CAMLB-SpMV: An Efficient Cache-Aware Memory Load-Balancing SpMV on CPU

Jihu Guo
National University of Defence Technology
Changsha, Hunan, China

Rui Xia
National University of Defence Technology
Changsha, Hunan, China

Xiang Zhang
National University of Defence Technology
Changsha, Hunan, China

Jie Liu
liujie@nudt.edu.cn
National University of Defence Technology
Changsha, Hunan, China

Abstract

Sparse Matrix-Vector Multiplication (SpMV) plays a crucial role in scientific computing, but severe load imbalance among threads restricts its performance. Previous load-balancing methods have primarily ignored the CPU's cache line-based memory access characteristics and the impact of data locality during workload evaluation and partitioning, leading to limited effect in load balancing.

To address this issue, we propose a cache-aware memory load-balancing SpMV algorithm, CAMLB-SpMV, based on the Compressed Sparse Row (CSR) format. We evaluate all memory access loads of CSR-based SpMV at the cache line level and utilize a sliding window to record accesses to x, enabling the load evaluation to perceive data locality. Finally, the total workload is evenly distributed to threads to achieve load-balancing.

Experimental results on 2661 sparse matrices from SuiteSparse sparse matrix dataset demonstrate that CAMLB-SpMV surpasses Intel MKL, Merge-Based, CSR5-AVX512, and CVR by an average factor of 1.16x, 1.19x, 2.16x, and 1.17x (up to 13.77x, 7.45x, 15.89x, and 8.63x), respectively, on Intel Xeon Platinum 9242. Moreover, it outperforms AMD AOCL, Merge-Based, and CSR5-AVX2 by an average factor of 2.70x, 1.62x, and 3.40x (up to 25.03x, 4.36x, and 13.7x) on AMD EPYC 7542.

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

Conference acronym 'XX, August 12–15, 2024, Gotland, Sweden

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Keywords: CPU, SpMV, Load balance, Cache line

ACM Reference Format:
Jihu Guo, Rui Xia, Xiang Zhang, and Jie Liu . 2024. CAMLB-SpMV: An Efficient Cache-Aware Memory Load-Balancing SpMV on CPU. In *Proceedings of The 53rd International Conference on Parallel Processing (Conference acronym 'XX)*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Sparse matrix-vector multiplication (SpMV) is a vital algorithm widely used in a variety of scientific computing applications in scenarios such as PageRank, graph partitioning and clustering, finite element analysis, and solving systems of linear equations [1, 11, 23, 32]. However, SpMV algorithms often become performance bottlenecks in related applications due to their profoundly memory-constrained nature [17, 18, 22, 37]. Therefore, efficient implementation and optimization of SpMV are crucial for improving the efficiency of scientific computing.

Multithreaded parallel computing on modern multicore Out-of-Order (OoO) CPUs improves SpMV's computational efficiency [40]. The most widely used parallel SpMV algorithm is CSR-based SpMV. The workload division method of CSR-based SpMV is Row-splitting, i.e., one row at a time is assigned to one thread for processing. However, the irregularity and randomness of the distribution of nonzeros in sparse matrices lead to a massive difference in the number of nonzeros to be computed when executing the parallel CSR-based SpMV algorithm, even if the number of rows allocated to each thread is equal, which leads to load imbalance.

To solve the load imbalance caused by the coarse-grained task division of Row-splitting, many works use the fine-grained Nnz-splitting method to divide the workload, i.e., assigning the same number of nonzeros to each thread [13, 15, 38, 39]. Alternatively, they may use other methods to achieve the same effect as Nnz-splitting, such as blocking [33], 2D-tiling [28], etc. Similarly, the Hybrid-splitting method divides the sum of the rows and the nonzeros as the workload equally to each thread and considers the data write-back overhead [31].

However, the granularity of workload division in previous load-balancing methods (usually a nonzero) is smaller than that of CPU memory access (a cache line), which leads to the fact that even if each thread is assigned the same number of nonzeros in Nnz-splitting (or number of rows + number of nonzeros in Hybrid-splitting), the number of cache lines to be read varies greatly. As a profoundly memory-constrained algorithm, the performance of the SpMV algorithm is severely impacted by such load imbalance in accesses. Moreover, previous load-balancing algorithms also ignore the data locality in workload partitioning workflow, which may lead to limited load-balancing effectiveness. For example, if the data locality of one thread is worse than other threads, even if the workload of each thread is equal, the thread is more likely to become a performance bottleneck.

To solve the above problems, we propose CAMLB-SpMV, a cache-aware memory access load-balancing SpMV algorithm based on the CSR format. CAMLB-SpMV evaluates the access load of CSR-based SpMV from the cache line level and uses a sliding window method to record random accesses to tune the workload evaluation so that the load division can perceive the data locality. Compared with previous load balancing methods, CAMLB aligns better with the cache line access characteristics of the CPU and achieves a more accurate load evaluation. Finally, the total read-write memory access load is divided equally to balance the workload.

We conducted experiments using 2661 sparse matrices collected from SuiteSparse [16] on Intel Xeon Platinum 9242 and AMD EPYC 7542 processors, respectively. Experimental results on the Intel CPU show that CAMLB-SpMV achieves an average speedup of 1.14x, 1.18x, 2.12x, and 1.15x (up to 13.77x, 7.45x, 15.89x, and 8.63x) compared to Intel Math Kernel Library (MKL, version 2022.1) [36], Merge-Based [31], CSR5-AVX512 [28], and CVR [38], respectively. Experimental results based on AMD CPUs show that CAMLB-SpMV achieves an average speedup of 2.70x, 1.62x, and 3.40x (up to 25.03x, 4.36x, and 13.7x) compared to AMD Optimizing CPU Libraries (AOCL, version 4.0.0) [3], Merge-Based [31], and CSR5-AVX2 [28], respectively. The contributions of this paper are as follows:

1. We analyze previous load-balancing methods and point out that SpMV on the CPU needs to evaluate workload in cache line granularity and consider data locality in workload evaluation.
2. We propose CAMLB-SpMV, a cache-aware memory load-balancing SpMV algorithm that achieves more precise workload evaluation and partitioning while also being able to perceive data locality on the CPU compared to previous load-balancing methods.
3. Our CAMLB-SpMV outperforms other benchmark SpMV algorithms on Intel Xeon Platinum 9242 and AMD EPYC 7542 CPUs, showing better average performance, especially on irregular sparse matrices.

2 BACKGROUND

2.1 CSR Format and CSR-based SpMV

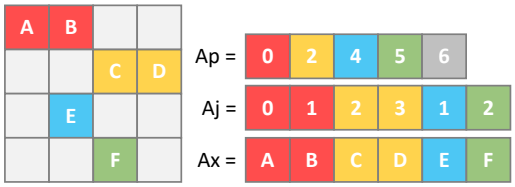


Figure 1. An example of the CSR format

Compressed Sparse Row (CSR) is the most widely used sparse matrix storage format, and Fig. 1 shows an example of CSR format. In CSR format, Ap records the subscripts of the first nonzero of each row in Aj and Ax. $Ap[i]$ denotes the starting position of the nonzeros of the i^{th} row in the Aj and Ax arrays, and $Aj[j]$ and $Ax[j]$ denote the column index and value of the j^{th} nonzero, respectively. CSR-based SpMV is the most widely used SpMV algorithm, and it is supported by several scientific computation libraries, such as Intel MKL [36], AMD AOCL [3], and Nvidia CuSparse. [34], and the parallel CSR-based SpMV algorithm is shown in the algorithm 1. The parallel CSR-based SpMV algorithm uses Row-splitting to distribute the workload, i.e., each thread computes and writes back one row at a time, which can lead to severe load imbalance.

Algorithm 1 Standard Parallel CSR-Based SpMV

Input: Ap, Aj, Ax, x
Output: y

```
1: #pragram 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]] × Ax[j]
6:     y[i] = sum
```

2.2 Previous Load-balancing Methods

To address the load imbalance caused by Row-splitting, previous load balancing methods adopted finer-grained load partitioning, primarily including Nnz-splitting and Hybrid-splitting.

Nnz-splitting is the most widely used load partitioning method, which considers the total number of nonzeros in the matrix as the total workload and assigns an equal number of nonzeros to each thread. For example, CVR [38] evenly distributes nonzeros before performing format conversion so that each thread processes the same number of nonzeros. In addition to directly allocating an equal number of nonzeros to each thread, CVB [33] partitions the matrix into blocks

based on the number of nonzero elements and evenly distributes the nonzero elements within each block for thread processing, CSR5 [28] applies 2D-tilling to CSR format and allocates to each thread. These methods achieve a similar effect with Nnz-splitting.

Merge-based SpMV [31] uses Hybrid-splitting, which considers the total workload as the sum of the total number of nonzeros and the number of rows in the matrix and assigns an equal number of nonzeros plus rows to each thread. Compared with Nnz-splitting, Hybrid-splitting considers the load of writing to y additionally. Figure 2 shows an example to illustrate the above three load partitioning methods.

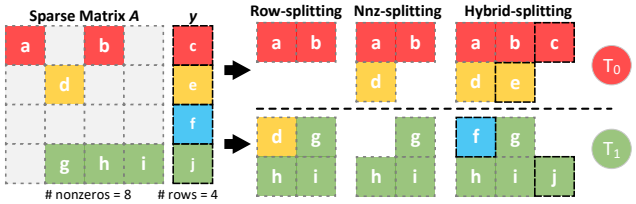


Figure 2. An example of Row-splitting, Nnz-splitting and Hybrid-splitting.

In CSR-based SpMV, threads access the Ap, Aj, and Ax arrays sequentially, while access to the x array is indirect and random. Additionally, since the CPU's memory access unit is typically a cache line with a size of 64 bytes, the number of cache lines that need to be read from the x array when computing nnz nonzeros falls within a range of $[nnz, \alpha \times nnz]$, where α is 8 when $typeof(x)$ is $double$, and α is 16 when $typeof(x)$ is $float$. Taking Figure 3 as an example, when computing $Aj[i:i+4]$, four cache lines from x are required, while when computing $Aj[i+4:i+8]$, only one cache line from x is needed. Therefore, the overly fine-grained load partitioning in Nnz-splitting and Hybrid-splitting results in different threads requiring different numbers of cache lines to be read.

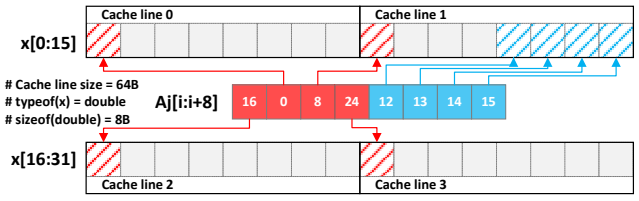


Figure 3. An example of random and imbalance memory access to x.

Figure 4 shows the ratio of the maximum number of cache lines to be read to the minimum number of cache lines to be read among 96 threads when using Row-splitting, Nnz-splitting, and Hybrid-splitting methods for workload partitioning in CSR-based SpMV on 2661 matrices on Intel Xeon Platinum 9242. Taking the matrix $dblp-2010$ as an example, when using Row-splitting, the thread requiring the minimum number of cache lines reads only 10945, while the one needing the maximum reads 15639, resulting in a ratio of 1.43. With Nnz-splitting, these numbers are 7786 and 16377, respectively, with a ratio of 2.10, and with Hybrid-splitting, they are 9556 and 16566, respectively, with a ratio of 1.73. The unequal distribution of cache line reads among threads leads to memory access load imbalance. The larger the ratio shown in Figure 4, the more severe the load imbalance. Additionally, as illustrated in Figure 4, memory access load imbalance exists in almost all matrices using previous load balancing methods. The average ratios are 2.65, 2.56, and 2.42 for Row-splitting, Nnz-splitting, and Hybrid-splitting, and for matrices with more than 100K nonzeros, are 2.22, 1.67, and 1.54, respectively. As SpMV algorithms are known to be deeply memory-bound, the difference in the number of cache lines read among threads leads to memory access load imbalance, which significantly affects their performance. Since Nnz-splitting and Hybrid-splitting methods do not consider the CPU's cache line-based memory access characteristics, their load-balancing effectiveness is limited. In addition, pre-

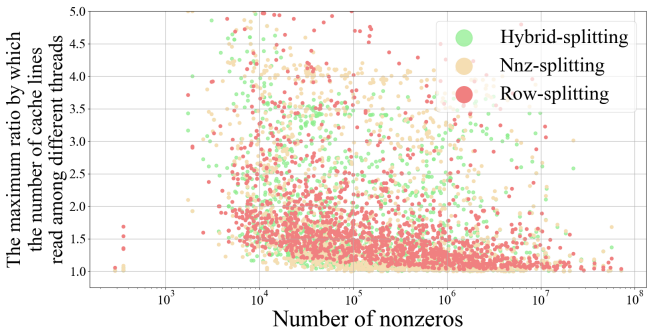


Figure 4. The maximum ratio of cache lines read by threads among 96 threads using different workload partitioning methods in each matrix, with a ratio closer to 1 indicating a more balanced memory access load for each thread.

vious load-balancing algorithms did not consider the impact of data locality during load partitioning. In parallel execution of the SpMV algorithm, data is reused among different threads. When the loads of each thread are equal, if one thread exhibits significantly poorer data locality than others, it becomes a bottleneck for performance. As shown in Fig. 5, data locality among **Thread 0**, **1**, and **2** is better than **Thread 3** on matrix $dblp-2010$ since **Thread 0**, **1**, and **2** access nearly the same range of x which generate many data reuse in Last Level Cache (LLC) and data locality among **Thread 2**, and **3** is better than **Thread 0**, and **1** on matrix $webbase-1M$ since the distributions of nonzeros in **Thread 0** and **1** are more random than in **Thread 2** and **3**. On modern out-of-order (OoO) CPUs, thread behavior is difficult

to predict, making it challenging to consider data locality during load partitioning.

To address these issues, we propose a cache-aware SpMV algorithm CAMLB-SpMV, which balances loads at the cache line level and can perceive cache.

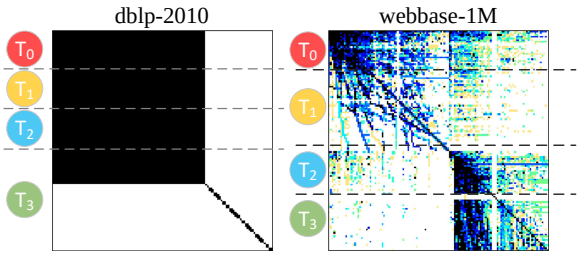


Figure 5. An example of equal workload distribution and different data locality.

3 METHODOLOGY

The vital points of CAMLB-SpMV lie in evaluating the SpMV workload and the equal workload distribution. This chapter first elaborates on how to assess the workload at the cache line level while being data locality aware (Section 3.1) and then explains how to allocate workload to threads based on the workload evaluation results (Section 3.2), and finally introduces our CAMLB-SpMV algorithm (Section 3.3).

3.1 Computing Cache-Aware Memory Load

Unlike Nnz-splitting and Hybrid-splitting, which partition the workload precisely at the granularity of a nonzero element, we partition the workload at the granularity of a cache line. Algorithm 2 presents pseudocode for computing cache-aware memory access load in CSR-based SpMV. There are two important variables: ***workload*** and ***swindow***.

workload stores the memory access load, where $workload[j]$-$workload[i]$ represents the total memory access load from computing the i^{th} nonzero element to the j^{th} nonzero element. Using ***workload***, we accurately determine the corresponding memory access load of processing a specific number of nonzeros.

The ***swindow*** (an abbreviation for Sliding Window used in this study) is a first-in-first-out (FIFO) queue with a length equal to the number of cache lines the cache can store divided by the number of threads (i.e., MAX_SIZE in Algorithm 2), recording cache line accesses in x. CAMLB achieves cache awareness in load evaluation by utilizing ***swindow***. Because it considers the data locality generated by random accesses to x, the evaluation of loads is more in line with real computational scenarios.

Algorithm 2 sequentially traversing the Ap, Aj, and Ax arrays, thus x, and y, but does not involve actual computation. When new cache lines need to be read, the value of $cache\_lines$ is increased to record the total memory accesses, as shown in lines 6, 11, 16, 18, and 20 in Algorithm 2. Lines 8-14 use ***swindow*** to record accesses to x. When ***swindow*** is not full, the non-repetitive cache line access records are saved in x. When ***swindow*** is full, it replaces the earliest cache line access record in ***swindow*** with the new cache line access record. When computing the last nonzero element of each row, the writing load to the y is recorded, as shown in lines 19-21 of Algorithm 2.

We use Figure 6 as an example for Algorithm 2 explanation. Figure 6(a) illustrates the cache line layout of the y, x, Ap, Aj, and Ax. Assuming each cache line can store two double-type data and four int-type data, a cache line can contain two values from Ax, two from x, two from y, four from Ap, or four from Aj, as shown in Figure 6(a).

Algorithm 2 Cache-Aware Memory Load Computing

Input: $Ap, Aj, rows, MAX\_SIZE$
Output: $workload$

```
1:  workload ← {0} # An array to record workload.
2:  cache_lines ← 0 # Number of accessed cache lines.
3:  swindow ← {} # A FIFO queue.
4:  for i = 0; i < rows; i++ do
5:      if Access new cache line in Ap then
6:          cache_lines++;
7:      for j = Ap[i]; j < Ap[i+1]; j++ do
8:          if Access new cache line in x then
9:              cache_line_addr ← new_cache_line_addr
10:             if cache_line_addr not in swindow then
11:                 cache_lines++;
12:                 if swindow.size() == MAX_SIZE then
13:                     swindow.pop().
14:                 swindow.push(cache_line_addr)
15:         if Access new cache line in Ax then
16:             cache_lines++;
17:         if Access new cache line in Aj then
18:             cache_lines++;
19:         if Access new cache line in y then
20:             cache_lines++;
21:         workload[j+1] ← cache_lines
```

Figure 6(b) uses the y, x, Ap, Aj, and Ax shown in Figure 6(a) to explain Algorithm 2. When computing the first nonzero element (i=0, j=0), $Ap[0]$ and $Ap[1]$ are read to find the range of nonzeros in Ax and Aj for this row. Then, $Ax[0]$, $Aj[0]$, and $x[Aj[0]]$ are read to perform $sum+=x[Aj[0]] \times Ax[0]$, requiring access a total of 4 cache lines. Since the ***swindow*** is empty, the access to **cache line 0** of x is recorded, and then update $workload[1]$ to $workload[0] + 4$.

When computing the second nonzero element (i=0, j=1), as the cache lines read during the computation of the first nonzero element already contain $Ap[0]$, $Ap[1]$, $Ax[1]$, and $Aj[1]$, only accessing $x[Aj[1]]$ results in accessing a new

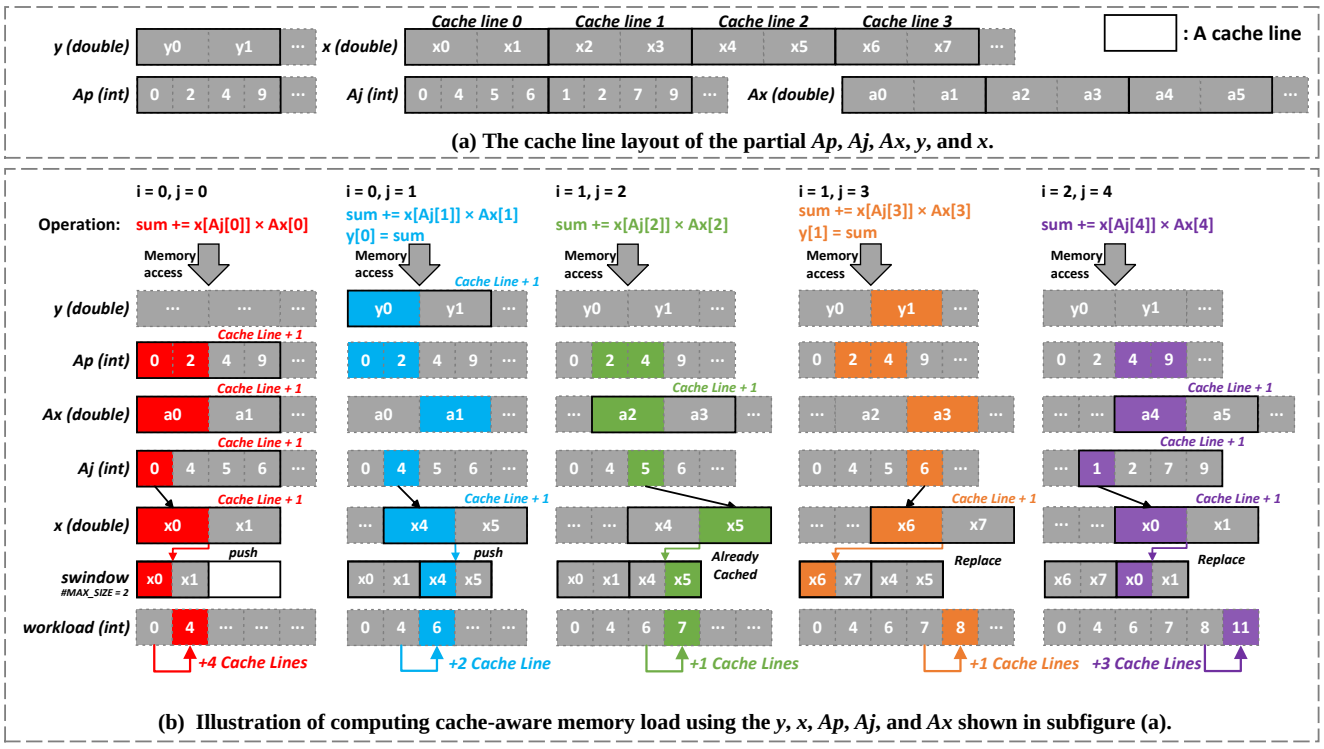


Figure 6. An example of computing cache-aware memory access load.

cache line (**cache line 2** of x). Then, the record of accessing **cache line 2** is added to the ***swindow***. Since the second nonzero element is the last in the first row, performing $y[0]$=*sum* requires reading a new cache line in y. Therefore, $workload[2]$ is updated to $workload[1] + 2$.

Processing the third nonzero (i=1, j=2) must access a new cache line containing $Ax[2]$. When accessing $x[Aj[2]]$, a record of accessing **cache line 2**, which contains $x[Aj[2]]$, is found. Therefore, we regard this cache line is already in the cache and do not need to access the main memory again. Then update $workload[3]$ to $workload[2] + 1$.

Accessing $x[Aj[3]]$ is not recorded in the ***swindow*** when computing the fourth nonzero (i=1, j=3). As the ***swindow*** is full (assuming MAX_SIZE=2 for explanation), the earliest record of accessing **cache line 0** in the ***swindow*** is replaced with the record of accessing **cache line 3**, and then update $workload[4]$ to $workload[3] + 1$.

When processing the fifth nonzero element (i=2, j=4), accessing $Ax[4]$ and $Aj[4]$ requires accessing two new cache lines. When accessing $x[Aj[4]]$, the earliest accessing record of **cache line 0** is replaced with the record of accessing **cache line 2**, and then update $workload[5]$ to $workload[4]+3$.

3.2 Workload Partitioning

If a sparse matrix has nnz nonzeros, $workload[nnz]$ represents the total memory load required to compute all nonzeros. Algorithm 3 shows how to divide the total workload. We use Figure 7 to explain Algorithm 3. As depicted in Figure 7, the total memory load of this sparse matrix is 47, and four threads are used for performing SpMV. Therefore, each thread is allocated an average workload of (47 + 4 - 1) / 4 = 12 (as indicated in line 3 of Algorithm 3). Then, we sequentially perform binary searches in the ***workload*** array to find the first index greater than or equal to 12, 24, 36, and 48 to get the ranges of nonzeros, resulting in indices 6, 12, 20, and 28, respectively. Hence, the ranges of nonzeros that each of the four threads needs to compute are [0, 6), [6, 12), [12, 20), and [20, 28), denoted as $thread\_nnz\_bound$ (as shown in line 8, and 9 of Algorithm 3). According to $thread\_nnz\_bound$, we perform binary searches in the Ap array to find the first index greater than or equal to 6, 12, 20, and 28 to get the row ranges and resulting in indices 2, 3, 5, and 8, respectively. Therefore, the row ranges that each of the four threads needs to process are [0, 2), [2, 3), [3, 5), and [5, 8), denoted as $thread\_row\_bound$ (as indicated in line 11, and 12 of Algorithm 3).

3.3 CAMLB-SpMV Algorithm

The implementation of CAMLB-SpMV is illustrated in Algorithm 4. Since we split each array continuously, only adjacent threads need reduction. We use $tail\_sum$ and $tail\_row$ to keep track of the $partial\_sum$ and the offset (aka row index) to y. Each thread has no write-back conflict with

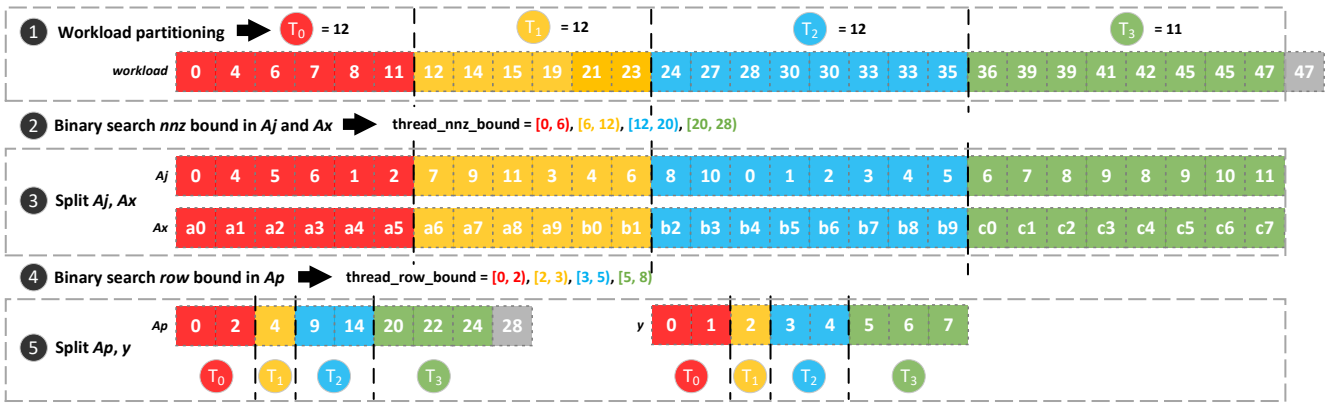


Figure 7. An illustration of workload partitioning.

Algorithm 3 Workload Partitioning

Input: $workload, Ap, nnz, rows$
Output: $thread\_nnz\_bound, thread\_row\_bound$

1: $total\_workload = workload[nnz]$
2: $thread\_num = omp\_get\_max\_threads()$
3: $workload\_per\_thread = total\_workload/thread\_num$
4: $thread\_nnz\_bound[0] = 0$
5: $thread\_row\_bound[0] = 0$
6: **#pragram omp parallel for**
7: **for** $tid = 0; tid < thread\_num; tid++$ **do**
8: $end = workload\_per\_thread \times (tid + 1)$
9: $nnz\_bound = BinarySearch(workload + 1, end)$
10: $thread\_nnz\_bound[tid + 1] = nnz\_bound$
11: $row\_bound = BinarySearch(Ap + 1, nnz\_bound)$
12: $thread\_row\_bound[tid + 1] = row\_bound$

other threads until computing the last row. Therefore, direct write-back to y is available, as shown in lines 10-14. When processing the partial of the last row, CAMLB-SpMV uses $tail\_sum[tid]$ to store the $partial\_sum$ and $tail\_row[tid]$ for the row index, as shown in lines 15-19. Finally, the main thread performs the reduction to obtain an entire result, as shown in lines 24-25 of the algorithm. This algorithm design is similar with Merge-based SpMV [31], however, there are two main differences: 1) CAMLB-SpMV partitions the matrix with equally allocating ***workload*** while Merge-based SpMV partitions the matrix with equal number of $nnz+rows$. Therefore, CAMLB-SpMV's workload evaluation method aligns better with the CPU memory access characteristic. 2) CAMLB-SpMV treats workload distribution as a preprocessing method, while Merge-Based performs workload boundary computing during every iteration of SpMV. Though the efficiency of the binary search is remarkable, thousands of iterations will also accumulate much overhead. So CAMLB-SpMV performs better while performing enough iterations of SpMV with only a tiny additional memory overhead, which is more friendly to iterative SpMV.

Algorithm 4 CAMLB-SpMV Algorithm

Input: $Ap, Aj, Ax, thread\_nnz\_bound, thread\_row\_bound$
Output: y

1: $thread\_num = omp\_get\_max\_threads()$
2: $tail\_sum[thread\_num] = \{0\}$
3: $tail\_row[thread\_num] = \{0\}$
4: **#pragram 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: $e\_row = thread\_row\_bound[tid + 1]$
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: $y[s\_row] = sum$
15: $partial\_sum = 0$
16: **for** $; s\_nnz < e\_nnz; s\_nnz++$ **do**
17: $partial\_sum+=x[Aj[s\_nnz]] \times Ax[s\_nnz]$
18: $tail\_sum[tid] = partial\_sum$
19: $tail\_row[tid] = e\_row$
20: **for** $tid = 0; tid < thread\_num; tid++$ **do**
21: $y[tail\_row[tid]]+=tail\_sum[tid]$

4 EVALUATION

4.1 Experimental Setup

We conduct our experiments on Intel Xeon Platinum 9242 and AMD EPYC 7542 processors, and the main parameters of the experimental platforms are shown in Table 1. The sparse matrix dataset used in the experiments consists of 2661 matrices from SuiteSparse [16], accounting for 92% of the total 2893 sparse matrices. These matrices cover a wide range of domains, including linear programming problems, combinatorial problems, computational fluid dynamics problems, undirected weighted graphs, and others, totaling 85 domains.

Our experiments compared Benchmark SpMV algorithms,

Table 1. 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

including vendor-supported Intel MKL [36] (version 2022.1), vendor-supported AMD AOCL (version 4.0.0) [3], Merge-based [31], CSR5 [28], and CVR [38]. For experiments based on Intel Xeon Platinum 9242, the CSR5-AVX512 version was used. Since AMD EPYC 7542 does not support the AVX512 instruction set, the CSR5-AVX2 version was used. All open-source codes were compiled using **icpx -O3 -qopenmp -xCORE-AVX512** or **clang++ -O3 -laoclsparse -lblis-mt -lpthread -fopenmp -mavx** with the publicly available source code provided by the authors.

4.2 Performance Comparison

We conducted 100 warm-up and 1000 SpMV runs using different SpMV algorithms for each matrix, and took the averages to evaluate the performance of SpMV. The performance test results on the Intel Xeon Platinum 9242 are shown in Fig. 8(a), and the results on the AMD EPYC 7542 are depicted in Fig. 8(b).

Table 2 presents the speedup ratio statistics. For analysis convenience, we separately calculated the speedup ratios of CAMLB relative to other Benchmark SpMV algorithms on sparse matrices with $nonzeros$<100K and $nonzeros$≥100K. For Intel MKL, we denote the optimized SpMV results using *mkl_sparse_optimize* as MKL-O. For AMD AOCL, we denote the optimized SpMV results using *aoclsparse_optimize* as AOCL-O. The data from Table 2 on Intel Xeon Platinum 9242 indicates that on matrices with $nonzeros$<100K, CAMLB achieves average speedup ratios of 1.04x, 1.16x, 1.92x, and 1.09x compared to Intel MKL, Merge-based, CSR5-AVX512, and CVR, respectively. The average speedup ratios for matrices with $nonzeros$≥100K raised to 1.34x, 1.25x, 2.52x, and 1.28x, respectively. The performance of CAMLB improves with the increase of nonzeros, primarily due to the increasing difference in memory access load among threads as the number of nonzeros increases, with memory access load balancing becoming the primary factor in performance improvement. On matrices with $nonzeros$<100K, CAMLB only achieves a speedup ratio of 0.96x relative to MKL-O and 0.98x for matrices with $nonzeros$≥100K. The slightly superior average performance of MKL-O is attributed to its efficient optimization strategies, including vectorization, format conversion, data prefetching, data reordering, etc. However, the load-balancing optimization of CAMLB also enhances SpMV performance, which is particularly evident with maximum speedup ratios of 11.55x over MKL-O on matrix *ins*2, which is highly irregular.

Table 2. Relative speedup of CAMLB-SpMV.

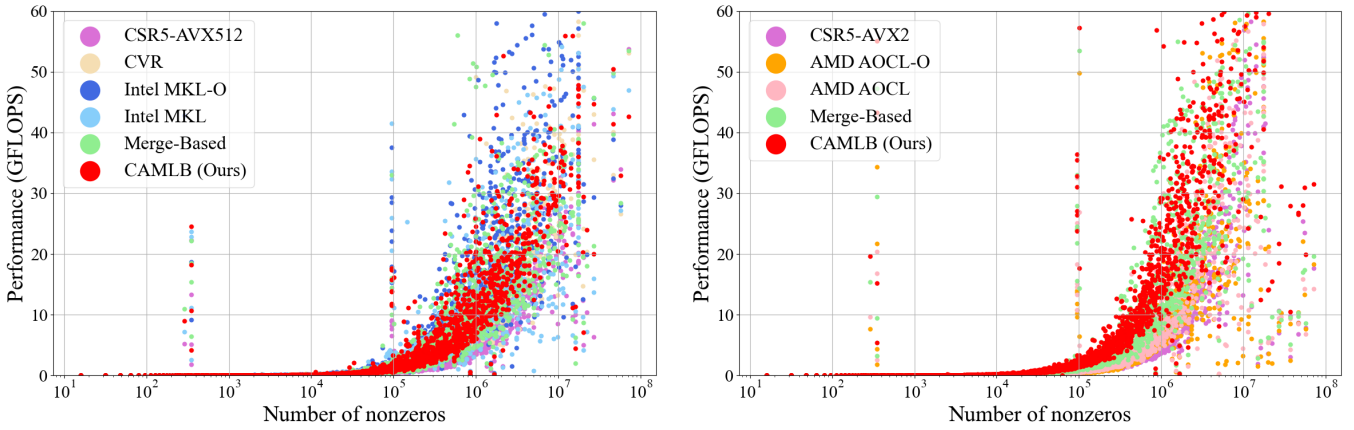
CAMLB vs.	nonzeros < 100K			nonzeros > 100K			total
	Max	Min	Avg.	Max	Min	Avg.	Avg.
MKL	4.11	0.23	**1.04**	**13.77**	0.12	**1.34**	**1.16**
MKL-O†	5.10	0.21	**0.96**	**11.55**	0.20	**0.98**	**0.97**
Merge	7.45	0.34	**1.16**	6.41	0.15	**1.25**	**1.19**
CSR5	**15.89**	0.48	**1.92**	13.68	0.50	**2.52**	**2.16**
CVR	5.75	0.25	**1.09**	**8.63**	0.22	**1.28**	**1.17**
AOCL	7.35	0.91	**2.85**	**25.03**	0.51	**2.48**	**2.70**
AOCL-O†	10.86	0.99	**3.57**	**58.25**	0.44	**3.02**	**3.35**
Merge	**4.36**	0.52	**1.74**	3.42	0.26	**1.45**	**1.62**
CSR5	10.01	1.28	**3.47**	**13.70**	0.62	**3.28**	**3.40**

† MKL-O represents the SpMV optimized using *mkl_sparse_optimize* provided by Intel MKL, while AOCL-O represents the SpMV optimized using *aoclsparse_optimize* provided by AMD AOCL.

On AMD EPYC 7542, CAMLB achieves average speedup ratios of 2.85x, 3.57x, 1.74x, and 3.47x compared to AOCL, AOCL-O, Merge-based, and CSR5-AVX2, respectively, on matrices with $nonzeros$<100K, and average speedup ratios of 2.70x, 3.35x, 1.62x, and 3.40x, on matrices with $nonzeros$≥100K. Besides, CAMLB achieves the maximum speedup ratios of up to 25.03x, 58.25x, 4.36x, and 13.7x compared to AOCL, AOCL-O, Merge-based, and CSR5-AVX2, respectively.

4.3 Memory Load Analysis

To provide a more detailed analysis, we selected 15 representative sparse matrices, and the information on these matrices is shown in Table 3 and the performance comparison results are shown in Figure 9. All data in Figure 10 and Figure 9 is normalized to CAMLB. From Figure 10, we see that CAMLB nearly minimizes the loads and misses in both L1 and LLC cache. MKL-O has more cache loads and misses than CAMLB since it needs to perform a more complicated SpMV algorithm and read more data caused by the additional format conversion overhead. Figure 10(a) shows that CAMLB achieved significantly lower overall misses at the L1 cache level compared to Intel MKL, Merge-based, and CVR since CAMLB considers the impact of data locality when partitioning loads, resulting in similar data locality for memory tasks assigned to each thread. Similarly, CSR5 achieved lower L1 cache miss counts by restructuring nonzeros into small blocks using 2D-tiling technology, which adapts well to the cache. However, the data in Figure 10(b) shows that CSR5



(a) Double precision CAMLB performance compared with CSR5-AVX512 version, CVR, Intel MKL-O, Intel MKL, and Merge-based on Intel Xeon Platinum 9242 (96 threads)

(b) Double precision CAMLB performance compared with CSR5-AVX2 version, AMD AOCL-O, AMD AOCL, and Merge-based on AMD EPYC 7542 (64 threads)

Figure 8. The two sub-figures above respectively show the performance (GFLOPS) of CAMLB on the Intel Xeon Platinum 9242 and AMD EPYC 7542 compared to other Benchmark SpMVs.

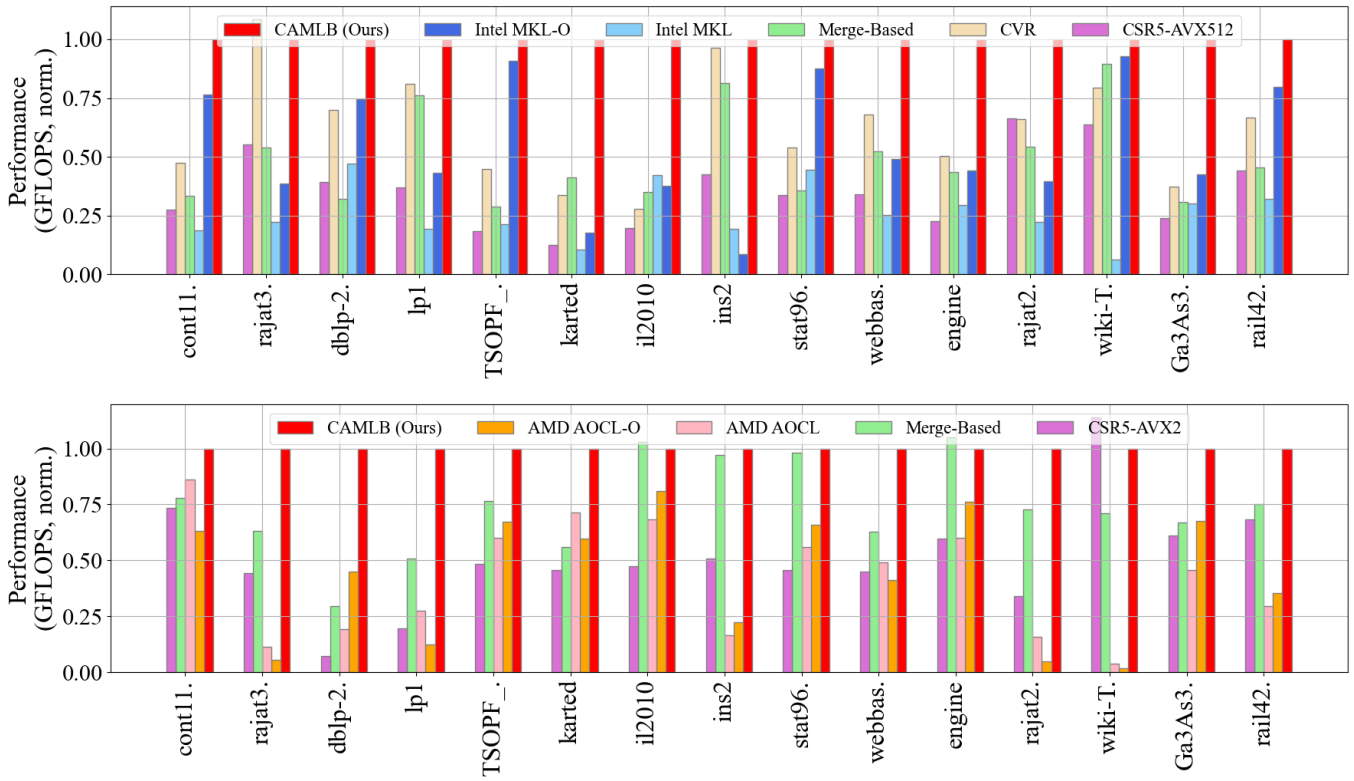
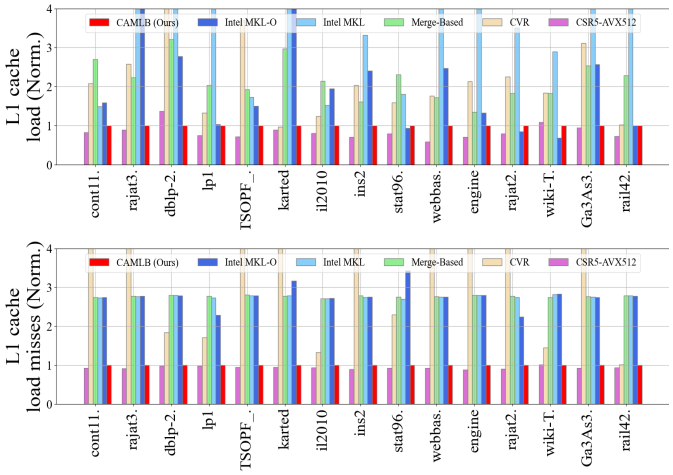
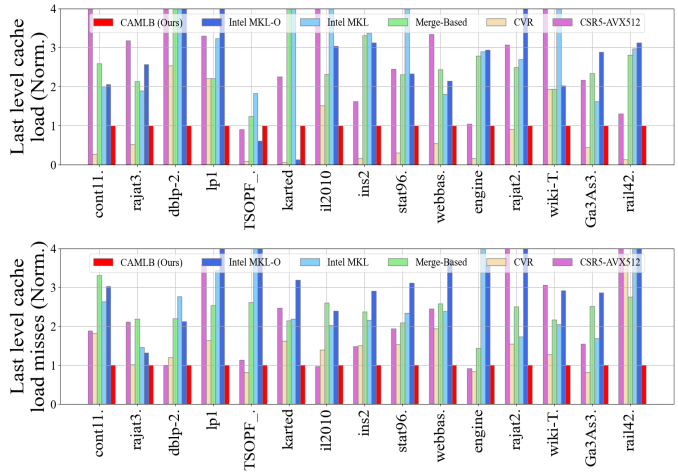


Figure 9. Double precision SpMV performance comparison of the 15 representative matrices on Intel Xeon Platinum 9242 (the sub-figure on the top) and AMD EPYC 7542 (the sub-figure at the bottom).

has more loads and misses in LLC. As depicted in Figure 10(b), CAMLB also exhibited fewer LLC miss counts, attributed to its cache-aware load-balancing partitioning strategy. With equal memory loads across threads, data reuse between threads increases. Besides, the lower LLC misses of CAMLB also indicate that CAMLB has lower data movement from main memory to LLC, efficiently alleviating the memory-bound problem. Although CVR achieves low LLC loads in some matrices, it suffers from higher LLC load misses, L1 loads, and L1 misses.



(a) The number of L1 cache loads (sub-figure at the top) and misses (sub-figure at the bottom) on 15 representative matrices.



(b) The number of LLC cache loads (sub-figure at the top) and misses (sub-figure at the bottom) on 15 representative matrices.

Figure 10. The number of L1 and Last-Level Cache (LLC) loads and misses occurred while performing SpMV with different algorithms on the Intel Xeon Platinum 9242.

Table 3. Information of the 15 representative matrices.

Matrix	Size	Nonzeros	Plot
cont11_l	1.47M×1.96M	5.38M	
rajat30	643K×643K	6,17M	
dblp-2010	326K×326K	1.61M	
lp1	534K×534K	1.63M	
TSOPF_FS_b162_c3	30K×30K	1.80M	
karted	46K×133K	1.77M	
il2010	451K×451K	2.16M	
ins2	309K×309K	2,75M	
stat96v2	29K×957K	2.85M	
webbase-1M	1M×1M	3.10M	
engine	143K×143K	4.71M	
rajat29	643K×643K	3.76M	
wiki-Talk	2.39M×2.39M	5.02M	
Ga3As3H12	61K×61K	5.97M	
rail4284	4K×1.09M	11.28M	

4.4 Preprocessing Overhead

Figure 11 shows the preprocessing time of CSR5, CVR, AOCL-O, MKL-O, and CAMLB. We see that the preprocessing time of CAMLB is shorter than MKL-O in a majority of the total matrices when *nonzeros* $< 10^6$ and shorter than CSR5, CVR, and AOCL-O when *nonzeros* $< 10^5$. When the number of nonzeros increases, the preprocessing time increases linearly since the workload computing algorithm requires a traverse of the CSR format. Although the preprocessing time of CAMLB gradually exceeds that of CSR5 and CVR when the number of *nonzeros* increases, in practice, SpMV is often iterated thousands of times, and the preprocessing time is usually acceptable for better performance.

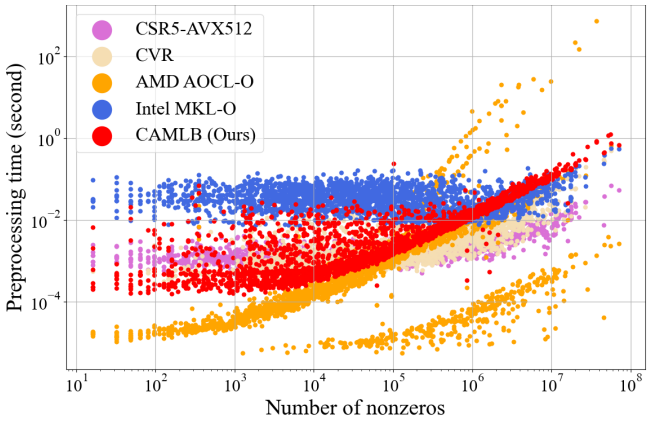


Figure 11. The preprocessing time (second) of CSR5, CVR, AOCL-O, MKL-O, and CAMLB on 2661 matrices.

5 RELATED WORK

Optimizing SpMV has been widely studied on CPUs, GPUs, and other architectures [5, 6, 13, 19, 21, 26–29, 38]. As a deeply memory-bound algorithm, the analyses of SpMV

point out that load imbalance severely affects the performance of SpMV [18, 20, 37].

Predominantly, CPU and GPU load balancing relies on Nnz-splitting [2, 4, 8, 9, 13, 15, 28, 33, 38, 39]. Dalton et al. [15] used Nnz-splitting for workload distribution and perform segment sum at both warp-level and device-level to get results. Chu et al. [13] proposed a more flexible splitting method that applies Row-splitting among workgroups and Nnz-splitting in each workgroup.

Load balancing among threads becomes increasingly crucial as modern CPUs are equipped with numerous cores [30]. Liu et al. proposed CSR5 [28], reorganizing the nonzeros into discrete units $Tile$, each consisting of $\omega \times \sigma$ nonzeros. By assigning $Tile$ to threads, CSR5 achieves an equal effect to Nnz-splitting. This strategy allows for more efficient computation and management of nonzero elements, enhancing the overall performance and load-balancing capabilities of SpMV. Xie et al. proposed CVR [38], which adopts the Nnz-splitting method to achieve load balancing. CVR also implements a *Steal* policy, strategically allocating nonzeros to ensure a balanced nonzero distribution among SIMD lanes and enhancing the effectiveness of the load balance. Merrill et al. [31] introduced Hybrid-splitting techniques for enhanced load distribution. Li et al. [26] proposed a new sparse matrix format, which considers the overhead of accessing x to improve locality on asymmetric multicore processors (AMPs) but the overhead of accessing other arrays is not considered.

Despite extensive literature emphasizing the utilization of SIMD [2, 8, 10, 24, 28, 38], blocking [10, 12, 14, 33, 39], format selection [7, 25, 35, 41–43], data locality [27, 33, 38, 40], particular components (such as matrix multiply-accumulate unit [29], atomic cache [21] etc.) and other strategies to optimize SpMV, the existing methods overlook the memory access load-balancing at the cache line level. It is essential to recognize that, due to the memory-bound feature of SpMV, memory access cost outweighs other factors [29, 40], and achieving load balance in memory access stands as the pivotal element in enhancing overall SpMV performance.

6 CONCLUSION

SpMV is an essential algorithm in scientific computing. However, significant load imbalance among threads hampers its performance. Our experimental results demonstrate that despite using previous load-balancing methods, an imbalanced memory access load still exists among threads. This is because previous load-balancing methods have overlooked the CPU's cache line-based memory access characteristics and the impact of data locality during load partitioning.

To solve these issues, we propose CAMLB-SpMV, a cache-aware memory load-balancing SpMV algorithm based on the CSR format. We assess all memory access loads of CSR-based SpMV at the cache line level. We employ a sliding window approach to record random accesses to x, enabling the load evaluation to be data locality-aware. Subsequently, load balancing distribution is achieved by equal load partitioning and distribution.

We conducted experiments using 2661 matrices obtained from SuiteSparse on both Intel Xeon Platinum 9242 and AMD EPYC 7542 processors. Results from experiments on the Intel CPU demonstrate that CAMLB-SpMV achieves an average speedup of 1.16x, 1.19x, 2.16x, and 1.17x (up to 13.77x, 7.45x, 15.89x, and 8.63x) compared to Intel MKL, Merge-Based, CSR5-AVX512, and CVR, respectively. Similarly, experimental results on the AMD CPU show that CAMLB-SpMV achieves an average speedup of 2.70x, 1.62x, and 3.40x (up to 25.03x, 4.36x, and 13.7x) compared to AMD AOCL, Merge-Based, and CSR5-AVX2. The memory load analysis demonstrates that CAMLB has a lower memory load and cache misses, improving the SpMV performance.

In addition, we hold a positive outlook on the future effectiveness of CAMLB-SpMV on GPUs, as GPUs also have caches. However, due to the differing memory access characteristics between CPUs and GPUs, the CAMLB-SpMV algorithm will require further refinement to adapt to the specific attributes of GPUs.

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] Advanced Micro Devices (AMD). 2024. AOCL-Sparse. https://github.com/amd/aocl-sparse.

[4] Hartwig Anzt, Terry Cojean, Chen Yen-Chen, Jack Dongarra, Goran Flegar, Pratik Nayak, Stanimire Tomov, Yuhsiang M Tsai, and Weichung Wang. 2020. Load-balancing sparse matrix vector product kernels on gpus. ACM Transactions on Parallel Computing (TOPC) 7, 1 (2020), 1–26.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.
[10] Bérenger Bramas and Pavel Kus. 2018. Computing the sparse matrix vector product using block-based kernels without zero padding on processors with AVX-512 instructions. PeerJ Computer Science 4 (2018), e151.
[11] 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.
[12] 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.
[13] 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.
[14] 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.
[15] 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.
[16] 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.
[17] 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.
[18] 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.
[19] 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.
[20] 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.
[21] Jihu Guo, Jie Liu, Qinglin Wang, and Xiaoxiong Zhu. 2023. Optimizing CSR-Based SpMV on a New MIMD Architecture Pezy-SC3s. In International Conference on Algorithms and Architectures for Parallel Processing. Springer, 22–39.
[22] Akira Imakura and Tetsuya Sakurai. 2017. Block Krylov-type complex moment-based eigensolvers for solving generalized eigenvalue problems. Numerical Algorithms 75 (2017), 413–433.
[23] 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.
[24] 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.
[25] 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.
[26] 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.
[27] 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.
[28] 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.
[29] 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.
[30] 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).
[31] 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.
[32] Mehryar Mohri. 2002. Semiring frameworks and algorithms for shortest-distance problems. Journal of Automata, Languages and Combinatorics 7, 3 (2002), 321–350.
[33] 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.
[34] Maxim Naumov, L Chien, Philippe Vandermersch, and Ujval Kapasi. 2010. Cusparse library. In GPU Technology Conference.
[35] 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.
[36] 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.
[37] 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.
[38] 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.
[39] 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.
[40] 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.
[41] 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.

[42] 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.

[43] 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.