thread \times tid, wl\_per\_thread \times (tid+1)]$. It can be found that the upper bound of thread tid is equal to the lower bound of thread $tid+1$. Therefore, we only need to calculate the upper bound of the workload of each thread and store its corresponding offset (row_end) in Ap and offset (nnz_end) in Aj and Ax as shown in lines 8-12. Eventually $thread\_nnz\_bound[tid]$ and $thread\_nnz\_bound[tid+1]$ denote the offsets in Aj and Ax of the first and last non-zero elements that the thread needs to compute, respectively, and $thread\_row\_bound[tid]$ and $thread\_row\_bound[tid+1]$ respectively denote the offsets in Ap of the first and last non-zero elements that the thread needs to compute. The binary

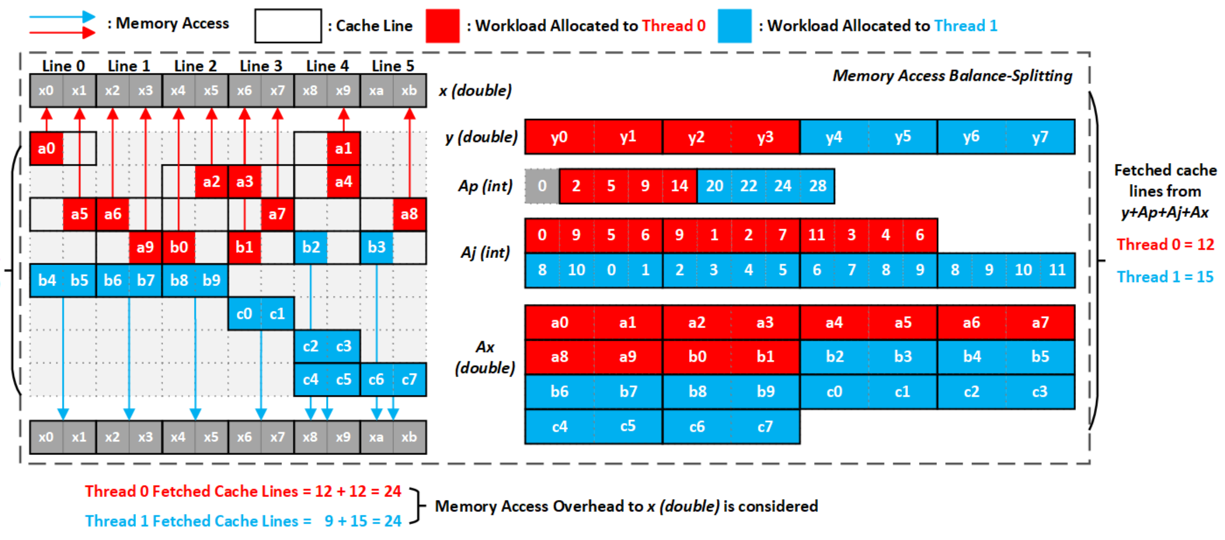
: Memory Access
: Cache Line
: Workload Allocated to Thread 0
: Workload Allocated to Thread 1
Memory Access Balance-Splitting
x (double)
y (double)
Ap (int)
Aj (int)
Ax (double)
Fetched cache lines from x
Thread 0 = 12
Thread 1 = 9
Fetched cache lines from y+Ap+Aj+Ax
Thread 0 = 12
Thread 1 = 15
Thread 0 Fetched Cache Lines = 12 + 12 = 24
Thread 1 Fetched Cache Lines = 9 + 15 = 24
Memory Access Overhead to x (double) is considered

Figure 6. An example of Memory Access Balance-Splitting.

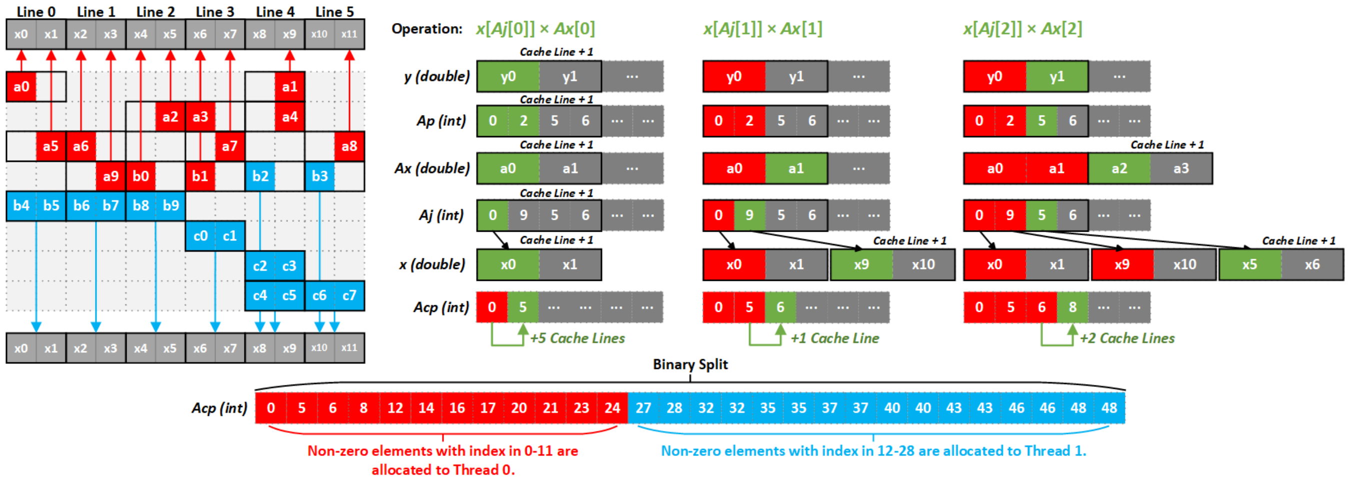

Figure 7. An example of obtaining **Acp**.

search Algorithm used in Algorithm 3 is shown in Algorithm 4.

4.4 MAB-SpMV Implementation

After realizing the workload range of each thread, the implementation of MAB-SpMV is illustrated in Algorithm 5. The input arrays *Ap*, *Aj*, and *Ax* are from the CSR format. *thread_nnz_bound* and *thread_row_bound*, which store the workload range of each thread, are calculated in Algorithm 3. Since the granularity of writing back to the y is in rows and different threads may compute elements of the same row, reduction is needed. Since we split the load continuously, only threads with adjacent IDs need reduction. We use *tail_sum* and *reduce_row* to keep track of the *partial_sum* and the offset (aka row index) to y. Each thread has no write-back conflict with other threads until computing the last row. Therefore, direct write-back to y is available, as shown in lines 10-15. When processing the partial of the last row, MAB-SpMV uses $tail\_sum[tid]$ to store the $partial\_sum$ and $reduce\_row[tid]$ for the row index, as shown in lines 17-22. Finally, the main thread performs the reduction to obtain an entire result, as shown in lines 24-25 of the algorithm.

The similarity between MAB-SpMV and Merge-Based SpMV is summarized below: 1) partitioning the matrix in two-dimensional, $nnz + rows$ in Merge-Based SpMV and $access + computing$ load in MAB-SpMV. 2) perform SpMV with CSR format. The difference is summarized as below: 1) Merge-Based SpMV combines the non-zero elements and

Algorithm 3 Calculating Workload Range

```
Input: $workload, Ap, nnz, rows$
Output: $thread\_nnz\_bound, thread\_row\_bound$
1: $total\_workload = workload[nnz - 1]$
2: $thread\_num = omp\_get\_max\_threads()$
3: $wl\_per\_thread = (total\_workload + thread\_num - 1)/thread\_num$
4: $thread\_nnz\_bound[0] = 0$
5: $thread\_row\_bound[0] = 0$
6: #program omp parallel for
7: for $tid = 0; tid < thread\_num; tid++$ do
8:     $end = wl\_per\_thread \times (tid + 1)$
9:     $nnz\_end = Binary\_Search(workload + 1, nnz, end)$
10:    $thread\_nnz\_bound[tid + 1] = nnz\_end$
11:    $row\_end = Binary\_Search(Ap + 1, rows, nnz\_end)$
12:    $thread\_row\_bound[tid + 1] = row\_end$
13: end for
```

Algorithm 4 Binary Search

```
Input: $array, length, target$
Output: $begin$
1: $begin = 0$
2: $end = length$
3: while $begin < end$ do
4:     $middle = (begin + end)/2$
5:     if $array[middle] \leq target$ then
6:         $begin = middle + 1$
7:     else
8:         $end = middle$
9:     end if
10: end while
```

Algorithm 5 MAB-Based SpMV

```
Input: $Ap, Aj, Ax, thread\_nnz\_bound, thread\_row\_bound$
Output: $y$
1: $thread\_num = omp\_get\_max\_threads()$
2: $tail\_sum[thread\_num]$
3: $reduce\_row[thread\_num]$
4: #program omp parallel for
5: for $tid = 0; tid <; tid++$ do
6:     $s\_nnz = thread\_nnz\_bound[tid]$
7:     $e\_nnz = thread\_nnz\_bound[tid + 1]$
8:     $s\_row = thread\_row\_bound[tid]$
9:     $s\_row = thread\_row\_bound[tid + 1]$
10:    for $s\_row < e\_row; s\_row++$ do
11:        $sum = 0$
12:        for $s\_nnz < Ap[s\_row + 1]; s\_nnz++$ do
13:            $sum+ = x[Aj[s\_nnz]] \times Ax[s\_nnz]$
14:        end for
15:        $y[s\_row] = sum$
16:    end for
17:    $partial\_sum = 0$
18:    for $s\_nnz < end\_nnz; s\_nnz++$ do
19:        $tail\_sum+ = x[Aj[s\_nnz]] \times Ax[s\_nnz]$
20:    end for
21:    $tail\_sum[tid] = partial\_sum$
22:    $reduce\_row[tid] = end\_row$
23: end for
24: for $tid = 0; tid < thread\_num; tid++$ do
25:    $y[reduce\_row[tid]]+ = tail\_sum[tid]$
26: end for
```

rows for load allocation, while MAB-SpMV evaluates the access load and computing load of all arrays used in CSR-Based SpMV, x, and y from the cache line level. 2) Merge-Based SpMV performs workload boundary computing during each iteration of SpMV, while MAB-SpMV does it as a preprocessing. Though the efficiency of the binary search is remarkable, thousands of iterations will also accumulate much overhead. Therefore, we choose to evaluate workload and calculate workload range in the preprocessing, which is more friendly to iterative SpMV.

5 EVALUATION

5.1 Experimental setup

Our approach is based on the CPU, the processor used is an Intel high performance processor Xeon Platinum 9242, which can be invoked with up to 96 threads, the main parameters of this processor are shown in Table 4.

Our experimental data contains 2661 matrices obtained from SuiteSparse, accounting for 92% of the total 2893 sparse matrices in the dataset, and covering domains including linear programming problem, combinatorial problem, computational fluid dynamics problem, undirected weighted graph, and 85 other areas.

We compare MAB-SpMV to today's SpMV Benchmarks on CPU, including MKL and MKL-Inspector. In addition, we also

Table 4. System characteristics.

Component	Characteristics
CPU	①Intel Xeon Platinum 9242 @ 2.30 GHz 48 cores, 96 threads, support AVX512 ②AMD EPYC 7542 @ 2.90 GHz 32 cores, 64 threads, support AVX2
Cache	①64KB L1 and 1MB L2 per core, 71.5MB shared LLC ②96KB L1 and 512KB L2 per core, 128MB shared LLC
Memory	①384GB, DDR4-2933 ②256GB, DDR4-3200
OS	①CentOS Linux release 7.9.2009 ②CentOS Linux release 7.8.2003

compare with open-source Benchmark SpMV algorithms CSR5, CVR, and Merge-Based SpMV. We obtain source codes from the authors' open-source code repositories. As for CSR5, the AVX512 version is chosen for comparison. All codes were compiled using icpx -O3 -qopenmp -xCORE-AVX512.

In this paper, we focus on the aspect of access load balancing. While acknowledging the existence of other optimizations for SpMV on CPU platforms, including matrix chunking, format conversion, SIMD instructions, and data prefetch instructions, which are instrumental for enhancing performance, it is noteworthy that the predominant bottleneck in SpMV often stems from data reading. As a result, our primary emphasis is on assessing the performance gains derived specifically from the implementation of access load balancing.

We conduct 1000 rounds of SpMV on each matrix, collected from SuiteSparse Matrix Collection [14], using different SpMV algorithms and taking the average value to evaluate the performance of SpMV. To standardize the floating point performance calculation method, we calculate the floating point performance only considering the number of non-zero elements (nnz). $FLOPS$ can be represented as:

$$FLOPS = \frac{2 \times nnz}{time}$$

Similarly bandwidth is also considered only for non-zero elements, then the $Bandwidth$ is:

$$Bandwidth = \frac{sizeof(FloatType) \times nnz}{time}$$

$Bandwidth$ and $FLOPS$ differ only by a factor, so we only show performance in floating-point operands per second in our experimental results.

5.2 Performance comparison

The detailed speedup results of MAB-SpMV compared to CSR5, CVR, Merge-Based SpMV, MKL, MKL-Inspector, and Parallel CSR are shown in Table 5.

Figure 8 shows the test results on 2661 matrices. Parts of the results in matrices with a small number of NNZ are also shown in the bottom left of each figure for comparison. As can be seen in Table 5 and Figure 8, the MAB-SpMV shows excellent performance in both small and large matrices relative to the CSR5, CVR, and CSR. Compared with the parallel CSR-Based SpMV, our MAB-SpMV realizes an average speedup ratio of 3.31. As a better load-balanced SpMV, MAB-SpMV achieves average speedup ratios of up to 5.60 and 3.14 times concerning CSR5 and CVR, which use NNZ-Splitting for load balancing. Though applying SIMD, CSR5 is even slower than parallel CSR-Based SpMV. Applying SIMD needs more instructions for complicated controlling computing procedures [34, 37]. Besides, the parameter σ in CSR5 must be carefully adjusted in different architectures. On the contrary, load balancing is a universal technique for speeding up SpMV.

We select eight representative matrices for further comparison in Table 6.

Our MAB-SpMV achieves an average 2.06x speedup ratio relative to MKL and 2.08x to MKL-Inspector. MKL uses padding to minimize the number of instructions and algorithm complexity while making better use of SIMD [34]. Based on MKL, MKL-Inspector applies other techniques to optimize SpMV. Though many optimizations are involved, the load imbalance restricts its performance. $ins2$ is a highly irregular matrix, and MAB-SpMV achieves a speedup of 13.22x over MKL-Inspector and 5.62x over MKL, much higher than the average speedup ratios, respectively. Our MAB-SpMV achieves an average speedup ratio of 1.69x against Merge-Based SpMV, which demonstrates the performance impact of uneven load on x accesses, and MAB-SpMV is a better load balancing SpMV. Though MAB-SpMV shows impressive performance, it is not always the optimal load balanced, such as in $ins2$, $cont1\_l$, and $TSOPF\_FS\_b300\_c1$. Merge-Based SpMV implements load balancing of accesses to Ap, Aj, Ax, and y, making a better load balance than Row-Splitting and NNZ-Splitting. However, Merge-Based SpMV does not consider the load imbalance caused by cache line granularity accesses to x, so the load balancing on partial matrices is not optimal. $boyd1$, $IG5-14$, and $watson\_1$ are highly irregular sparse matrices, and our MAB-SpMV achieves significant speedups over MKL, MKL-Inspector, and Merge-Based SpMV due to better load balance. $apache2$ is a large matrix with almost non-zero elements distributed on the diagonal. Although $apache2$ has a regular distribution of non-zero elements, the matrix size is too large, resulting in poor data localization. Therefore, the main overhead is to access data, so the access-balanced MAB-SpMV outperforms others.

5.3 Preprocessing Overhead

As shown in Fig. 9, the preprocessing time of MAB-SpMV surpasses that of CVR and CSR5 when dealing with a small number of non-zero elements ($nnz < 10^6$).

In our experiments, a parallel algorithm was employed for prefix sum computation to obtain $workload$, necessitating traverse Ap, Aj, and $workload$ arrays twice, resulting in a notable number of memory accesses. Across the entire dataset, the average preprocessing time of MAB-SpMV is 312% of CSR5, 331% of CVR, and only 41% of MKL-Inspector. However, as the quantity of non-zero elements increases ($nnz > 10^6$), the preprocessing cost of MAB-SpMV becomes 151% of CSR5, 76% of CVR, and 33% of MKL-Inspector on average. In practical applications, preprocessing overhead constitutes only a tiny fraction of the total. Although the average preprocessing time of MAB-SpMV is 3.12 times that of CSR5 and 3.31 times that of CVR, it requires only an average of 11.01 and 10.34 SpMVs, respectively, to amortize the preprocessing overhead to be faster than CSR5 and CVR.

Table 5. MAB-SpMV speedup over other benchmark SpMVs

	Algorithms	CSR5	CVR	Merge-Based	MKL	MKL-Inspector	Parallel CSR
Speedup	Total Matrices	5.6	3.14	1.69	2.06	2.08	3.31
	Large Matrices ($NNZ > 10^5$)	5.41	2.94	1.54	2.1	1.83	3.27
	Small Matrices ($NNZ \leq 10^5$)	5.72	3.28	1.78	2.03	2.25	3.34
Cases	Better*/Total	2658/2661	2391/2416†	2514/2661	2581/2661	2471/2661	2638/2661

† CVR-SpMV in some matrices will raise segment fault, these matrices are not considered.
∗ Better means matrices in which MAB-SpMV outperform the compared SpMV.

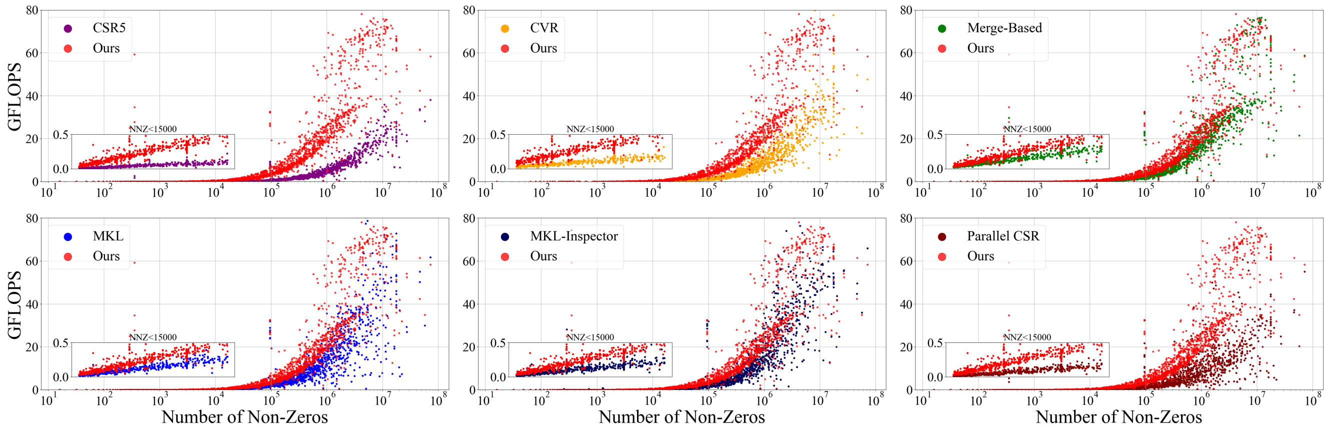

Figure 8. MAB-SpMV **vs.** other benchmark SpMVs

Table 6. Speedup Ratio of 8 Representative Matrices

Matrix	MKL-I†	MKL	Merge†
apache2	2.29	11.76	4.11
IG5-14	6.88	6.66	4.84
cont1_l	0.45	10.89	0.91
ins2	13.21	5.62	0.40
boyd1	3.12	11.27	1.29
watson_1	4.36	4.47	2.33
fpga_dcop_33	6.54	5.01	2.09
TSOPF_FS_b300_c1	0.33	8.87	0.79

† MKL-I refers to MKL-Inspector and Merge refers to Merge-Based SpMV.

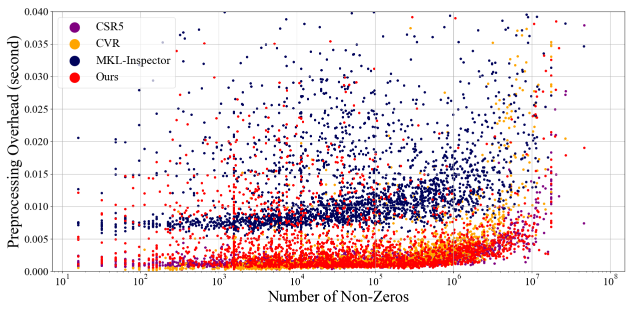

Figure 9. Preprocessing time compared with CSR5, CVR, MKL-Inspector

6 CONCLUSION

Sparse Matrix-Vector Multiplication (SpMV) holds significant importance in scientific computing. However, the irregular distribution of non-zero elements in sparse matrices introduces a substantial load imbalance across threads. Prior studies often overlook the access load imbalance resulting from the significant overhead of irregular accesses to x. Our detailed experimental analysis scrutinizes the influence of x accesses on SpMV performance, considering CPU platform access granularity characteristics. Our findings emphasize that x accesses are pivotal in determining SpMV performance. Addressing the challenge of load balancing of memory access, we propose MAB-SpMV, specifically designed to achieve memory access balance. Experimental results validate that achieving load balance in memory access significantly enhances SpMV performance.

Our experiments, conducted on 2661 sparse matrices, showcase the superiority of our MAB-SpMV. It achieves average speedup ratios of 2.06x, 2.08x, 1.69x, 3.14x, and 5.60x compared to MKL, MKL-Inspector, Merge-Based SpMV, CVR, and CSR5. MAB-SpMV incurs only a minimal preprocessing overhead, underscoring its efficiency and applicability in real-world scenarios.

References

[1] Abal-Kassim Cheik Ahamed and Frederic Magoules. 2012. Iterative methods for sparse linear systems on graphics processing unit. In 2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems. IEEE, 836–842.

[2] Mohammad Almasri and Walid Abu-Sufah. 2020. CCF: An efficient SpMV storage format for AVX512 platforms. Parallel Comput. 100 (2020), 102710.

[3] Arash Ashari, Naser Sedaghati, John Eisenlohr, Srinivasan Parthasarath, and P Sadayappan. 2014. Fast sparse matrix-vector multiplication on GPUs for graph applications. In SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE, 781–792.

[4] Nathan Bell and Michael Garland. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In Proceedings of the conference on high performance computing networking, storage and analysis. 1–11.

[5] Akrem Benatia, Weixing Ji, Yizhuo Wang, and Feng Shi. 2016. Sparse matrix format selection with multiclass SVM for SpMV on GPU. In 2016 45th International Conference on Parallel Processing (ICPP). IEEE, 496–505.

[6] Akrem Benatia, Weixing Ji, Yizhuo Wang, and Feng Shi. 2016. Sparse matrix format selection with multiclass SVM for SpMV on GPU. In 2016 45th International Conference on Parallel Processing (ICPP). IEEE, 496–505.

[7] Haodong Bian, Jianqiang Huang, Runting Dong, Lingbin Liu, and Xiaoying Wang. 2020. CSR2: a new format for SIMD-accelerated SpMV. In 2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID). IEEE, 350–359.

[8] Haodong Bian, Jianqiang Huang, Lingbin Liu, Dongqiang Huang, and Xiaoying Wang. 2021. ALBUS: A method for efficiently processing SpMV using SIMD and Load balancing. Future Generation Computer Systems 116 (2021), 371–392.

[9] Sergey Brin and Lawrence Page. 1998. The anatomy of a large-scale hypertextual web search engine. Computer networks and ISDN systems 30, 1-7 (1998), 107–117.

[10] Aydin Buluç, Jeremy T Fineman, Matteo Frigo, John R Gilbert, and Charles E Leiserson. 2009. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures. 233–244.

[11] Genshen Chu, Yuanjie He, Lingyu Dong, Zhezhao Ding, Dandan Chen, He Bai, Xuesong Wang, and Changjun Hu. 2023. Efficient Algorithm Design of Optimizing SpMV on GPU. In Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing. 115–128.

[12] Huanyu Cui, Nianbin Wang, Yuhua Wang, Qilong Han, and Yuezhu Xu. 2022. An effective SPMV based on block strategy and hybrid compression on GPU. The Journal of Supercomputing (2022), 1–22.

[13] Steven Dalton, Luke Olson, and Nathan Bell. 2015. Optimizing sparse matrix—matrix multiplication for the gpu. ACM Transactions on Mathematical Software (TOMS) 41, 4 (2015), 1–20.

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

[15] A Dziekonski, M Rewienski, Piotr Sypek, A Lamecki, and Michał Mrozowski. 2017. GPU-accelerated LOBPCG method with inexact null-space filtering for solving generalized eigenvalue problems in computational electromagnetics analysis with higher-order FEM. Communications in Computational Physics 22, 4 (2017), 997–1014.

[16] Athena Elafrou, Georgios Goumas, and Nectarios Koziris. 2017. Performance analysis and optimization of sparse matrix-vector multiplication on modern multi-and many-core processors. In 2017 46th International Conference on Parallel Processing (ICPP). IEEE, 292–301.

[17] Salvatore Filippone, Valeria Cardellini, Davide Barbieri, and Alessandro Fanfarillo. 2017. Sparse matrix-vector multiplication on GPGPUs. ACM Transactions on Mathematical Software (TOMS) 43, 4 (2017), 1–49.

[18] Georgios Goumas, Kornilios Kourtis, Nikos Anastopoulos, Vasileios Karakasis, and Nectarios Koziris. 2009. Performance evaluation of the sparse matrix-vector multiplication on modern architectures. The Journal of Supercomputing 50 (2009), 36–77.

[19] Akira Imakura and Tetsuya Sakurai. 2017. Block Krylov-type complex moment-based eigensolvers for solving generalized eigenvalue problems. Numerical Algorithms 75 (2017), 413–433.

[20] Jeremy Kepner, David Bader, Aydın Buluç, John Gilbert, Timothy Mattson, and Henning Meyerhenke. 2015. Graphs, matrices, and the GraphBLAS: Seven good reasons. Procedia Computer Science 51 (2015), 2453–2462.

[21] Chenyang Li, Tian Xia, Wenzhe Zhao, Nanning Zheng, and Pengju Ren. 2021. SpV8: Pursuing optimal vectorization and regular computation pattern in SpMV. In 2021 58th ACM/IEEE Design Automation Conference (DAC). IEEE, 661–666.

[22] Jiajia Li, Guangming Tan, Mingyu Chen, and Ninghui Sun. 2013. SMAT: An input adaptive auto-tuner for sparse matrix-vector multiplication. In Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation. 117–126.

[23] Wenxuan Li, Helin Cheng, Zhengyang Lu, Yuechen Lu, and Weifeng Liu. 2023. HASpMV: Heterogeneity-Aware Sparse Matrix-Vector Multiplication on Modern Asymmetric Multicore Processors. In 2023 IEEE International Conference on Cluster Computing (CLUSTER). IEEE, 209–220.

[24] Changxi Liu, Biwei Xie, Xin Liu, Wei Xue, Hailong Yang, and Xu Liu. 2018. Towards efficient spmv on sunway manycore architectures. In Proceedings of the 2018 International Conference on Supercomputing. 363–373.

[25] Weifeng Liu and Brian Vinter. 2015. CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In Proceedings of the 29th ACM on International Conference on Supercomputing. 339–350.

[26] Yuechen Lu and Weifeng Liu. 2023. DASP: Specific Dense Matrix Multiply-Accumulate Units Accelerated General Sparse Matrix-Vector Multiplication. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. 1–14.

[27] Deborah T Marr, Frank Binns, David L Hill, Glenn Hinton, David A Koufaty, J Alan Miller, and Michael Upton. 2002. Hyper-Threading Technology Architecture and Microarchitecture. Intel Technology Journal 6, 1 (2002).

[28] Duane Merrill and Michael Garland. 2016. Merge-based sparse matrix-vector multiplication (spmv) using the csr storage format. Acm Sigplan Notices 51, 8 (2016), 1–2.

[29] Mehryar Mohri. 2002. Semiring frameworks and algorithms for shortest-distance problems. Journal of Automata, Languages and Combinatorics 7, 3 (2002), 321–350.

[30] Naveen Namashivavam, Sanyam Mehta, and Pen-Chung Yew. 2021. Variable-sized blocks for locality-aware SpMV. In 2021 IEEE/ACM International Symposium on Code Generation and Optimization

(CGO). IEEE, 211–221.

[31] Maxim Naumov, L Chien, Philippe Vandermersch, and Ujval Kapasi. 2010. Cusparse library. In GPU Technology Conference.

[32] Naser Sedaghati, Te Mu, Louis-Noel Pouchet, Srinivasan Parthasarathy, and P Sadayappan. 2015. Automatic selection of sparse matrix representation on GPUs. In Proceedings of the 29th ACM on International Conference on Supercomputing. 99–108.

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

[34] Tian Xia, Gelin Fu, Chenyang Li, Zhongpei Luo, Lucheng Zhang, Ruiyang Chen, Wenzhe Zhao, Nanning Zheng, and Pengju Ren. 2022. A Comprehensive Performance Model of Sparse Matrix-Vector Multiplication to Guide Kernel Optimization. IEEE Transactions on Parallel and Distributed Systems 34, 2 (2022), 519–534.

[35] Biwei Xie, Jianfeng Zhan, Xu Liu, Wanling Gao, Zhen Jia, Xiwen He, and Lixin Zhang. 2018. Cvr: Efficient vectorization of spmv on x86 processors. In Proceedings of the 2018 International Symposium on Code Generation and Optimization. 149–162.

[36] Shengen Yan, Chao Li, Yunquan Zhang, and Huiyang Zhou. 2014. yaSpMV: Yet another SpMV framework on GPUs. Acm Sigplan Notices 49, 8 (2014), 107–118.

[37] Serif Yesil, Azin Heidarshenas, Adam Morrison, and Josep Torrellas. 2020. Speeding up spmv for power-law graph analytics by enhancing locality & vectorization. In SC20: International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE, 1–15.

[38] Serif Yesil, Azin Heidarshenas, Adam Morrison, and Josep Torrellas. 2023. WISE: Predicting the Performance of Sparse Matrix Vector Multiplication with Machine Learning. In Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming. 329–341.

[39] Yue Zhao, Jiajia Li, Chunhua Liao, and Xipeng Shen. 2018. Bridging the gap between deep learning and sparse matrix format selection. In Proceedings of the 23rd ACM SIGPLAN symposium on principles and practice of parallel programming. 94–108.

[40] Yue Zhao, Weijie Zhou, Xipeng Shen, and Graham Yiu. 2018. Overhead-conscious format selection for SpMV-based applications. In 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS). IEEE, 950–959.