HASpMV: Heterogeneity-Aware Sparse Matrix-Vector Multiplication on Modern Asymmetric Multicore Processors

Wenxuan Li
Super Scientific Software Laboratory
China University of Petroleum-Beijing
Beijing, China
wenxuan.li@student.cup.edu.cn

Helin Cheng
Super Scientific Software Laboratory
China University of Petroleum-Beijing
Beijing, China
helin.cheng@student.cup.edu.cn

Zhengyang Lu
Super Scientific Software Laboratory
China University of Petroleum-Beijing
Beijing, China
zhengyang.lu@student.cup.edu.cn

Yuechen Lu
Super Scientific Software Laboratory
China University of Petroleum-Beijing
Beijing, China
yuechenlu@student.cup.edu.cn

Weifeng Liu
Super Scientific Software Laboratory
China University of Petroleum-Beijing
Beijing, China
weifeng.liu@cup.edu.cn

2023 IEEE International Conference on Cluster Computing (CLUSTER) | 979-8-3503-0792-4/23/$31.00 ©2023 IEEE | DOI: 10.1109/CLUSTER52292.2023.00025

***Abstract*—Sparse matrix-vector multiplication (SpMV) is a fundamental routine in computational science and engineering. Its optimization methods on various homogeneous parallel processors, such as CPUs and GPUs, received much attention. Recently, asymmetric multicore processors (AMPs) have heterogeneous performance and efficient cores (e.g., P- and E-cores from Intel and Apple, or Big.LITTLE cores from ARM), or cores with different cache structures (e.g., cores with/without 3D V-Cache from AMD) are becoming one of the mainstream in desktop and workstation computers. However, there lacks heterogeneity-aware research on accelerating SpMV on AMPs.**

We in this paper propose a parallel algorithm called heterogeneity-aware SpMV (HASpMV) for improving the performance of SpMV on the latest 12th- and 13th-Gen AMPs from Intel and Ryzen 9 AMPs from AMD. We first micro-benchmark bandwidth and multi-/single-core SpMV to collect performance characteristics and to motivate our algorithm design, and then develop several optimization techniques to assign workloads between the two types of cores for achieving significantly better cache locality and load balancing. The experimental results show that compared to the latest version of the Intel oneMKL library and the open-source works CSR5 and merge-SpMV, HASpMV achieves an average speedup of 2.61x, 2.31x, and 3.73x (up to 5.23x, 4.46x, and 8.23x) on the i9-12900KF processor. On the i9-13900KF processor, HASpMV achieves an average speedup of 3.17x, 1.52x, and 2.23x (up to 9.46x, 5.31x, and 4.49x). Additionally, when comparing AMD Ryzen 9 7950X3D and 7950X AMPs, HASpMV brings an average speedup of 1.43x, 1.3x, and 1.29x (up to 6.28x, 7.8x, and 10.8x) over AMD Optimizing CPU Libraries (AOCL), CSR5, and merge-SpMV, respectively.

***Index Terms*—Sparse matrix-vector multiplication, Asymmetric multicore processor, Heterogeneity-aware algorithm.**

I. Introduction

Sparse matrix-vector multiplication (SpMV) is an operation to multiply a sparse matrix A and a dense vector x and to give a resulting dense vector y. It is one of the most time-consuming routines in iterative sparse linear solvers (e.g., the conjugate gradient method [1]–[3]) and graph processing frameworks (e.g., the GraphBLAS standard [4]–[6]). Besides, SpMV is in the level-2 sparse basic linear algebra subprograms (sparse BLAS) [7], [8], and has been widely studied on a number of homogeneous parallel processors, e.g., CPUs [9]–[11], GPUs [12]–[20], and long vector processors [21], [22].

Compared with the classic homogeneous processors, the emerging asymmetric multicore processors (AMPs) consist of at least two kinds of cores of distinct compute/memory capacities and of the same ISA [23] or different ISAs [24]. ARM chips with Big.LITTLE cores, Apple and Intel chips with performance and efficient cores (i.e., P- and E-cores), as well as AMD chips including Core Chiplet Dies (CCDs) with and without 3D V-Cache represent the modern AMPs that utilize a single ISA, and are becoming mainstream in the desktop and workstation computers due to overall better performance and energy efficiency [25].

However, despite the potential advantages of using AMPs for SpMV, to the best of our knowledge, there is no existing SpMV methods can well exploit the two kinds of cores in AMPs. The challenges are mainly from that the architecture designs (e.g., frequency, the number of SIMD units and cache hierarchies/capacities) of the two types of cores in Intel and AMD AMPs. Two different cores of modern AMPs will in general lead to task assignment and load balancing problems [26], [27]. In addition, caching behavior and the number of nonzeros of rows (i.e., row length distribution) in sparse matrices can also be very irregular [13], [14], [16], [28]. As a result, the irregularity from the two sides (i.e., processor-side and matrix-side) makes accelerating SpMV on AMPs more difficult than on homogeneous processors.

To address the above challenges, we in this paper propose a parallel algorithm called heterogeneity-aware SpMV

2168-9253/23/$31.00 ©2023 IEEE
DOI 10.1109/CLUSTER52292.2023.00025

(HASpMV) for the latest AMPs from Intel and AMD. Firstly, to understand performance characteristics of different cores, we study the bandwidth and compute power of them through running micro-benchmarks including the stream package [29] and multi-/single-core SpMV. Secondly, to improve cache locality and load balancing of SpMV, we develop a 2-level partitioning scheme that divides a sparse matrix into two parts for P- and E-cores of Intel, and CCD0 and CCD1 of AMD, respectively, and further splits each part according to different number of cores. Thirdly, to store the auxiliary information required in running SpMV, we design a new sparse format called heterogeneity-aware compressed sparse row (HACSR), which is a minor variant of the standard CSR format and provides very low format conversion cost.

In the experiments, we compare our HASpMV with four baseline SpMV methods on Intel and AMD AMPs, i.e., the latest Intel oneMKL library (compare only on Intel AMPs), AMD Optimizing CPU Libraries (AOCL) (compare only on AMD AMPs), the open-source work CSR5 [14] and merge-SpMV [15]. Through benchmarking all 2888 sparse matrices from the SuiteSparse Matrix Collection [30], our experimental results show that HASpMV is significantly faster than the other methods on different AMPs. Specifically, the experimental results show that compared to the latest version of the Intel oneMKL library and the open-source works CSR5 and merge-SpMV, HASpMV achieves an average speedup of 2.61x, 2.31x, and 3.73x (up to 5.23x, 4.46x, and 8.23x) on the i9-12900KF processor. On the i9-13900KF processor, HASpMV achieves an average speedup of 3.17x, 1.52x, and 2.23x (up to 9.46x, 5.31x, and 4.49x). Additionally, when comparing AMD Ryzen 9 7950X3D and 7950X AMPs, HASpMV brings an average speedup of 1.43x, 1.3x, and 1.29x (up to 6.28x, 7.8x, and 10.8x) over AMD Optimizing CPU Libraries (AOCL), CSR5, and merge-SpMV, respectively.

This work makes the following contributions:

- We understand key performance behaviors of SpMV on two kinds of different cores in Intel and AMD AMPs through micro-benchmarking;
- We propose the HASpMV algorithm and the HACSR sparse format for better cache locality and load balancing in SpMV on AMPs;
- We achieve significant performance gain on 2888 matrices over existing SpMV work on the latest Intel and AMD AMPs.

II. Background

A. Sparse Matrix-Vector Multiplication

SpMV refers to the multiplication of a sparse matrix and a dense vector. Figure 1 shows a simple example, and Algorithm 1 presents the pseudocode for parallel SpMV in the CSR format. As can be observed, despite the good parallelism of SpMV (as multiplying rows is independent, as seen in line 1), the workload can become highly imbalanced when row lengths are uneven (see line 3).

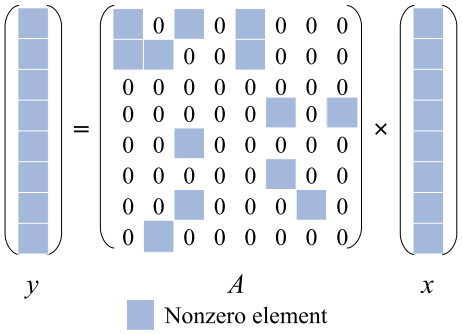


Fig. 1 An example of SpMV, where A is sparse, and x and y are dense.

Algorithm 1 A pseudocode of simple CSR SpMV .

```
1: for i = 0 → numRows in parallel do
2:     y[i] ← 0
3:     for j = csrRowPtr[i] → csrRowPtr[i + 1] do
4:         y[i] ← y[i] + csrVal[j] ∗ x[csrColIdx[j]]
5:     end for
6: end for
```

B. Asymmetric Multicore Processor

At present, there are much research on parallel algorithms [31]–[35], AMPs are becoming the mainstream processors. AMPs typically include at least two types of cores with different frequencies, cache capacities, numbers of cores and SIMD units, and core-group settings. Figure 2 shows the core composition and cache configuration of three AMPs: Intel Core i9-12900KF, Intel Core i9-13900KF and AMD Ryzen 9 7950X3D, which are each composed of two sets of cores.

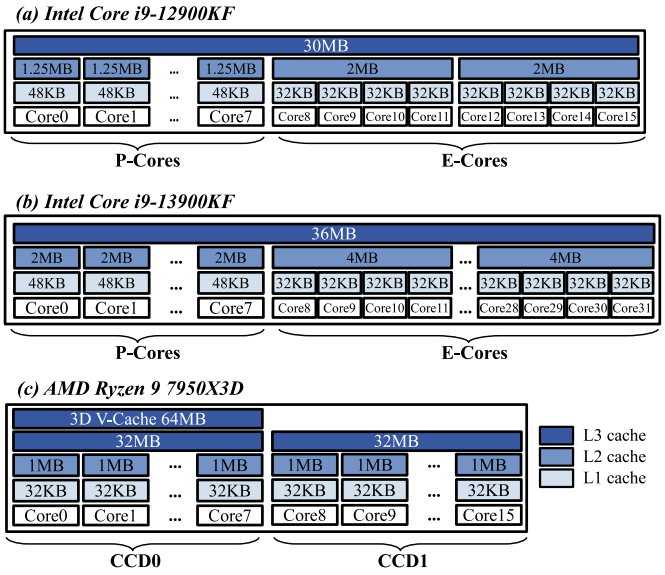


Fig. 2 The core composition and cache configuration of three AMPs.

The Intel Core i9-12900KF is composed of eight P-cores and eight E-cores, while the Intel Core i9-13900KF is composed of eight P-cores and 16 E-cores. The AMD Ryzen 9 7950X3D is composed of two CCDs: CCD0 with a stacked

3D V-Cache of 64MB (working as part of L3 cache), and CCD1 without the 3D V-Cache. Its homogeneous version, AMD Ryzen 9 7950X, lacks the stacked 3D V-Cache in its CCD0, meaning that the two CCDs are the same.

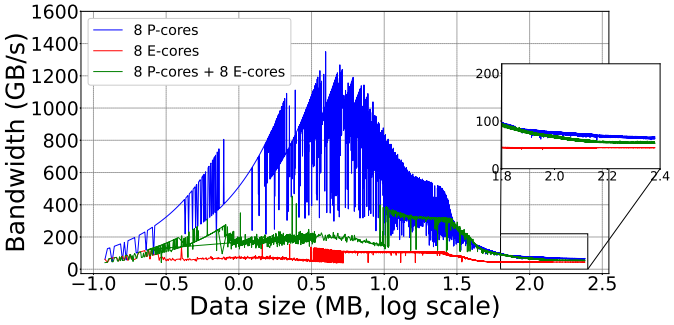
III. Micro-Benchmarks and Motivations

In order to establish the foundation for our algorithm design, it is crucial to understand the performance characteristics of the two types of cores found in modern AMPs. To achieve this, We conduct a series of micro-benchmarks using the three AMPs, as illustrated in Figure 2. Further details about the AMPs are presented in Table I (Before the section V). The following provides an overview of our micro-benchmarking process: (1) The stream triad test [29] to see the achievable bandwidth of the three AMPs. On Intel AMPs, we test all P-cores, all E-cores and all P- and E-cores. On AMD AMPs, we test all cores of CCD0, all cores of CCD1 and all cores of CCD0 and CCD1, (2) A simple parallel SpMV test using Algorithm 1 to see the performance of the two kind of cores alone and combined, and (3) A simple serial SpMV test to see the correlation of row length and SpMV performance differences of a single P- or E-core of Intel, and core of CCD0 or CCD1 of AMD.

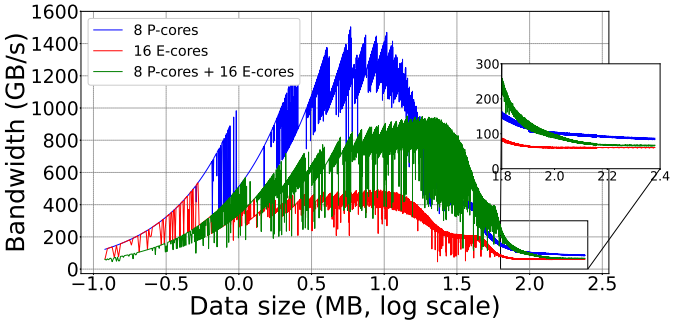
A. Stream Triad Bandwidth Test

Because SpMV is a memory-bound operation [8], first to understand memory access behavior of P- or/and E-cores of Intel (CCD0 or/and CCD1 of AMD) will help our algorithm design. We test the OpenMP version of the triad bandwidth in the stream package with small intervals of the input vectors on three AMPs. In Figure 3 we show the bandwidth under the three composition ways of the two kinds of cores. As can be seen, on both Intel AMPs, bandwidth of pure P-cores is almost always higher than pure E-cores, and in most cases is also higher than using all P- and E-cores (except the region between ∼16 MB and ∼80 MB on i9-13900KF, which exceeds the cache capacity of P-cores, but is within the overall on-chip cache size). In particular, after the bandwidth drops to the DRAM plateau (the two enlarged areas in Figure 3), bandwidth of pure P-cores is still higher than P- plus E-cores. For the AMD AMPs, CCD0 and CCD1 have roughly the same bandwidth, However, with the increase of data size, the bandwidth of CCD0 and CCD1 gradually increases from lower than the CCDs's bandwidth to higher than the CCDs's bandwidth, and finally the bandwidth of the three cases is in a flat bandwidth state with the same value. It is worth noting that compared to CCD1, the bandwidth of CCD0 starts to decline when the data size is larger, due to the fact that CCD0 has a larger L3 cache.

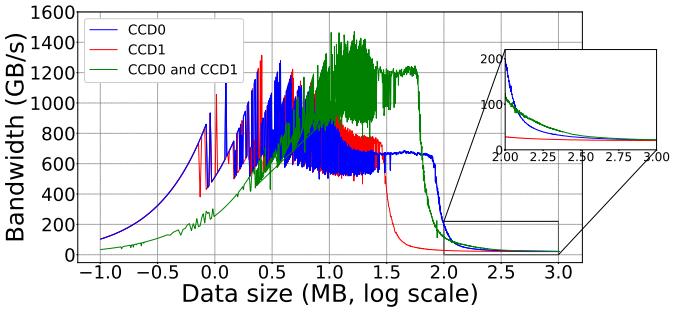
This fact gives us the **first motivation**: Since the performance of the two kind of cores is different for the same data size matrix, simply adding OpenMP pragmas to the for loops and assigning the same amount of workload to cores may bring load imbalance and slow down overall performance on AMPs. Therefore, our algorithm need to assign the best amount of workload to exploit the two types of cores.



(a) Intel Core i9-12900KF



(b) Intel Core i9-13900KF



(c) AMD Ryzen 9 7950X3D

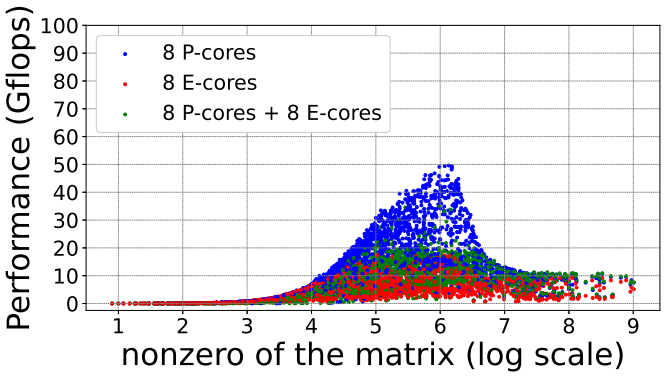
Fig. 3 Stream triad bandwidth test on the two Intel AMPs and one AMD AMP. The y-axis is bandwidth in GB/s, and the x-axis in log10 scale is the total size of input and output vectors.

B. Simple Parallel SpMV Performance Test

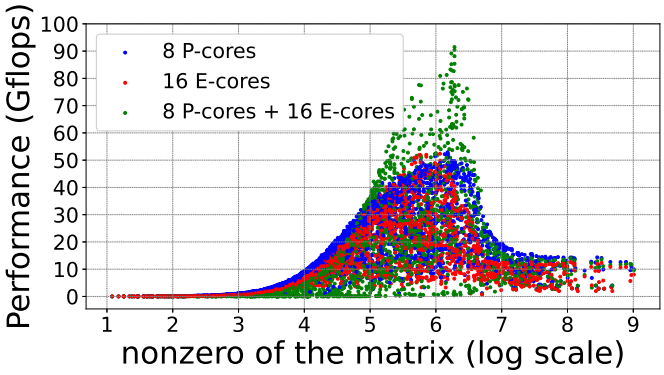
We then use the simple parallel CSR-format SpMV code with all 2888 sparse matrices in the SuiteSparse Matrix Collection [30] to test three sets of data on three AMPs. On the Intel AMPs, we use all P - cores, all E - cores and all P- and E- cores to test SpMV, and on AMD AMPs, use all cores of CCD0, all cores of CCD1, all CCD0 and CCD1 cores to test SpMV. Figure 4 shows the SpMV performance on the Intel and AMD processors. It can be seen that in Intel AMPs pure P-cores are still overall the fastest. However, in 278 and 739 cases out of the 2888 matrices, on i9-13900KF, pure E-cores and P- plus E-cores can bring higher performance than pure P-cores, respectively. This result may come from the fact that the increase in the number of E-cores in the 13-Gen narrowed the performance gap between P-cores and E-cores. In AMD Ryzen 9 7950X3D, there are 1596 and 1519 cases out of the 2888 matrices showing that a single CCD0 and CCD1 bring

higher performance than CCD0 plus CCD1, which may be caused by the imbalance in task allocation.

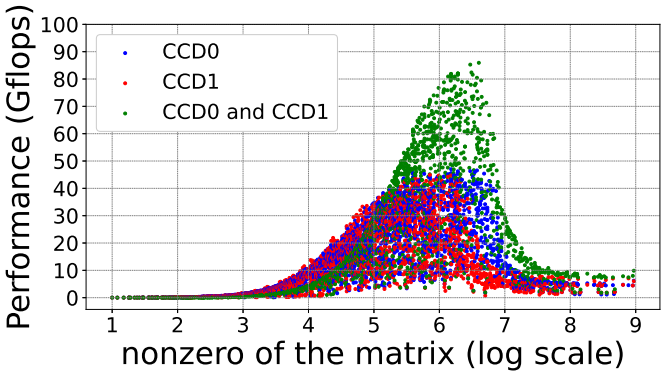
This brings our **second motivation**: using both kinds of cores hopefully achieve higher SpMV performance than pure P-cores of Intel (or CCD0 of AMD) due to a variety of cache behaviors. As a result, the task assignment in our algorithm should not only balance the amount of work, but also consider to achieve higher cache utilization of the workload.



(a) Intel Core i9-12900KF



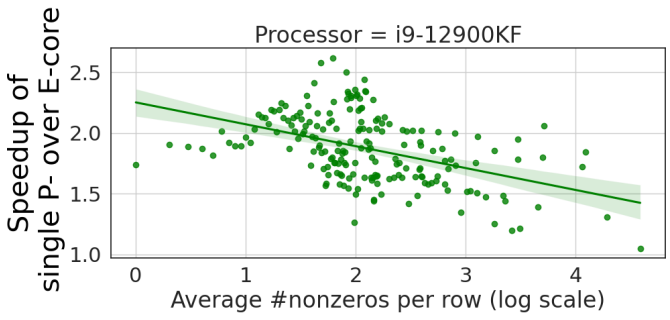
(b) Intel Core i9-13900KF



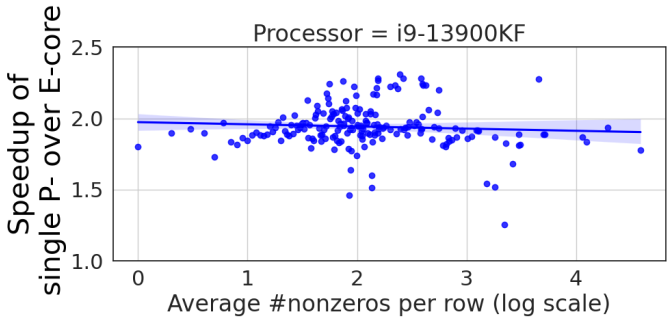
(c) AMD Ryzen 9 7950X3D

Fig. 4 Parallel SpMV performance of the 2888 SuiteSparse matrices on the three AMPs. The y-axis is performance in GFlops, and the x-axis in log10 scale is the number of nonzeros of the matrices.

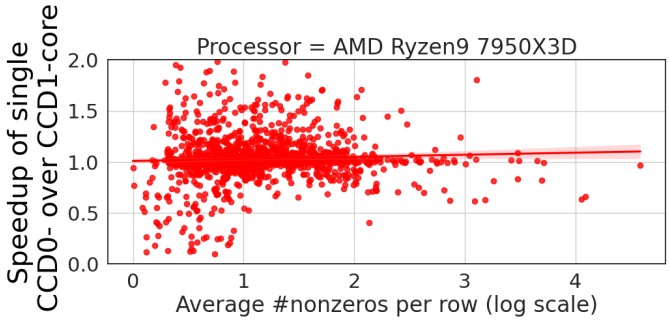
C. Correlation of Row Length and SpMV Performance Test



(a) Intel Core i9-12900KF



(b) Intel Core i9-13900KF



(c) AMD Ryzen 9 7950X3D

Fig. 5 The correlation of the average row length of the 2888 matrices (x-axis in log10 scale) and relative SpMV performance of a single P- over E-core of Intel (CCD0- over CCD1-core of AMD), and the y-axis represents as speedups. Performance of matrices with the same average row lengths are averaged to make the figures clearer, so that the number of dots is less than 2888. The lines drawn show linear regression of scatter dots.

On the basis of the above two micro-benchmarks, we further investigate how to assign tasks to the cores. Because the basic working unit of SpMV is row of the matrix A, and row length (i.e., the number of nonzeros in a row) largely impacts the amount of each workload and cache behavior of accessing the vector x, we conduct a micro-benchmark to see the correlation of SpMV performance provided by a single P- and E-core (core of CCD0 and CCD1) and average row lengths of the sparse matrices, Figure 5 demonstrates the correlation. As can be seen, on i9-12900KF, a P-core gives $\sim$2x and up to $\sim$2.5x performance over an E-core for matrices of short and medium-size rows respectively, but this advantage quickly decreases when the rows are getting longer (for very long rows, P and E-

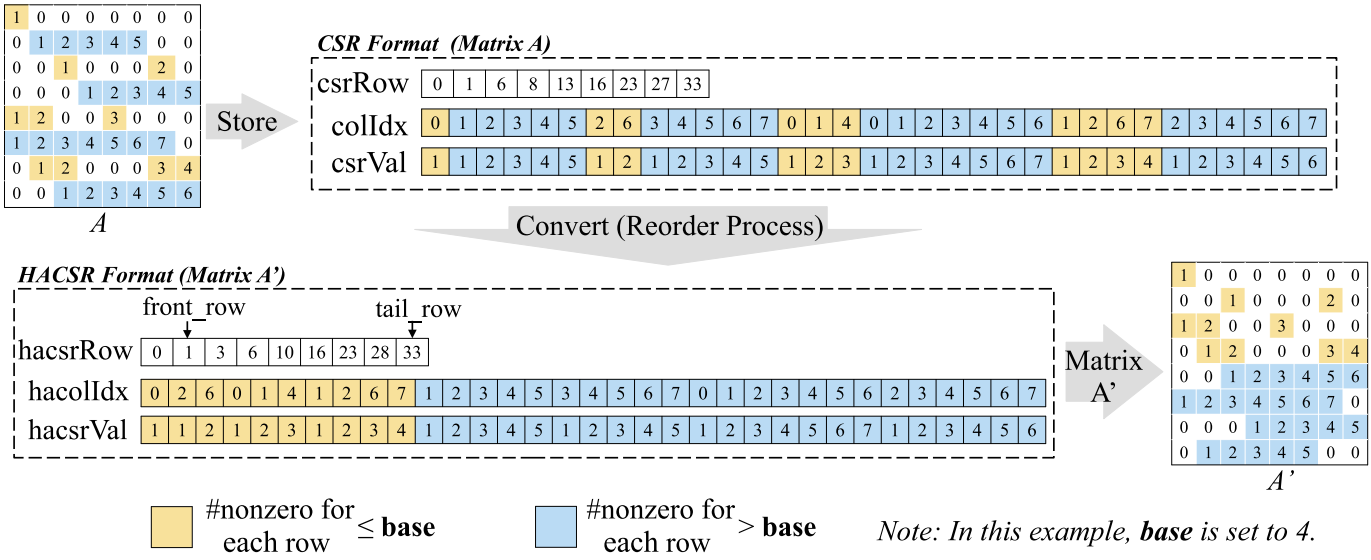


Fig. 6 An example of reordering an 8-by-8 sparse matrix. It reflects the transformation process from CSR to HACSR, in which `hacsrRow`, `hacalIdx` and `hacsrVal` are the data structures of HACSR, and the process from matrix A to matrix A' reflects the process of matrix reorder.

core have similar performance). In contrast, on i9-13900KF, P-core is almost always 2x faster than an E-core, due to possibly improved architecture design. As for 7950X3D, the difference between the two cores is only reflected in the L3 Cache size, so there is no obvious speedup.

This makes the **third motivation** of this work: only considering balancing and cache behavior in workload assignment of the SpMV algorithm may be not enough, since the nature order of the rows of a sparse matrix may not provide the best performance on AMPs. So, it may be better to reorder rows according their lengths, and to give rows of the 'best' lengths to most suitable cores.

IV. Methodology

A. Overview

In this section, we will introduce the HACSR sparse format and delve into the details of the implementation of our HASpMV algorithm. These developments are based on the valuable insights and findings obtained from the micro-benchmarks conducted in the previous section.

Our method takes into consideration the performance difference between the two cores of AMPs when handling long and short rows in sparse matrix-vector multiplication. To optimize performance, we first reorder the matrix to obtain the HACSR format, which is a minor variant of the standard CSR format (detailed in Section IV-B). Furthermore, in order to address the challenge of matrix structure imbalance, we utilize the cache line cost as the basis for matrix partitioning, rather than simply partitioning based on the number of nonzeros (nnz). This allows us to achieve better load balancing across the AMP cores. We design a two-level matrix partitioning approach, partitioning the matrix into two parts based on the bandwidth test of the two types of cores in AMPs. Subsequently, we assign an equal amount of tasks to each core within the same group, effectively solving the challenge caused by processor imbalance. All the aforementioned aspects are discussed in detail in Section IV-C. Additionally, Section IV-D provides a comprehensive description of the execution phase, including the handling of conflicting rows and the incorporation of additional performance improvement techniques. Through these approaches, we aim to maximize the utilization of the AMP cores and optimize the overall performance of SpMV. The detailed explanations in the respective sections will provide a comprehensive understanding of our method's implementation and its effectiveness in tackling the challenges posed by matrix structure and processor imbalance.

B. HACSR format

We show the HACSR format in Figure 6, which is a more suitable matrix compression format for AMPs. It first determines a threshold to distinguish between long and short rows, which the variable $base$ in the figure. Then the matrix is reordered according to the length of each row, so that rows with length less than $base$ are placed in the front part of the matrix, and rows with length greater than $base$ are placed in the back part of the matrix. Reordering places rows back from the first row and forward from the last row according to the $front\_row$ and $tail\_row$ pointers (described in Section IV-C), which results in HACSR in very short time.

C. Preprocessing Phase of HASpMV

The matrix reorder is described in detail in Algorithm 2, where a new array $hacsrRowPtr[]$ is used to store the offset of the first nonzero in each row after reordering (only suitable for Intel 12th Gen Core AMPs, so $csrRowPtr$ is used in the following algorithm statements). The $front\_row$ and $tail\_row$ pointers are used to point the place of next row during matrix reordering: If the length of one row(nnz_num

on line 3) is less than $base$, the row is placed at the position pointed by $front\_row$ (lines 5 through 9); otherwise, the row is placed at the position pointed by $tail\_row$ (lines 10 through 14). In HASpMV, only the row pointers are reordered, and no additional operations are needed for the column indices and values, so the matrix can be reordered in a very short time.

Algorithm 2 The process of reordering matrix in HASpMV

Require: $csrRowPtr[\,], m$
Ensure: $hacsrRowPtr[\,], row\_begin\_nnz[\,]$
1: $front\_nnz \leftarrow 0, tail\_nnz \leftarrow nnz, front\_row \leftarrow 0, tail\_row \leftarrow m-1$
2: **for** $i = 0 \rightarrow m-1$ **do**
3: $nnz\_num \leftarrow csrRowPtr[i+1] - csrRowPtr[i]$
4: $row\_len[i] \leftarrow nnz\_num$
5: **if** $nnz\_num < base$ **then**
6: $row\_begin\_nnz[front\_row] \leftarrow csrRowPtr[i]$
7: $hacsrRowPtr[front\_row+1] \leftarrow front\_nnz + nnz\_num$
8: $front\_nnz \leftarrow front\_nnz + nnz\_num$
9: $front\_row \leftarrow front\_row + 1$
10: **else**
11: $row\_begin\_nnz[tail\_row] \leftarrow csrRowPtr[i]$
12: $tail\_nnz \leftarrow tail\_nnz - nnz\_num$
13: $hacsrRowPtr[tail\_row] \leftarrow tail\_nnz$
14: $tail\_row \leftarrow tail\_row - 1$
15: **end if**
16: **end for**

To more accurately measure the amount of tasks, we take the cost of cache line required for accessing vector x in SpMV operation as the basis for allocating tasks. In Algorithm 3, we provide an approximate cache line cost calculation method, which calculates the cache line cost of each row and stores it in the temporary variable $cost\_x$ (lines 1 to 9). Then the prefix sum operation is performed and the $cost\_sum[]$ array is used to record the total cache line cost before the current row (including the current row). Meanwhile, the left part of Figure 7 also presents the computation strategy of cache line cost. In the figure, we set the cache line size to 16 bits for the sake of presentation (in fact, it is 64 bits on the processors we use), so the vector x is stored in four cache lines, and each row of the 8×8 matrix A is also cut into four parts. The two nonzeros in the same part will record the cost of only one cache line.

To achieve better load balancing, we design a two-level matrix partition method to assign appropriate workload for each core. On the first level, we partition the matrix between the heterogeneous cores. As shown in the left part of Figure 7, we partition the matrix into two parts, one for P-cores (CCD0) and one for E-cores (CCD1). On the second level, as shown in the right of the figure, we partition the two parts of the matrix equally to the two types of homogeneous cores, respectively. Meanwhile, the calculation strategy of task load mentioned in the previous paragraph was used in the whole process. Algorithm 3 shows the detail of partitioning matrix A. Firstly, a threshold $costp$ is calculated based on $P\_proportion$ (differ in different processor) and $COST$ (i.e., the cache line cost of the whole matrix, which corresponds to $cost\_sum[m-1]$ in Algorithm 3), and a binary search is performed to find the row where the threshold is located (lines 1 to 2). Then the matrix task size corresponding to each sum is calculated (line 3 to 4), and then the number of starting and ending rows and nonzero positions of each part are calculated (line 5 to 20). After the above operations, we complete the matrix preprocessing operation suitable for AMPs, and assign suitable tasks to each core. In this phase, HASpMV address the double imbalance challenge brought by heterogeneous architecture and sparse matrix structure.

Algorithm 3 The method to calculate cache line costs in HASpMV

Require: m,$csrRowPtr[\,]$,$csrColIdx[\,]$
Ensure: $cost\_sum[\,]$
1: **for** $i = 0 \rightarrow m$ **do**
2: $cost\_x \leftarrow 0$
3: $ben \leftarrow -1$
4: **for** $j = csrRowPtr[i] \rightarrow csrRowPtr[i+1]$ **do**
5: **if** $csrColIdx[j]/8 > ben$ **then**
6: $cost\_x \leftarrow cost\_x + 1$
7: $ben \leftarrow csrColIdx[j]/8$
8: **end if**
9: **end for**
10: $cost\_sum[i] \leftarrow cost\_sum[i] + cost\_x$
11: $cost\_sum[i+1] \leftarrow cost\_sum[i]$
12: **end for**

Algorithm 4 The process of partitioning matrix in HASpMV

Require: $COST$,$csrRowPtr[\,]$,$csrColIdx[\,]$,$P\_proportion$,m,nnz
Ensure: $nnz\_l[\,]$.$nnz\_r[\,]$,$pl[\,]$,$pr[\,]$
1: $costp \leftarrow COST * P\_proportion$
2: $row\_mid \leftarrow$ *binary search which row the costp is from* 0 *to* m
3: $gapp \leftarrow ceil(costp/CORE\_NUM\_P)$
4: $gape \leftarrow ceil((COST - costp)/CORE\_NUM\_E)$
5: $bound \leftarrow 0$
6: **for** $i = 0 \rightarrow CORE\_NUM\_P + CORE\_NUM\_E$ **do**
7: **if** $i < CORE\_NUM\_P$ **then**
8: $bound \leftarrow bound + gapp$
9: $row \leftarrow$ *binary search which row the bound is from* 0 *to* row_mid
10: **else**
11: $bound \leftarrow bound + gape$
12: $row \leftarrow$ *binary search which row the bound is from* row_mid *to* m
13: **end if**
14: $pr[i] \leftarrow row$
15: $pl[i+1] \leftarrow row$
16: $nnz\_r[i] \leftarrow$ *binary search the nnz from* $csrColIdx[i]$ *to* $csrColIdx[i]$
17: $nnz\_l[i] \leftarrow nnz\_r[i]$
18: **end for**
19: $pr[CORE\_NUM\_P + CORE\_NUM\_E - 1] \leftarrow m$
20: $nnz\_r[CORE\_NUM\_P + CORE\_NUM\_E - 1] \leftarrow nnz$

D. Execution Phase of HASpMV

In this part, HASpMV first binds each part of the task to the corresponding core (by adjusting the OpenMP environment variable '$GOMP\_CPU\_AFFINITY$'). Note that splitting the task can lead to multiple cores computing the same $y[i]$, as shown in red in the right part of Figure 7. HASpMV uses an extra array $extra\_y[]$ to store the tail conflict result of each core and append it to the vector y at the end, as shown in the formula in the bottom left corner of Figure 7. Moreover, we describe this procedure in detail in Algorithm 5. HASpMV parallelizes the partitioned parts, corresponding to one core in each part. Since matrix is partitioned strictly by cache line cost, for a core it may be allocated to only one row of nonzeros (lines 5 to 8) or incomplete lines plus a full number of lines (lines 9 to 14), and each core will only record the collision data processed by the last line. And add it to the vector y at the end (lines 15-17).

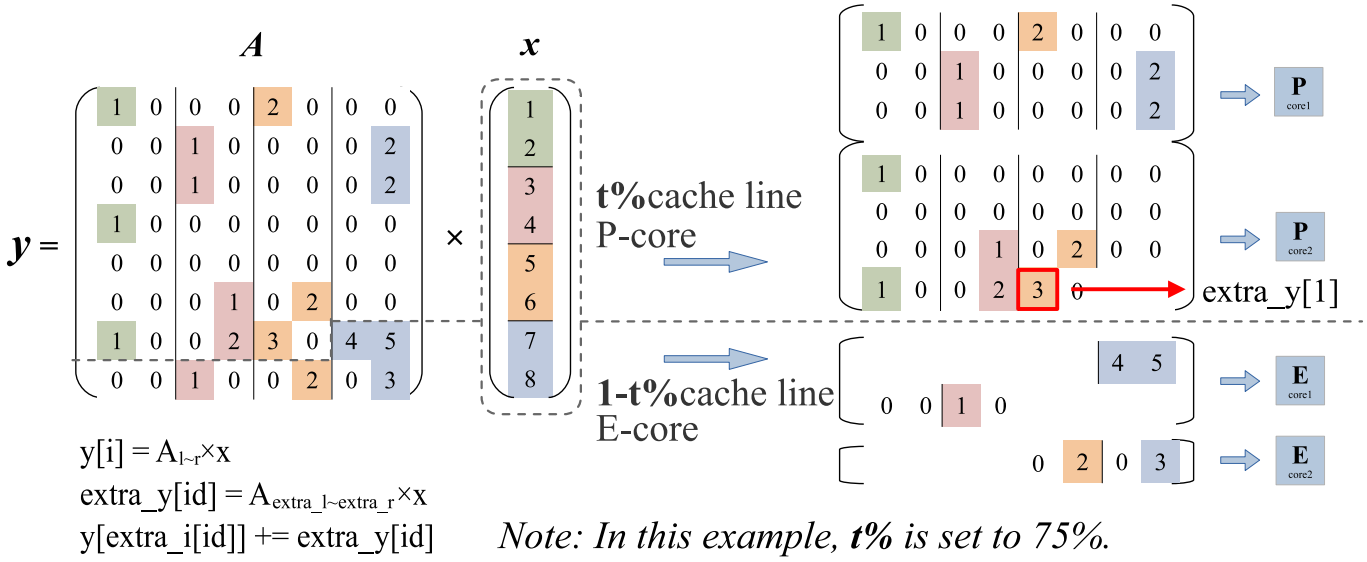


Fig. 7 An example of dividing an 8-by-8 sparse matrix by cache line (take Intel AMPs as an example, assuming two P-cores and two E-cores are used). Nonzero digits represent the number of the cache lines and where they are located. In the example, the cache line size is 16. The matrix is partitioned into two parts for P-cores and E-cores by the ratios of 75% to 25% (size 12 and 4, respectively).

TABLE I
The specifications of the systems tested.

Processor	12th-Gen Intel Core i9-12900KF		13th-Gen Intel Core i9-13900KF		AMD Ryzen 9 7950X		AMD Ryzen 9 7950X3D	
Core type	P-core	E-core	P-core	E-core	CCD1	CCD2	CCD1	CCD2
number of cores	8	8	8	16	8	8	8	8
L1 data cache	8×48KB	8×32KB	8×48KB	16×32KB	8×32KB	8×32KB	8×32KB	8×32KB
L2 cache	8×1.25MB	2×2MB	8×2MB	4×4MB	8×1MB	8×1MB	8×1MB	8×1MB
L3 cache	30MB		36MB		1×32MB	1×32MB	32MB+64MB	1×32MB
DRAM	DDR5 4800 MT/s		DDR5 5600 MT/s		DDR5 4800 MT/s		DDR5 4800 MT/s	
DRAM capacity	32 GB		32 GB		32 GB		32 GB	
Base/Max Frequency	3.2/5.1 GHz	2.4/3.9 GHz	3.0/5.4 GHz	2.2/4.3 GHz	4.2/5.7 GHz		4.2/5.7 GHz	
Operating system	Ubuntu 22.10				Ubuntu 22.04			
Compiler	gcc 12.2.0				gcc 11.3.0			
Vendor math library	Intel oneMKL v2023.0				AOCL−Sparse version 4.0.0			

Algorithm 5 A pseudocode for computational part of HASpMV

Require: $CORE\_P$, $CORE\_E$, $nnz\_l[\ ]$, $nnz\_r[\ ]$, $csrRowPtr[\ ]$, $csrColIdx[\ ]$, $csrVal[\ ]$, $pl[\ ]$, $pr[\ ]$, $x[\ ]$
Ensure: $y[\ ]$
Initialize $extra\_y[\ ]$ to be an array of size $CORE\_P + CORE\_E$ initialized with 0
Set number of threads for parallelization
#pragma omp parallel for
for $id = 0$ to $CORE\_P + CORE\_E - 1$ **do**
  if $pl[id] = pr[id]$ **then**
    $extra\_y[id] \leftarrow avx2\_kernel(id, nnz\_l[id], nnz\_r[id], csrColIdx, csrVal, x)$
    continue
  end if
  $y[pl[id]] \leftarrow avx2\_kernel(id, nnz\_l[id], csrRowPtr[pl[id] + 1], csrColIdx, csrVal, x)$
  for $i = pl[id] + 1$ to $pr[id] - 1$ **do**
    $y[i] \leftarrow avx2\_kernel(id, csrRowPtr[i], csrRowPtr[i + 1], csrColIdx, csrVal, x)$
  end for
  $extra\_y[id] \leftarrow avx2\_kernel(id, csrRowPtr[pr[id]], nnz\_r[id], csrColIdx, csrVal, x)$
end for
for $id = 0$ to $CORE\_P + CORE\_E - 1$ **do**
  $y[pl[id]] \leftarrow y[pl[id]] + extra\_y[id - 1]$
end for

To further improve the algorithm performance, we also utilize the AVX2 and loop unrolling technologies in the implementation which is shown in Algorithm 6. Specifically, the AVX2 intrinsic $\_mm256\_loadu\_pd$ is used to load four nonzero double values from the input array $csrVal$ into a 256-bit vector val. Meanwhile, the AVX2 intrinsic $\_mm256\_set\_pd$ is used to set the four corresponding values from the input array x to be operated on in the vector vec. The AVX2 intrinsic $\_mm256\_fmadd\_pd$ is then used to perform the multiplication between val and vec, and add the result to the accumulator vector res. This process is repeated until all the nonzero double values in the given range are processed. Finally, a single horizontal addition using $\_mm256\_hadd\_pd$ is used to complete the calculation, and the resulting scalar value is returned. In the above operations, HASpMV can avoid data collisions to ensure accuracy and further improve performance. In the above operations, HASpMV can avoid data conflicts to ensure accuracy and quickly calculate the value of vector y.

Algorithm 6 The AVX2 and loop unrolling technologies function for HASpMV

```
function AVX2_KERNEL(id, begin, end, csrColIdx, csrVal, x)
    Determine the core type according to id and determine the threshold Len
    length ← end − begin
    if length < 4 then
        sum ← 0
        for j = begin to end − 1 do
            sum ← sum + csrVal[j] × x[csrColIdx[j]]
        end for
        return sum
    end if
    remainder ← 0
    res_y ← 0
    res ← _MM256_SETZERO_PD
    j ← begin
    if length < Len then
        remainder ← length%4
        loop ← length/4
        for i = 0 to loop − 1 do
            val ← _MM256_LOADU_PD(&csrVal[j])
            vec ← _MM256_SET_PD(x[csrColIdx[j + 3]], x[csrColIdx[j + 2]], x[csrColIdx[j + 1]], x[csrColIdx[j]])
            res ← _MM256_FMADD_PD(val, vec, res)
            j ← j + 4
        end for
    else
        loop ← length/8
        remainder ← length%8
        for i = 0 to loop − 1 do
            val ← _MM256_LOADU_PD(&csrVal[j])
            vec ← _MM256_SET_PD(x[csrColIdx[j + 3]], x[csrColIdx[j + 2]], x[csrColIdx[j + 1]], x[csrColIdx[j]])
            res ← _MM256_FMADD_PD(val, vec, res)
            j ← j + 4
            Repeat the previous four lines                    ▷ loop unrolling
        end for
    end if
    res ← _MM256_HADD_PD(res, res)
    res_y ← res[0] + res[2]
    for jj = end − remainder to end − 1 do
        res_y ← res_y + csrVal[jj] × x[csrColIdx[jj]]
    end for
    return res_y
end function
```

V. Experiment

A. Experimental Setup

We conducted our tests on three different platforms, which include the latest flagship Intel 12th- and 13th-Gen Core i9-12900KF and i9-13900KF CPUs with P- and E-cores respectively, as well as the latest AMD Ryzen 9 7950X3D CPU with CCD0 and CCD1. The specifications of these platforms are provided in Table I, while their block diagrams are illustrated in Figure 2. Additionally, we also utilized an AMD Ryzen 9 7950X CPU, the details of which are also listed in Table I.

For the AMD Ryzen 9 7950X CPU, we set its frequency to be the same as that of the AMD Ryzen 9 7950X3D. The remaining settings were kept consistent between both generations of processors, with the only difference being that the L3 cache of CCD0 in the AMD Ryzen 9 7950X3D features a 3D V-Cache, resulting in an additional capacity of 64MB compared to the L3 cache of CCD1. Therefore, we conducted a set of experiments to compare the performance of the AMD Ryzen 9 7950X and the AMD Ryzen 9 7950X3D, considering the AMD Ryzen 9 7950X as the homogeneous processor form of the AMD Ryzen 9 7950X3D.

It is important to note that the installed Ubuntu 22.10 operating system supports a Linux version of Intel thread director, which may provide better scheduling for OpenMP programs on Intel AMPs, potentially leading to improved performance.

In addition, we utilize the entire dataset from the SuiteSparse Matrix Collection [30], which consists of 2888 matrices, as our primary dataset. Furthermore, to provide a more comprehensive analysis, we select 22 representative matrices from this collection, which are also partially used in existing works [14], [16], [21]. These representative matrices are listed in Table II. These matrices exhibit varying sizes (indicated by the '#rows' and 'Nnz' items) and sparse structures (indicated by the 'Nnz per row' item). They are widely used in diverse fields and serve as good representatives for studying and evaluating the performance of different algorithms and techniques in sparse matrix computations.

TABLE II
The 22 representative sparse matrices.

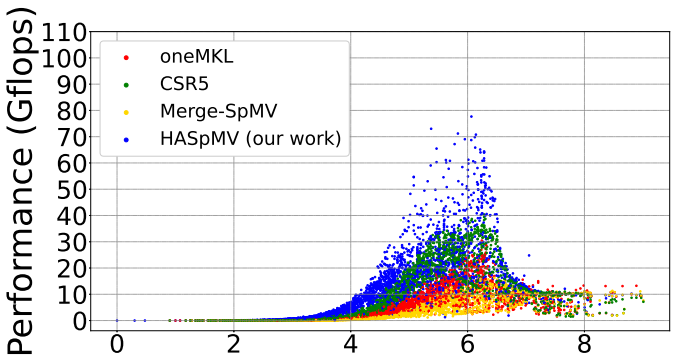
Name	#rows	Nnz	Nnz per row (min, avg, max)
consph	83K	6.0M	1, 72, 81
Ga41As41H72	268K	18.5M	18, 68, 702
conf5_4−8×8-10	49K	1.9M	39, 39, 39
webbase−1M	1M	3.1M	1, 3, 4.7K
cop20k_A	121K	2.6M	0, 21, 81
in−2004	1.4M	16.9M	0, 12, 7.8K
pdb1HYS	36K	4.3M	18, 119, 204
ASIC_680k	683K	3.9M	1, 6, 395K
Si41Ge41H72	186K	15.0M	13, 80, 662
circuit5M	5.6M	59.5M	1, 10, 1.29M
rma10	47K	2.4M	4, 50, 145
FullChip	2.9M	26.6M	1, 9, 2.3M
mip1	66K	10.4M	4, 155, 66.4K
mac_econ_fwd500	207K	1.3M	1, 6, 44
cant	62K	4.0M	1, 64, 78
dc2	117K	766K	1, 7, 114K
shipsec1	141K	7.8M	24, 55, 102
n4c6−b7	163K	1.3M	8, 8, 8
Dubcova2	65K	1M	4, 15, 25
viscorocks	37.8K	1.1M	16, 30, 42
dawson5	51K	1M	1, 19, 33
G_n_pin_pout	100K	1M	0, 10, 25

We compare our HASpMV with four other existing SpMV implementations: (1) Compared only on Intel AMPs, the inspector-executor functions *mkl_sparse_set_mv_hint* and *mkl_sparse_d_mv* in the newest vendor-supported Intel oneMKL (oneAPI Math Kernel Library) version 2023.0, (2) compared only on AMD AMPs, the *AOCL-Sparse* functions in the newest vendor-supported AMD AOCL (AMD Optimizing CPU Libraries version 4.0.0, (3) the AVX2 version of the open-source CSR5-format SpMV [14], and (4) the open-source Merge-SpMV method using a minor variant of the CSR format [15]. Noted that all tests are in double precision.

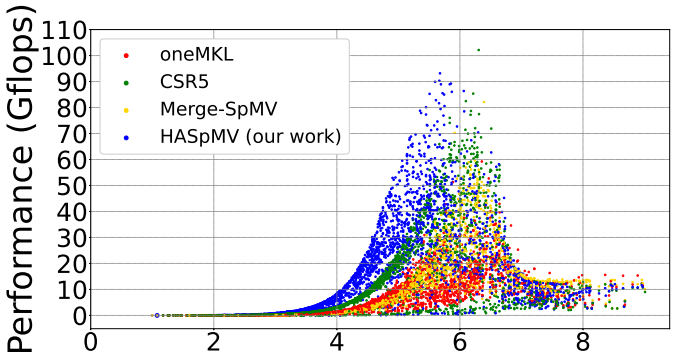
B. SpMV Performance Comparison

We show the performance results of our HASpMV and other SpMV works in Figure 8. The experimental results show that compared to the latest version of the Intel oneMKL library

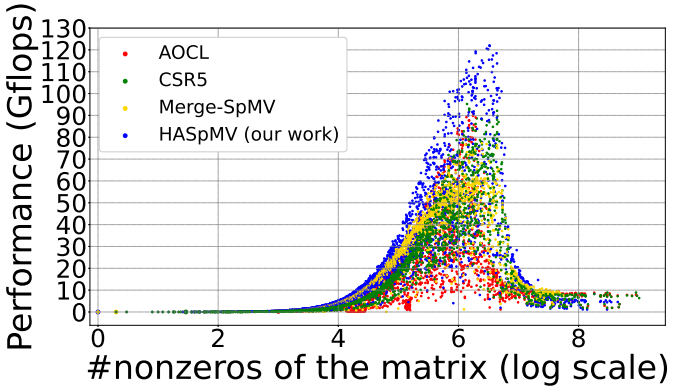
and the open-source works CSR5 and merge-SpMV, HASpMV achieves an average speedup of 2.61x, 2.31x, and 3.73x (up to 5.23x, 4.46x, and 8.23x) on the i9-12900KF processor. On the i9-13900KF processor, HASpMV achieves an average speedup of 3.17x, 1.52x, and 2.23x (up to 9.46x, 5.31x, and 4.49x). Additionally, when comparing AMD Ryzen 9 7950X3D and 7950X AMPs, HASpMV brings an average speedup of 1.43x, 1.3x, and 1.29x (up to 6.28x, 7.8x, and 10.8x) over AMD Optimizing CPU Libraries (AOCL), CSR5, and merge-SpMV, respectively.



(a) i9-12900KF



(b) i9-13900KF



(c) AMD Ryzen 9 7950X3D

Fig. 8 Performance comparison of oneMKL or AOCL, CSR5, Merge-SpMV and our HASpMV algorithm.

For more detailed analyses, in figure 11, we demonstrate the SpMV performance of the 22 representative matrices. As can be seen, for the matrices with evenly distributed nonzeros, such as 'conf5_4−8×8−10' having 39 nonzeros in each row, our method can be 1.84x faster than the second fastest method CSR5. This is because that HASpMV considers heterogeneity of AMPs and achieve better load balancing at runtime. As for the matrices with diverse caching costs of the rows, such as the matrix 'rma10', our HASpMV is more than 2x faster than CSR5 and Merge-SpMV. The reason is that the number of cache lines accessed of each row is taken into consideration in our task assignment policies. For the matrices with power-law characteristics, such as 'webbase-1M' and 'FullChip', our method still can obtain comparable performance over the others, it is due to the fact that load balancing is also a key optimization in CSR5 and Merge-SpMV, and the cache behavior of the matrices brings relatively less impact.

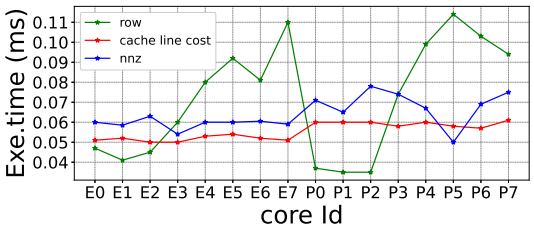
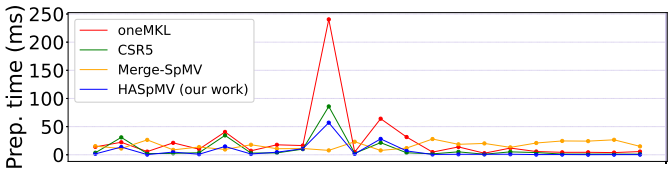
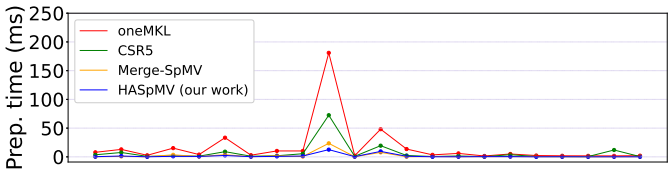


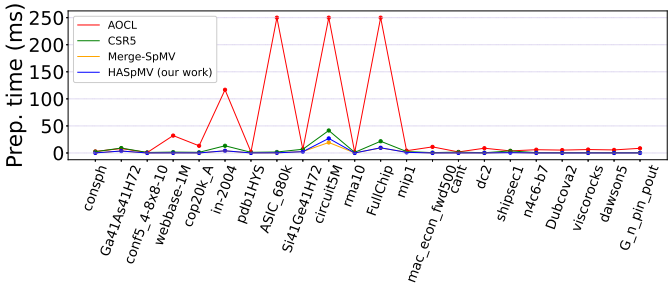
Fig. 9 Execution time of each core on i9-12900KF when matrix 'rma10' is partitioned by row, by nnz and by cache line, respectively.



(a) Intel Core i9-12900KF



(b) Intel Core i9-13900KF



(c) AMD Ryzen 9 7950X3D

Fig. 10 Comparison of the preprocessing time of the 22 matrices before running SpMV. Note: For illustrative purposes, AOCL's processing time of 250ms on subfigure (c) represents a very large preprocessing time (over 10,000ms) rather than a real 250ms.

To detail the balance of our partitioning strategy, the exe-

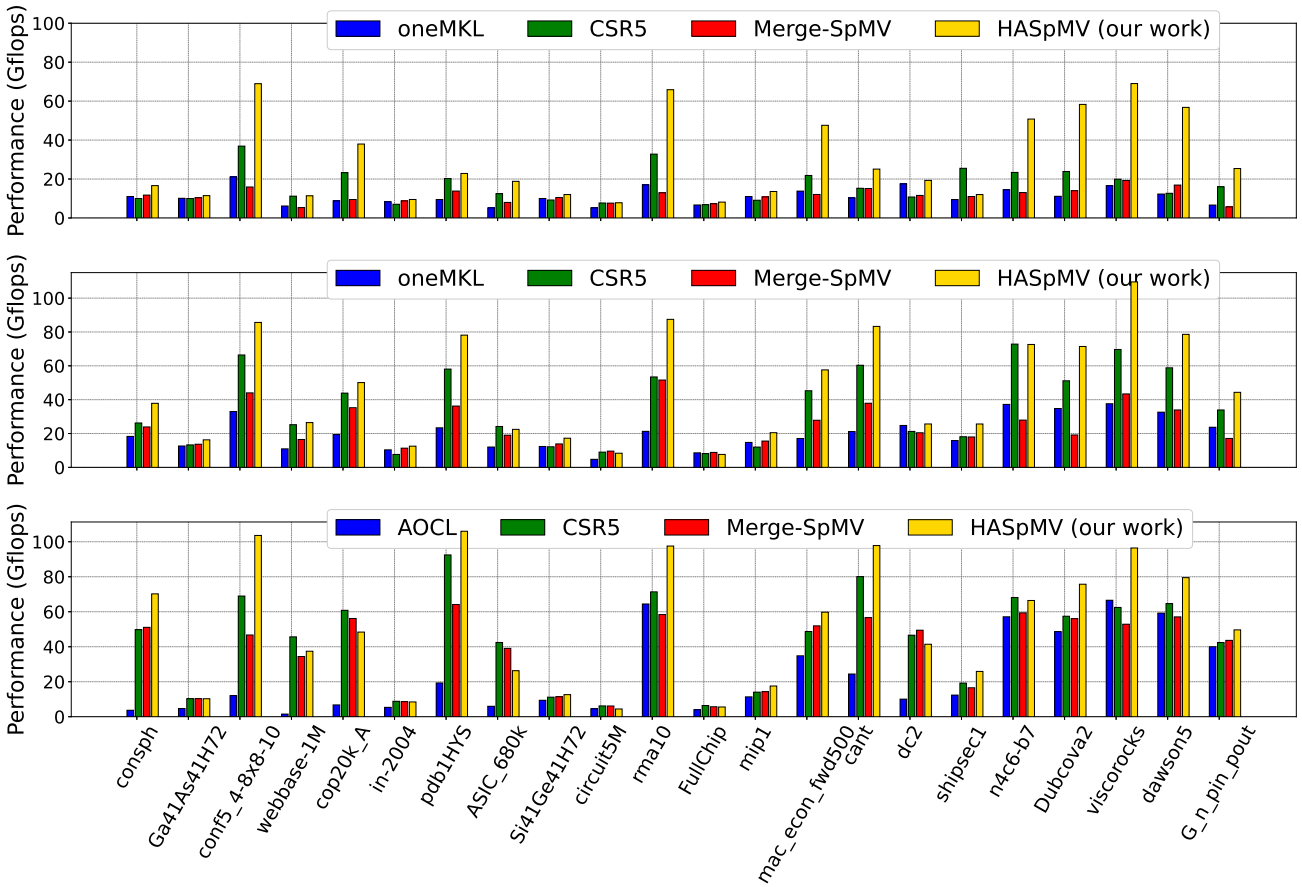


Fig. 11 Performance of the 22 representative matrices on i9-12900KF (top), i9-13900KF (middle) and AMD (bottom).

cution time of each core on i9-12900KF with two different partitioning methods for matrix 'rma10' is shown in Figure 9. It can be seen that with row partitioning, the execution time between cores varies greatly and is not balanced, and the partition according to nnz also has a certain gap in the load, while our method stabilizes between 0.05 and 0.06ms with cache line cost partitioning, and the load is very balanced.

C. Preprocessing Overhead Analysis

All the four SpMV methods need to analyze the sparsity structure of the input matrix, and generate auxiliary arrays if needed before calculation. We record and plot their preprocessing costs in Figure 10. It can be observed that the preprocessing time of HASpMV is almost always the lowest on i9-13900KF and 7950X3D (In addition, in the preprocessing of AOCL, its $aoclsparse\_optimize$ function takes too long to process some matrices, even more than 10,000ms. For the convenience of presentation, we show them uniformly as 250ms, rather than the real 250ms) , and is in general lower than the other methods on i9-12900KF (except for several matrices with very uneven nonzero distribution, in such case an reordering is needed). This advantage is mainly from that we keep the CSR format unchanged and only add a little information in HACSR for guiding the task assignment.

VI. Related Work

There is much research optimized SpMV on homogeneous parallel processors and mainly focused on the following directions: (1) **Reducing memory footprint of sparse matrix.** The memory footprint of sparse matrices can be effectively reduced by studying and using different compression techniques and storage formats, as described in [9], [10], [17], [36]–[41]. (2) **Increasing data locality of vector access.** To reduce data access overhead and improve computational performance, researchers have explored ways to improve data locality for vector access in works such as [9]–[11], [13], [16], [17], [37]–[39], [42]–[56]. (3) **Utilizing wide SIMD units.** Many works [13], [14], [21], [45], [50]–[52], [57]–[61] can show that the use of SIMD units is an efficient way to optimize the performance of the SpMV algorithm. (4) **Improving load balancing.** To resolved the load imbalance problem, researchers have proposed various load balancing strategies including dynamic task allocation, adaptive scheduling, and data partitioning methods in papers such as [14], [15], [28], [45], [57], [60], [62]–[67]. (5) **Selecting the best format and algorithm via machine learning.** In works such as [12], [68]–[74], researchers have used machine learning techniques to select the best sparse matrix format and algorithm. This approach further improves SpMV performance by analyzing

the characteristics of the input matrix and computational resources to automatically determine the most suitable storage format and computation method for the task.

Integrating heterogeneous cores into a single chip is getting more and more attention in modern architecture design. Kumar et al. demonstrated that single-ISA AMPs could deliver significant energy benefits [23] and performance improvements [25]. Power et al. [24] proposed new cache coherence policies for integrated CPU-GPU AMPs. On the system level, Yu et al. [75] developed a heterogeneity-aware OS scheduler, Chen et al. [76] designed energy-efficient task mapping and resource management policies, and Niu et al. [26] developed asymmetry-aware locking scheme. On the application side, Balakrishnan et al. [77] discussed the performance impact from AMPs, and Saez et al. [27] developed methods that can migrate data-parallel OpenMP codes to fit AMPs.

In comparison, there lacks parallel algorithms, in particular irregular methods, designed for AMPs. Our another recent work called HASpGEMM [32] optimized SpGEMM on AMPs, and to our knowledge, HASpMV is the first parallel sparse matrix algorithm dedicated to utilizing single-ISA AMPs for SpMV, and achieved significant better performance over the work designed for homogeneous processors.

VII. Conclusion

We in this paper have proposed the HASpMV algorithm and the HACSR sparse format for SpMV on modern AMPs. Our approach understood key performance characteristics of P- and E-cores of Intel (CCD0 and CCD1 of AMD) through micro-benchmarking, and developed techniques for better cache locality and load balancing. The experiments on three latest Intel and AMD AMPs showed significant speedups over existing works.

ACKNOWLEDGMENTS

We deeply appreciate the invaluable comments from all the reviewers. Weifeng Liu is the corresponding author of this paper. This research was supported by the National Natural Science Foundation of China under Grant No.61972415.

References

[1] R. Li and Y. Saad, "GPU-Accelerated Preconditioned Iterative Linear Solvers," *The Journal of Supercomputing*, vol. 63, pp. 443–466, 2013.

[2] Y. M. Tsai, T. Cojean, and H. Anzt, "Sparse Linear Algebra on AMD and NVIDIA GPUs–The Race is on," in *ISC '20*, 2020.

[3] M. Naumov, "Parallel Solution of Sparse Triangular Linear Systems in The Preconditioned Iterative Methods on The GPU," *NVIDIA Corp., Westford, MA, USA, Tech. Rep.*, 2011.

[4] T. G. Mattson, C. Yang, S. McMillan, A. Buluç, and J. E. Moreira, "GraphBLAS C API: Ideas for Future Versions of The Specification," in *HPEC '17*, 2017.

[5] J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke, S. McMillan, C. Yang, J. D. Owens, M. Zalewski, T. Mattson, and J. Moreira, "Mathematical foundations of the GraphBLAS," in *HPEC '16*, 2016.

[6] C. Yang, A. Buluç, and J. D. Owens, "Implementing Push-Pull Efficiently in GraphBLAS," in *ICPP '18*, 2018.

[7] W. Liu, "Parallel and Scalable Sparse Basic Linear Algebra Subprograms," Ph.D. dissertation, University of Copenhagen, 2015.

[8] I. S. Duff, M. A. Heroux, and R. Pozo, "An Overview of the Sparse Basic Linear Algebra Subprograms: The New Standard from the BLAS Technical Forum," *ACM Transactions on Mathematical Software*, vol. 28, no. 2, pp. 239–267, 2002.

[9] A. Buluç, S. Williams, L. Oliker, and J. Demmel, "Reduced-Bandwidth Multithreaded Algorithms for Sparse Matrix-Vector Multiplication," in *IPDPS '11*, 2011.

[10] K. Kourtis, V. Karakasis, G. Goumas, and N. Koziris, "CSX: An Extended Compression Format for Spmv on Shared Memory Systems," in *PPoPP '11*, 2011.

[11] A. N. Yzelman and R. H. Bisseling, "Cache-Oblivious Sparse Matrix-Vector Multiplication by Using Sparse Matrix Partitioning Methods," *SIAM Journal on Scientific Computing*, vol. 31, no. 4, pp. 3128–3154, 2009.

[12] Z. Du, J. Li, Y. Wang, X. Li, G. Tan, and N. Sun, "AlphaSparse: Generating High Performance SpMV Codes Directly from Sparse Matrices," in *SC '22*, 2022.

[13] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop, "A Unified Sparse Matrix Data Format for Efficient General Sparse Matrix-Vector Multiplication on Modern Processors with Wide SIMD Units," *SIAM Journal on Scientific Computing*, vol. 36, no. 5, pp. C401–C423, 2014.

[14] W. Liu. and B. Vinter, "CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication," in *ICS '15*, 2015.

[15] D. Merrill and M. Garland, "Merge-Based Parallel Sparse Matrix-Vector Multiplication," in *SC '16*, 2016.

[16] Y. Niu, Z. Lu, M. Dong, Z. Jin, W. Liu, and G. Tan, "TileSpMV: A Tiled Algorithm for Sparse Matrix-Vector Multiplication on GPUs," in *IPDPS '21*, 2021.

[17] S. Yan, C. Li, Y. Zhang, and H. Zhou, "yaSpMV: Yet Another SpMV Framework on GPUs," in *PPoPP '14*, 2014.

[18] Y. Lu and W. Liu, "DASP: Specific Dense Matrix Multiply-Accumulate Units Accelerated General Sparse Matrix-Vector Multiplication," in *SC '23*, 2023.

[19] J. Gao, W. Ji, Z. Tan, Y. Wang, and F. Shi, "TaiChi: A Hybrid Compression Format for Binary Sparse Matrix-Vector Multiplication on GPU," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 12, pp. 3732–3745, 2022.

[20] E. Karimi, N. B. Agostini, S. Dong, and D. Kaeli, "VCSR: An Efficient GPU Memory-Aware Sparse Format," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 12, pp. 3977–3989, 2022.

[21] C. Gómez, F. Mantovani, E. Focht, and M. Casas, "Efficiently Running SpMV on Long Vector Architectures," in *PPoPP '21*, 2021.

[22] C. Gómez, F. Mantovani, E. Focht, and M. Casas, "HPCG on Long-Vector Architectures: Evaluation and Optimization on NEC SX-Aurora and RISC-V," *Future Generation Computer Systems*, vol. 143, pp. 152–162, 2023.

[23] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen, "Single-ISA Heterogeneous Multi-Core Architectures: the Potential for Processor Power Reduction," in *MICRO '03*, 2003.

[24] J. Power, A. Basu, J. Gu, S. Puthoor, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood, "Heterogeneous System Coherence for Integrated CPU-GPU Systems," in *MICRO '13*, 2013.

[25] R. Kumar, D. Tullsen, P. Ranganathan, N. Jouppi, and K. Farkas, "Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance," in *ISCA '04*, 2004.

[26] N. Liu, J. Gu, D. Tang, K. Li, B. Zang, and H. Chen, "Asymmetry-Aware Scalable Locking," in *PPoPP '22*, 2022.

[27] J. C. Saez, F. Castro, and M. Prieto, "Enabling Performance Portability of Data-Parallel OpenMP Applications on Asymmetric Multicore Processors," in *ICPP '20*, 2020.

[28] J. L. Greathouse and M. Daga, "Efficient sparse matrix-vector multiplication on gpus using the csr storage format," in *SC '14*, 2014.

[29] J. D. McCalpin, "Memory Bandwidth and Machine Balance in Current High Performance Computers," *IEEE computer society technical committee on computer architecture*, vol. 2, no. 19-25, 1995.

[30] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Transactions on Mathematical Software*, vol. 38, no. 1, pp. 1–25, 2011.

[31] F. Zhang, J. Zhai, B. He, S. Zhang, and W. Chen, "Understanding Co-Running Behaviors on Integrated CPU/GPU Architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 3, pp. 905–918, 2016.

[32] H. Cheng, W. Li, Y. Lu, and W. Liu, "HASpGEMM: Heterogeneity-Aware Sparse General Matrix-Matrix Multiplication on Modern Asymmetric Multicore Processors," in *ICPP '23*, 2023.

[33] W. Liu and B. Vinter, "Speculative Segmented Sum for Sparse Matrix-vector Multiplication on Heterogeneous Processors," *Parallel Computing*, vol. 49, pp. 179–193, 2015.

[34] A. Li, W. Liu, M. R. B. Kristensen, B. Vinter, H. Wang, K. Hou, A. Marquez, and S. L. Song, "Exploring and Analyzing the Real Impact of Modern On-Package Memory on HPC Scientific Kernels," in *SC '17*, 2017.

[35] F. Zhang, W. Liu, N. Feng, J. Zhai, and X. Du, "Performance Evaluation and Analysis of Sparse Matrix and Graph Kernels on Heterogeneous Processors," *CCF Transactions on High Performance Computing*, vol. 50, pp. 36–77, 2019.

[36] J. R. Rice and R. F. Boisvert, *Solving Elliptic Problems Using ELLPACK*. Springer Science & Business Media, 1984.

[37] E.-J. Im, "Optimizing the Performance of Sparse Matrix-Vector Multiplication," Ph.D. dissertation, University of California, Berkeley, 2000.

[38] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms," *Parallel Computing*, pp. 1–12, 2009.

[39] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel Sparse Matrix-Vector and Matrix-Transpose-Vector Multiplication Using Compressed Sparse Blocks," in *SPAA '09*, 2009.

[40] J. W. Choi, A. Singh, and R. W. Vuduc, "Model-Driven Autotuning of Sparse Matrix-Vector Multiply on GPUs," in *PPoPP '10*, 2010.

[41] Y. Zhang, S. Li, F. Yuan, D. Dong, X. Yang, T. Li, and Z. Wang, "Memory-aware Optimization for Sequences of Sparse Matrix-Vector Multiplications," in *IPDPS '23*, 2023.

[42] R. W. Vuduc, "Automatic Performance Tuning of Sparse Matrix Kernels," Ph.D. dissertation, University of California, Berkeley, 2003.

[43] R. Vuduc, J. Demmel, and K. Yelick, "OSKI: A Library of Automatically Tuned Sparse Matrix Kernels," *Journal of Physics: Conference Series*, vol. 16, no. 1, p. 521, 2005.

[44] A. N. Yzelman and R. H. Bisseling, "Two-Dimensional Cache-Oblivious Sparse Matrix-Vector Multiplication," *Parallel Computing*, vol. 37, no. 12, pp. 806–819, 2011.

[45] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, "Efficient Sparse Matrix-vector Multiplication on x86-based Many-core Processors," in *ICS '13*, 2013.

[46] M. Martone, "Efficient Multithreaded Untransposed, Transposed or Symmetric Sparse Matrix-Vector Multiplication with the Recursive Sparse Blocks Format," *Parallel Computing*, vol. 40, no. 7, pp. 251–270, 2014.

[47] P. Guo, L. Wang, and P. Chen, "A Performance Modeling and Optimization Analysis Tool for Sparse Matrix-Vector Multiplication on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 5, pp. 1112–1123, 2013.

[48] A. N. Yzelman and D. Roose, "High-Level Strategies for Parallel Shared-Memory Sparse Matrix-Vector Multiplication," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 1, pp. 116–125, 2013.

[49] A. Ashari, N. Sedaghati, J. Eisenlohr, and P. Sadayappan, "A Model-Driven Blocking Strategy for Load Balanced Sparse Matrix-Vector Multiplication on GPUs," *Journal of Parallel and Distributed Computing*, vol. 76, pp. 3–15, 2015.

[50] A. Elafrou, G. Goumas, and N. Koziris, "Performance Analysis and Optimization of Sparse Matrix-Vector Multiplication on Modern Multi- and Many-Core Processors," in *ICPP '17*, 2017.

[51] B. Xie, J. Zhan, X. Liu, W. Gao, Z. Jia, X. He, and L. Zhang, "CVR: Efficient Vectorization of SpMV on x86 Processors," in *CGO '18*, 2018.

[52] A. Elafrou, V. Karakasis, T. Gkountouvas, K. Kourtis, G. Goumas, and N. Koziris, "SparseX: A Library for High-Performance Sparse Matrix-Vector Multiplication on Multicore Platforms," *ACM Transactions on Mathematical Software*, vol. 44, no. 3, pp. 1–32, 2018.

[53] A. Elafrou, G. Goumas, and N. Koziris, "Conflict-Free Symmetric Sparse Matrix-Vector Multiplication on Multicore Architectures," in *SC '19*, 2019.

[54] C. Alappat, A. Basermann, A. R. Bishop, H. Fehske, G. Hager, O. Schenk, J. Thies, and G. Wellein, "A Recursive Algebraic Coloring Technique for Hardware-Efficient Symmetric Sparse Matrix-Vector Multiplication," *ACM Transactions on Parallel Computing*, vol. 7, no. 3, pp. 1–37, 2020.

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

[56] X. You, C. Liu, H. Yang, P. Wang, Z. Luan, and D. Qian, "Vectorizing SpMV by Exploiting Dynamic Regular Patterns," in *ICPP '22*, 2022.

[57] N. Bell and M. Garland, "Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors," in *SC '09*, 2009.

[58] B. Su and K. Keutzer, "clSpMV: A Cross-Platform OpenCL SpMV Framework on GPUs," in *ICS '12*, 2012.

[59] C. Lehnert, R. Berrendorf, J. P. Ecker, and F. Mannuss, "Performance Prediction and Ranking of SpMV Kernels on GPU Architectures," in *Euro-Par '16*, 2016.

[60] H. Anzt, T. Cojean, C. Yen-Chen, J. Dongarra, G. Flegar, P. Nayak, S. Tomov, Y. M. Tsai, and W. Wang, "Load-Balancing Sparse Matrix Vector Product Kernels on GPUs," *ACM Transactions on Parallel Computing*, vol. 7, no. 1, pp. 1–26, 2020.

[61] H. Bian, J. Huang, L. Liu, D. Huang, and X. Wang, "ALBUS: A Method for Efficiently Processing SpMV Using SIMD and Load Balancing," *Future Generation Computer Systems*, vol. 116, pp. 371–392, 2021.

[62] X. Yang, S. Parthasarathy, and P. Sadayappan, "Fast Sparse Matrix-vector Multiplication on GPUs: Implications for Graph Mining," *Proceedings of the VLDB Endowment*, 2011.

[63] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarath, and P. Sadayappan, "Fast Sparse Matrix-Vector Multiplication on GPUs for Graph Applications," in *SC '14*, 2014.

[64] M. Daga and J. L. Greathouse, "Structural Agnostic SpMV: Adapting CSR-Adaptive for Irregular Matrices," in *HiPC '15*, 2015.

[65] Y. Liang, W. T. Tang, R. Zhao, M. Lu, H. P. Huynh, and R. S. M. Goh, "Scale-Free Sparse Matrix-Vector Multiplication on Many-Core Architectures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 12, pp. 2106–2119, 2017.

[66] M. Steinberger, R. Zayer, and H. Seidel, "Globally Homogeneous, Locally Adaptive Sparse Matrix-vector Multiplication on the GPU," in *ICS '17*, 2017.

[67] H. Mi, X. Yu, X. Yu, S. Wu, and W. Liu, "Balancing Computation and Communication in Distributed Sparse Matrix-Vector Multiplication," in *CCGRID '23*, 2023.

[68] J. Li, G. Tan, M. Chen, and N. Sun, "SMAT: An Input Adaptive Auto-tuner for Sparse Matrix-vector Multiplication," in *PLDI '13*, 2013.

[69] N. Sedaghati, T. Mu, L. Pouchet, S. Parthasarathy, and P. Sadayappan, "Automatic Selection of Sparse Matrix Representation on GPUs," in *ICS '15*, 2015.

[70] A. Benatia, W. Ji, Y. Wang, and F. Shi, "Sparse Matrix Format Selection with Multiclass SVM for SpMV on GPU," in *ICPP '16*, 2016.

[71] Y. Zhao, W. Zhou, X. Shen, and G. Yiu, "Overhead-Conscious Format Selection for SpMV-Based Applications," in *IPDPS '18*, 2018.

[72] Y. Zhao, J. Li, C. Liao, and X. Shen, "Bridging the Gap Between Deep Learning and Sparse Matrix Format Selection," in *PPoPP '18*, 2018.

[73] G. Tan, J. Liu, and J. Li, "Design and Implementation of Adaptive SpMV Library for Multicore and Many-Core Architecture," *ACM Transactions on Mathematical Software*, vol. 44, no. 4, pp. 1–25, 2018.

[74] S. Yesil, A. Heidarshenas, A. Morrison, and J. Torrellas, "WISE: Predicting the Performance of Sparse Matrix Vector Multiplication with Machine Learning," in *PPoPP '23*, 2023.

[75] T. Yu, R. Zhong, V. Janjic, P. Petoumenos, J. Zhai, H. Leather, and J. Thomson, "Collaborative Heterogeneity-Aware OS Scheduler for Asymmetric Multicore Processors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 5, pp. 1224–1237, 2021.

[76] J. Chen, M. Manivannan, M. Abduljabbar, and M. Pericàs, "ERASE: Energy Efficient Task Mapping and Resource Management for Work Stealing Runtimes," *ACM Transactions on Architecture and Code Optimization*, vol. 19, no. 2, pp. 1–29, 2022.

[77] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai, "The Impact of Performance Asymmetry in Emerging Multicore Architectures," in *ISCA '05*, 2005.