Optimizing CSR-Based SpMV on a New MIMD Architecture Pezy-SC3s

Jihu Guo[1,2][0009−0002−9811−0234], Jie Liu[1,2], Qinglin Wang[1,2][0000−−0002−8286−6566], and Xiaoxiong Zhu[1,2]

[1] Science and Technology on Parallel and Distributed Processing Laboratory, National University of Defense Technology, Changsha 410073, China
[2] School of Computer Science, National University of Defense Technology, Changsha 410073, China
guojihu@nudt.edu.cn

Abstract. Sparse matrix-vector multiplication (SpMV) is extensively used in scientific computing and often accounts for a significant portion of the overall computational overhead. Therefore, improving the performance of SpMV is crucial. However, sparse matrices exhibit a sporadic and irregular distribution of non-zero elements, resulting in workload imbalance among threads and challenges in vectorization. To address these issues, numerous efforts have focused on optimizing SpMV based on the hardware characteristics of computing platforms. In this paper, we present an optimization on CSR-Based SpMV, since the CSR format is the most widely used and supported by various high-performance sparse computing libraries, on a novel MIMD computing platform Pezy-SC3s. Based on the hardware characteristics of Pezy-SC3s, we tackle poor data locality, workload imbalance, and vectorization challenges in CSR-Based SpMV by employing matrix chunking, applying Atomic Cache for workload scheduling, and utilizing SIMD instructions during performing SpMV. As the first study to investigate SpMV optimization on Pezy-SC3s, we evaluate the performance of our work by comparing it with the CSR-Based SpMV and SpMV provided by Nvidia's CuSparse. Through experiments conducted on 2092 matrices obtained from SuiteSparse, we demonstrate that our optimization achieves a maximum speedup ratio of x17.63 and an average of x1.56 over CSR-Based SpMV and an average bandwidth utilization of 35.22% for large-scale matrices ($nnz \geq 10^6$) compared with 36.17% obtained using CuSparse. These results demonstrate that our optimization effectively harnesses the hardware resources of Pezy-SC3s, leading to improved performance of CSR-Based SpMV.

Keywords: SpMV · Optimization · CSR-Based SpMV · Pezy-SC3s.

1 Introduction

Sparse matrix-vector multiplication (SpMV) is a fundamental operation denoted by $y = Ax$, where A is a sparse matrix and x and y are dense vectors. SpMV finds widespread application in high-performance computing domains such as graph

analysis, machine learning, deep learning, and more. Its significance extends to solving sparse linear systems, eigenvalue systems, Krylov subspace methods, and similar problems [4, 7, 8, 18, 20, 23, 28, 9, 24]. Given the impact of SpMV performance on these domains, improving the efficiency of sparse matrix-vector multiplication becomes crucial, as SpMV often dominates the computational overhead of related tasks.

However, optimizing SpMV poses significant challenges. Sparse matrices exhibit a sparse and irregular distribution of non-zero elements, with different matrices having distinct sparse patterns. Consequently, unbalanced workload distribution among threads and difficulty in vectorization. Numerous studies have introduced well-designed sparse matrix storage formats, including ELL, DIA, and BCSR, among others, tailored to different sparse patterns. Workload balancing approaches such as CSR5 [14], Merge-based CSR [16], yaSpMV [31], and HCC [13] have also been proposed to address workload imbalance between threads. Additionally, methods like CVR [30], SpV8 [12], ELL-R [25], ELLR-T [26], BiELL [32], have tackled the issue of limited vectorization. These endeavors have predominantly focused on CPU and GPU platforms. However, this paper presents SpMV optimization on a novel MIMD platform called Pezy-SC3s, as illustrated in Fig. 1. Pezy-SC3s comprises two Prefectures, each equipped with an

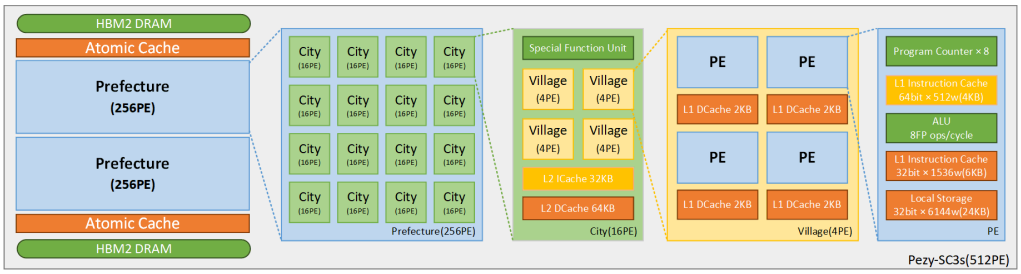

Fig. 1. Pezy-SC3s Block Diagram

Atomic Cache and an HBM2 module. The Atomic Cache facilitates atomic operations on internal data, and Table 1 presents the specific parameters associated with the Atomic Cache. Max Operations/Clock column refers to the maximum

Table 1. Parameters of Atomic Cache

Size (Chip Total)	WAY	Line size	Max Operations/Clock
1KB (16KB)	4	256B	32

32 operations per clock cycle that can be maintained when the number of op-

erations performed by the atomic cache in each clock cycle is not greater than this value. Otherwise, the performance will drop dramatically.

Each Prefecture encompasses 16 cities, and each city contains 4 villages. Within each city, there exists an L2 Cache that is shared by 128 threads. Additionally, a Special Function Unit is present in each city to handle division, modulo, and square root operations. Within each Village, there are 4 Processing Elements (PEs), each accompanied by an L1 Cache. The PEs house a Local Storage of up to 24KB, and the size of the Local Storage can be adjusted by modifying the stack space size of threads. Furthermore, Pezy-SC3s supports 128-bit SIMD instructions.

We have selected CSR-based SpMV as the fundamental basis for our optimization approach. There are several reasons for this choice. Firstly, no SpMVs have been specifically designed for Pezy-SC3s thus far. Secondly, CSR-based SpMV has been extensively utilized and established as benchmarks in numerous previous studies [14, 30, 12, 16, 31]. Lastly, CSR-based SpMV is also the supported SpMV in various high-performance scientific sparse computing libraries such as MKL [27] and CuSparse [3]. Please refer to Fig. 2 for an illustration of the CSR format. *rowDelimiters* stores the indexes range of non-zero elements

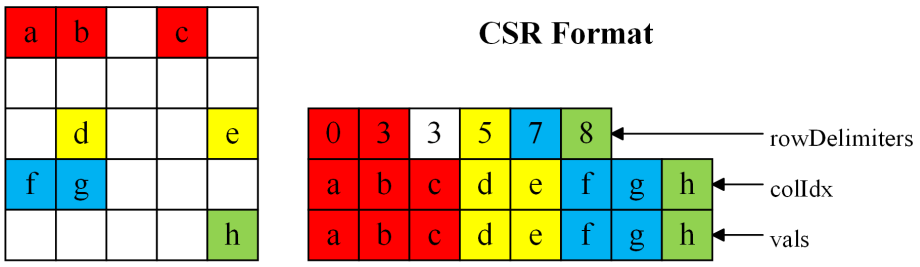

Fig. 2. CSR Format

of rows, and $colIdx$ and $vals$ store the corresponding column indexes and values of the non-zero elements.

The parallel CSR-Based SpMV algorithm on Pezy-SC3s is presented in Algorithm 1. Unlike Nvidia's SIMT GPUs, parallel CSR-Based SpMV on Pezy-SC3s necessitates a for-loop instruction to specify the operations required for each thread. In Fig. 1, each thread handles computations for a single row at a time and updates the corresponding values in the y array. Here, $rowNums$ and $threadNums$ represent the total number of rows in the matrix and the maximum number of threads supported by the hardware platform, respectively. However, this straightforward parallelization approach can result in workload imbalance among threads and limited data reuse in the cache, especially when the number of threads ($threadNums$) becomes excessively large, leading to row-based workload imbalance [17]. Furthermore, the parallel CSR-Based SpMV primar-

Algorithm 1 Parallel CSR-Based SpMV on Pezy-SC3s

Input: x, rowDelimiters, colIdx, vals;
Output: y;

```
1: gid = getgid(); // Get thread id.
2: for i = gid; i < rowNums; i += threadNums do
3:     sum = 0.0;
4:     for j = rowDelimiters[i]; j < rowDelimiters[i+1]; j ++ do
5:         sum += x[colIdx[j]] × vals[j];
6:     y[i] = sum;
7: flush_L2(); // Write back y by flush_L2.
8: return y;
```

ily focuses on inter-row parallelism, with all scalar operations performed within each row.

To address these issues, we optimize the CSR-Based SpMV based on hardware characteristics of Pezy-SC3s. Firstly, we introduce matrix chunking to enhance data locality. The number of matrix chunks is determined through careful testing of various chunking parameters. Additionally, we utilize the Atomic Cache to dynamically guide threads in fetching the next row to be computed, thereby achieving intra-block workload balance. Finally, we incorporate 128-bit SIMD instructions within each row to improve vectorization. By fully leveraging the hardware capabilities of Pezy-SC3s, we effectively mitigate issues related to poor data locality, workload imbalance, and difficult vectorization in CSR-Based SpMV.

Through extensive experiments conducted on 2092 matrices obtained from SuiteSparse [5], our optimization demonstrates remarkable results. We achieve a maximum speedup ratio of x17.63 compared to the CSR-Based SpMV. For small-scale matrices ($nnz < 10^6$), we achieve a maximum speedup ratio of x10.84 and an average speedup ratio of x1.24. Similarly, for large-scale matrices ($nnz \geq 10^6$), we achieve a maximum speedup ratio of x17.63 and an average speedup ratio of x2.97 (average speedup on the total 2092 matrices is x1.57). Moreover, our average bandwidth utilization on large-scale matrices reaches 35.22%, which can further increase to 62.54% when the matrix size becomes larger ($nnz \geq 10^7$). These figures demonstrate a satisfactory performance compared with the corresponding figures of 36.17% and 45.56% obtained using the Nvidia CuSparse library, highlighting the effective utilization of Pezy-SC3s hardware resources in our approach.

2 Related Work

Sparse matrices typically contain a small percentage (usually only 1%) of non-zero elements compared to the total number of elements in the matrix. To conserve space and enhance operation efficiency, we commonly utilize sparse matrix formats to store and perform operations solely on the non-zero elements. How-

ever, the distribution of these non-zero elements often exhibits irregular patterns. As a result, significant variations in the number of non-zero elements between rows arise, leading to issues such as unbalanced load across operating units and challenging vectorization during SpMV computations.

Various types of sparse matrices exhibit distinct sparse patterns, prompting numerous studies to propose novel storage formats tailored to different patterns. For instance, the ELL format compresses all elements to the left and employs two-dimensional arrays to store the values and column coordinates of the non-zero elements. The row coordinate in the array corresponds to the y coordinate for writing, and the length of each row depends on the maximum number of non-zero elements in the matrix. The DIA format arranges non-zero elements in a diagonal structure, using an $OFFSET$ array to record the offset of each column relative to the main diagonal. BCSR, on the other hand, adopts a blocked CSR format, storing non-zero elements in matrix blocks. It employs two arrays to track row offsets and column coordinates of these blocks, utilizing dense matrix operations within the blocks.

Efforts to address workload imbalance between threads have also been explored. CSR5 [14], for example, chunks the matrix into tiles based on the CSR format to ensure that each block contains an equal number of non-zero elements. These blocks are then evenly distributed among the threads. Merge-based CSR [17] not only prioritizes balancing the number of non-zero elements but also considers the load associated with writing back to the y array. It accomplishes this by merging the $rowDelimiters$ array of CSR into the $colIdx$ array, which is subsequently divided equally among threads to achieve workload balance. Another approach employed in yaSpMV [31] involves chunking matrices in a row- and column-insensitive manner, resulting in significant compression ratios. HCC [13] utilizes fixed-distance chunking in the column direction and further chunks based on the number of non-zero elements in the row direction. This strategy ensures both data locality of x and workload balance.

To tackle the challenge of insufficient vectorization, various techniques have been proposed. CVR [30] schedules tasks to each thread based on row granularity, enabling full utilization of SIMD instructions. SpV8 [12] separates vectorization and scalarization, maximizing vectorization operations for aligned non-zero elements and resorting to scalarization for non-aligned elements, thereby achieving enhanced vectorization. BiELL [32], an extension of the ELL format, further compresses ELL by employing folding techniques to avoid excessive padding resulting from excessively long lines with numerous zero elements.

These advancements have predominantly focused on CPU or GPU architectures. However, in this paper, we aim to optimize the widely used CSR-Based SpMV specifically for the Pezy-SC3s, a MIMD computing platform. Our goal is to address challenges associated with poor data localization, workload imbalance, and insufficient vectorization encountered during SpMV computations in the CSR-Based SpMV on Pezy-SC3s and fully utilize the MIMD computing resources.

Algorithm 2 Optimized Parallel CSR-Based SpMV on Pezy-SC3s

Input: x, rowDelimiters, colIdx, vals, blockDelimiters;
Output: y;

```
1: GLOBAL nextRowIdx[4096];
2: blockSize = 64;
3: gid = getgid(); // Get thread id.
4: beginRow = blockDelimiters[gid / blockSize], endRow = blockDelimiters[gid /
   blockSize + 1];
5: i = beginRow + gid % blockSize;
6: pz_atomic_store(&nextRowIdx[gid / blockSize × 64], beginRow + blockSize);
7: for i do = 0; i < endRow; i = pz_atomic_inc(&nextRowIdx[gid / blockSize]
8:     sum2 = {0.0, 0.0};
9:     eleBegin = rowDelimiters[i], eleEnd = rowDelimiters[i + 1];
10:    span = (eleEnd - eleBegin) & ∼ 1 ;
11:    tail = (eleBegin ^ eleEnd) & 1;
12:    for j do = eleBegin; j < eleEnd; j ++
13:        vv = {vals[j], vals[j+1]};
14:        vx = {x[colIdx[j]], x[colIdx[j+1]]};
15:        sum2 += vv × vx;
16:    y[i] = sum2.x + sum2.y + tail × (x[colIdx[eleEnd-1]] × vals[eleEnd-1]);
17: flush_L2(); // Write back y by flush_L2.
18: return y;
```

3 Methodology

SpMV is an operation that is memory-intensive and constrained by limited bandwidth. The inherent sparsity of the matrices introduces workload imbalance, limiting the efficient utilization of computing resources. Furthermore, the irregular distribution of non-zero elements in sparse matrices presents an additional challenge in achieving optimal hardware performance. The irregularity hinders the full vectorization of the operation[1, 15, 6, 2, 19, 10]. To address these challenges, we propose optimizations for CSR-Based SpMV for the Pezy-SC3s platform. Our approach involves chunking the matrix based on row granularity to enhance data locality, utilizing the Atomic Cache for efficient workload balance, and leveraging SIMD instructions to improve intra-row computational parallelism. The optimized algorithm is presented in Algorithm 2, and detailed explanations of the algorithm will be provided in the subsequent subsections.

3.1 Row Granularity Matrix Partition

Based on the hardware architecture of Pezy-SC3s, it is observed that there is no Last Level Cache (LLC). Instead, data is delivered directly to the L2 Cache through HBM2 when required by the PEs. As a result, the L2 Cache stores almost all the reusable data among threads. To improve data locality in the L2 Cache, we partition the matrix into chunks to make threads that share the same

L2 Data Cache to compute the adjacent rows, enabling threads to collaborate effectively.

The performance is tested with various matrix chunking numbers, and we select af_1_k101 and $F1$ matrices which have the sufficient number of non-zero elements to test the performance of SpMV to plot the results in Fig. 3. The findings demonstrate that SpMV performance improves with an increase in the number of matrix chunkings but sharply drops when the number becomes excessively high. The optimal performance is achieved with 64 chunkings, with 64 threads assigned to each chunking, respectively. These threads share the L2 Cache. Since the best performance is observed with 64 chunkings, it is chosen as the chunking number (Algorithm 2, line 2).

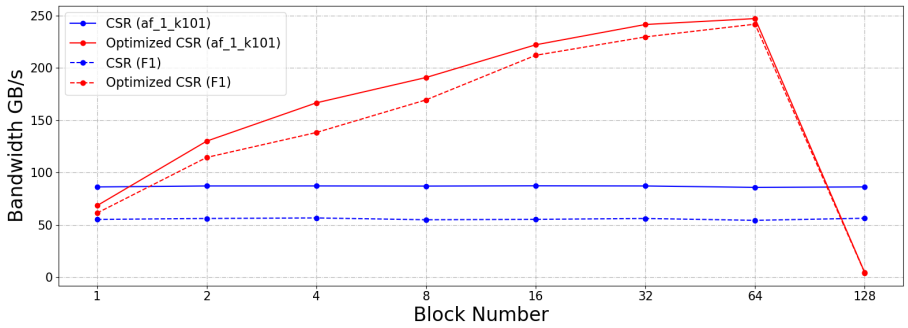

Fig. 3. Bandwidth obtained in different number of matrix chunks

During chunking, the number of non-zero elements in each row is determined using the $rowDelimiters$ array. The matrix is then divided into 64 blocks using row granularity division, aiming to ensure an approximately equal total number of non-zero elements in each block. Based on the $rowDelimiters$, the total number of non-zero elements in the range is calculated as $rowDelimiters[numRows] - rowDelimiters[0]$. The division point, denoted as m, corresponds to the maximum coordinate to make $rowDelimiters[m] - rowDelimiters[0]$ less than or equal to $(rowDelimiters[numRows] - rowDelimiters[0])/2$. Using m, the original range is divided into $[0, m)$ and $[m, numRows)$. The chunk with a higher number of non-zero elements, assuming it to be $[m, numRows)$, is further partitioned, and this step is repeated until 64 segments are obtained. Subsequently, each segment is assigned to the corresponding thread block in order. The chunking process is illustrated in Fig. 4. Finally, an array of $blockDelimiters$ with a length of 65 is obtained, where $blockDelimiters[i]$ represents the starting row of the matrix block assigned to the i^{th} thread group, and $blockDelimiters[i+1]$ represents the ending row of the matrix block assigned to the i^{th} thread group (line 3 of Algorithm 2).

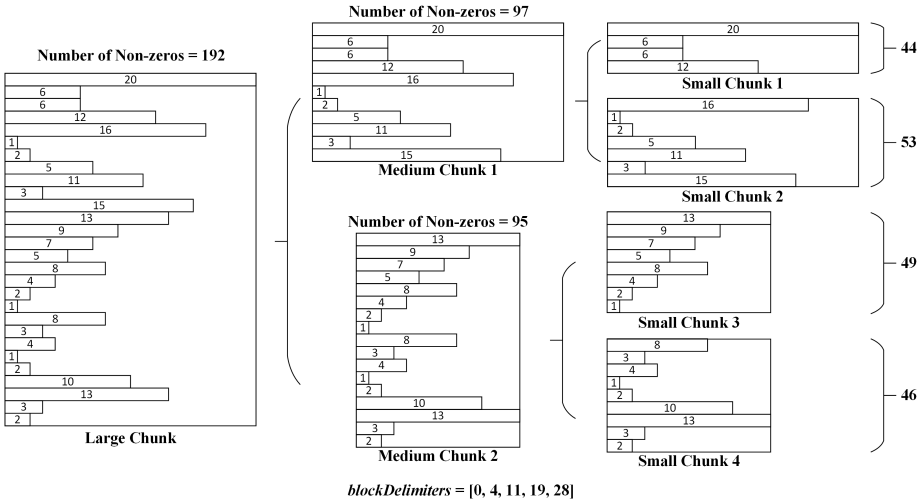

Fig. 4. In this matrix partition process, we first divide the matrix (Large Chunk) with 28 rows into 2 Medium Chunks. Then, divide Medium Chunk 1 since Medium Chunk 1 has more non-zero elements ($97 \geq 95$). Finally, divide Medium Chunk 2.

3.2 Workload Balance Within Matrix Chunks

During the chunking phase, the matrix is divided based on row granularity, with each chunk consisting of complete rows. When a thread is assigned a single row, it can directly write back the result once its computation is complete. However, when multiple threads are responsible for a row, they must wait for each other to finish their computations before performing the write-back. In this case, only one thread needs to write back, but multiple threads have to wait for this write-back operation. On Pezy-SC3s with 4096 threads, using multiple threads to compute a row introduces significant delays as threads wait for each other. Conversely, assigning a single thread to a row causes load imbalance within the Chunk. To address this issue, we utilize the atomic cache hardware in Pezy-SC3s, which has high performance and supports atomic operations, for single thread computing single row workload balance.

In Algorithm 1, $threadNums$ instructs the next row to be computed by a thread. For example, with 4096 threads, thread i would compute lines i, $i+4096$, $i+4096\times 2$, and so on. This approach results in workload imbalance, as depicted in Fig. 5(a). The red thread is assigned most of the total tasks, while the blue and yellow threads are assigned only a small percentage. Consequently, the yellow, blue, and green threads spend considerable time waiting for the red thread, leading to the underutilization of computing resources. By leveraging Atomic Cache, we can guide the computation order for each group of threads, ensuring a roughly equal distribution of tasks among threads and reducing the waiting time between them, thereby improving workload balance.

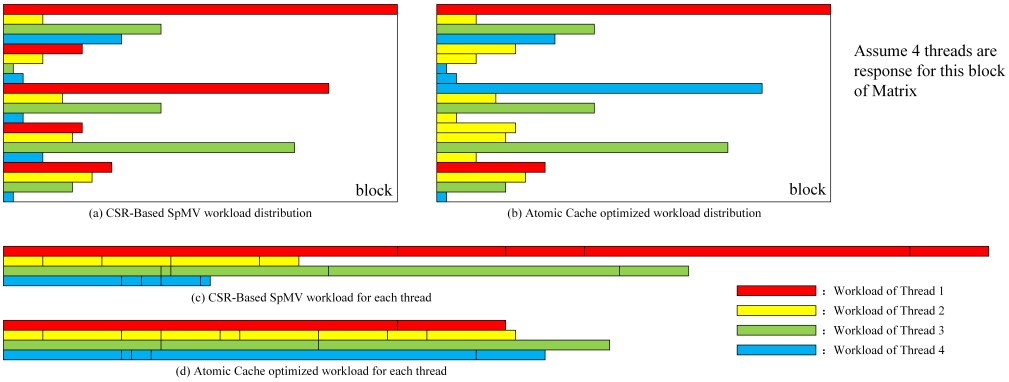

Fig. 5. Workload Assignment by Atomic Cache

The array $nextRowIdx$ (line 1 of Algorithm 2) is an int ($sizeof(int)$ = 4Byte) array of size 4096. It stores a value at every 64 locations, corresponding to the cache line size of the atomic cache (256Byte). This arrangement ensures that the values used by each thread group are stored in a single cache line, and the cache line is exclusively accessed and modified by the threads in the same group, enhancing the performance of atomic operations. The value stored in $nextRowIdx$ guides the threads in a thread group to the next line they should operate on. For instance, if the group size is 64, there will be 4096/64=64 threads in each group. The value in $nextRowIdx$ is loaded into the atomic cache using the pz_atomic_store instruction (line 6 of Algorithm 2), and each thread increments the value when it fetches the value. The fetch-increment operation is achieved through $pz\_atomic\_inc(\&nextRowIdx[gid/blockSize])$, which returns the value stored in $nextRowIdx[gid/blockSize]$ and adds 1 to it in the atomic cache (Algorithm 2, line 6).

The atomic add operation on Pezy-SC3s' atomic cache differs from that on Nvidia GPUs. Pezy-SC3s' Atomic Cache delivers exceptionally high performance, capable of executing up to 32 operations in a single clock (refers to Table 1 for details). Each operation retrieves the $nextRowIdx$ for one thread, enabling the thread to identify the corresponding row for computation. Moreover, since the distribution of elements within each row of the sparse matrix is typically uneven, the time required to complete a row varies among most threads in the group. Consequently, it is uncommon for all threads to wait together for the atomic cache to compute $nextRowIdx$. This dynamic scheduling of thread tasks effectively achieves workload balance among threads with row-granularity partitioning, as illustrated in Fig. 5(d).

Moreover, it is also essential to minimize the $Interval$ between threads, which refers to the number of rows between any two threads, since increasing the $Interval$ leads to reduced data reuse between threads, as depicted in Fig. 6. In this figure, assuming there are 4 threads processing a matrix chunk and the workload of each row equals to x2 L2 Cache Line Size. When the $Interval$

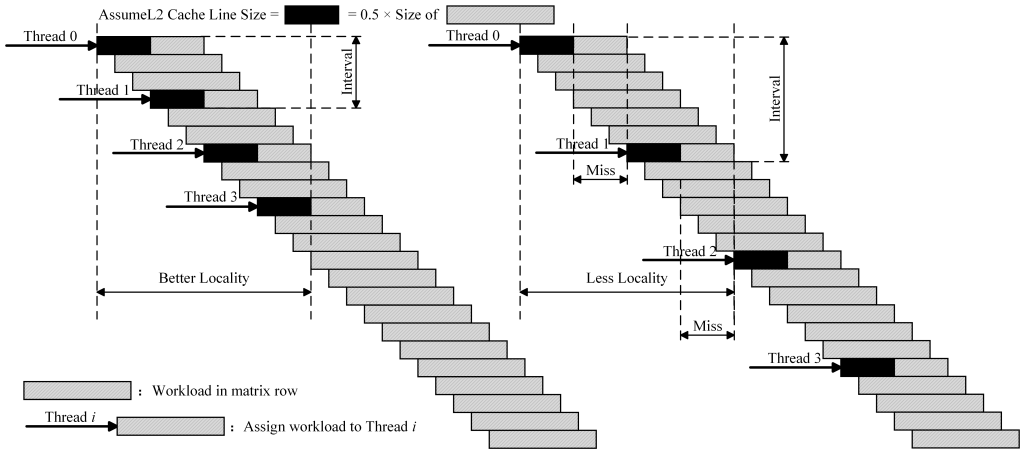

Fig. 6. Minimizing the Interval between threads to reduce cache miss.

between threads is less than or equal to 3, data can be well reused. However, if the *Interval* exceeds 3, it results in Cache Miss, which increases with the workload of each row. By applying Atomic Cache, we ensure that every thread will compute the adjacent row ($Interval = 1$), which also dramatically improves the data locality in the shared cache level [30].

3.3 Vectorization

Two basic vectorization strategies in SpMV are cross-row and in-row vectorization [12]. Pezy-SC3s owns 4096 threads, enables simultaneous processing of 4096 rows, resulting in inherent cross-row parallelism. To leverage this parallelism, we have partitioned the matrix, ensuring that each block contains complete rows. Within each block, 64 threads work in parallel to compute the result for each row, without any dependencies among the threads. Additionally, Pezy-SC3s provides 128-bit SIMD instructions, allowing us to increase the in-row computation parallelism by processing two numbers per thread at a time. This further enhances the SIMD parallelism of SpMV (see Algorithm 2, lines 8, 13-16).

4 Experimental Results and Evaluation

We conducted SpMV performance testing on 2092 matrices collected from the SuiteSparse [5] sparse matrix database, comparing our optimized SpMV implementations to the CSR-Based SpMV and CuSparse SpMV. Each SpMV execution consisted of 1000 warm-up executions and 3000 actual executions. On Pezy-SC3s, we evaluated the optimized SpMV based on double-precision floating-point performance, bandwidth utilization, and the speedup ratio achieved compared to the CSR-Based SpMV [11]. On Nvidia 3090ti GPU with CuSparse, due to the

different hardware configurations, we mainly evaluate the bandwidth utilization achieved by our optimized CSR-Based SpMV and CuSparse SpMV. We used the CSR-Based SpMV code provided by CuSparse for its double floating-point performance and bandwidth utilization test.

4.1 Preprocessing Overhead

In our method, the only preprocessing step involved is the chunking process. As mentioned in section 3.1, we select 64 as the chunking number, resulting in a fixed total of 63 chunking operations and a size of $65 \times sizeof(int)$ space overhead. Since the number of chunks remains constant regardless of matrix size, the preprocessing time for chunking is negligible compared to the time required for executing a single SpMV. We measured the preprocessing time and the average time of a CSR-Based SpMV for each of the 2092 matrices. The results are illustrated in Fig. 7, demonstrating that the preprocessing time overhead is minimal and consistently lower than the time required for two CSR-Based SpMV. Re-

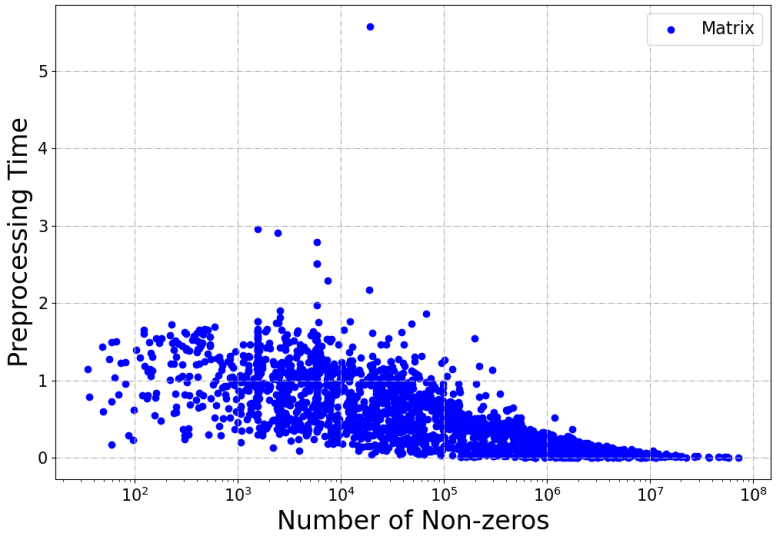

Fig. 7. Preprocessing Time (Normalize to average single CSR-Based SpMV time)

garding space overhead, the additional space required is a size of $65 \times sizeof(int)$ due to the fixed number of chunks. This means that the number of bytes added is fixed to $65 \times sizeof(int)$ for any matrix, which is negligible compared to the overall memory usage in real applications and the space occupied by sparse matrices.

4.2 Floating-point Performance

To obtain the floating-point performance, we considered one multiplication and one addition operation for each non-zero element. Assuming a total of nnz non-zero elements and a SpMV execution time of $time$, $GFlops$ is denoted as follows:

$$GFlops = \frac{2 \times nnz}{time} \tag{1}$$

We performed tests on the 2092 matrices by running CSR-Based SpMV, our optimized SpMV on Pezy-SC3s, and the CSR-Based SpMV provided by CuSparse on 3090ti. We measured the corresponding $GFlops$ achieved by executing SpMV for each matrix, and the results are shown in Fig. 8 and Fig. 9.

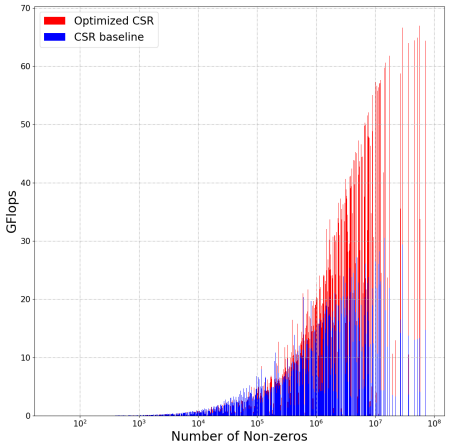

Fig. 8. Pezy-SC3s SpMV GFlops

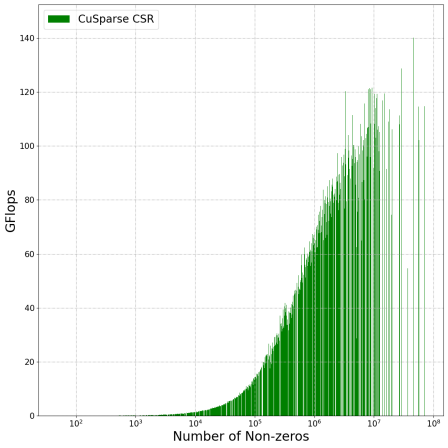

Fig. 9. 3090ti SpMV GFlops

The results indicate that the computing power of the hardware platform becomes more evident as the number of non-zero elements increases. When the number of non-zero elements is small ($nnz < 10^6$), all three SpMVs achieve relatively low GFlops, which gradually increase as the number of non-zero elements in the matrix grows. When the number of non-zero elements becomes sufficiently large ($nnz \geq 10^6$), our optimization of CSR-Based SpMV on Pezy-SC3s demonstrates a more significant improvement. CSR-Based SpMV also performs better when the data volume is very small ($nnz < 10^4$) due to shorter waiting times between threads and no need to wait for Atomic Cache workload balance. However, for practical applications, larger matrices are more commonly encountered and tend to have a more significant time overhead. The optimization effect becomes noticeable when the number of non-zero elements reaches around 10^6 and continues to improve with increasing non-zero elements.

4.3 Bandwidth Utilization

Bandwidth utilization is a crucial performance metric for SpMV implementations [11]. Due to the bandwidth-constrained and access-intensive nature of SpMV, achieving high bandwidth utilization is a critical optimization goal in computing platform with many cores [29, 21]. In our experiments, we focused on the number of bytes occupied by non-zero elements as the key factor in bandwidth calculations, which is a common approach in SpMV optimization efforts [14, 17, 31]. Assuming the floating-point type is *double*, a matrix with *nnz* non-zero elements, and an average time of *time* required for a single SpMV operation, the bandwidth calculation formula is:

$$Bandwidth = \frac{nnz \times sizeof(double)}{time} \tag{2}$$

Figure 10 and Figure 11 present the achieved bandwidth values for 2092 matrices on Pezy-SC3s and 3090ti using the three SpMV implementations, respectively.

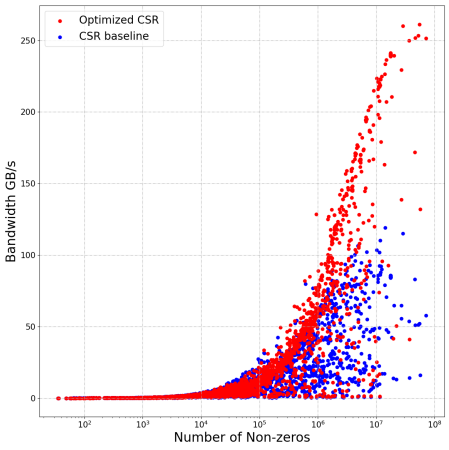

Fig. 10. Pezy-SC3s SpMV BandWidth

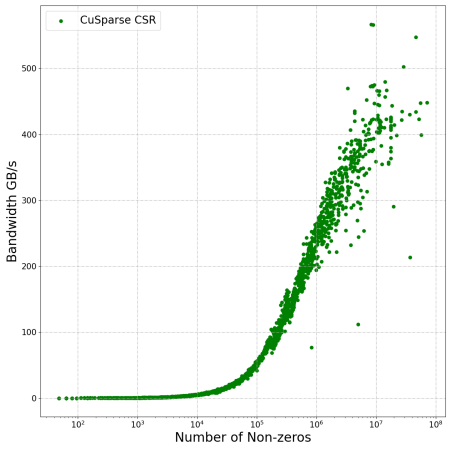

Fig. 11. 3090ti SpMV BandWidth

The results demonstrate that the bandwidth values for all three methods increase as the number of non-zero elements increases. This trend aligns with the measured *GFlops* in Fig. 8 and Fig. 9 and holds consistent in general. On Pezy-SC3s, our method exhibits more significant improvements in bandwidth utilization as the number of non-zero elements increases compared to CSR-Based SpMV. The effect of our optimization may not be apparent when the number of non-zero elements is less than 2^6, but as the number of non-zero elements in the matrix grows, the importance of workload balance between threads becomes more pronounced. The workload imbalance between different threads increases

as the number of non-zero elements in the matrix rises, making our optimization increasingly effective in addressing the workload imbalance issue. Notably, for matrices with the number of non-zero elements larger than 2^6, our optimization demonstrates considerable advantages and even achieves multiplicative performance gains compared to CSR-Based SpMV on certain matrices.

Furthermore, we conducted a comparison of SpMV bandwidth utilization on Nvidia GPUs. We evaluated the bandwidth of Pezy-SC3s and 3090ti using the open-source bandwidth test code STREAM benchmark [22]. The maximum value from the STREAM benchmark test results is denoted as R_{max} (307.62GB/s on Pezy-SC3s and 929.88GB/s on 3090ti), and the average bandwidth measured for each matrix during SpMV execution on both platforms is represented as R. The bandwidth utilization formula is as follows:

$$BandwidthUtilization = \frac{R}{R_{max}} \times 100\% \tag{3}$$

The specific test results are shown in Table [reference missing]. We tested SpMV on the 3090ti using Nvidia's highly optimized sparse library CuSparse. For each matrix, we performed 1000 warm-up SpMV operations followed by 3000 SpMV operations, averaging the total elapsed time to obtain the single SpMV elapsed time. The final comparison results are presented in Fig. 12. Our optimization

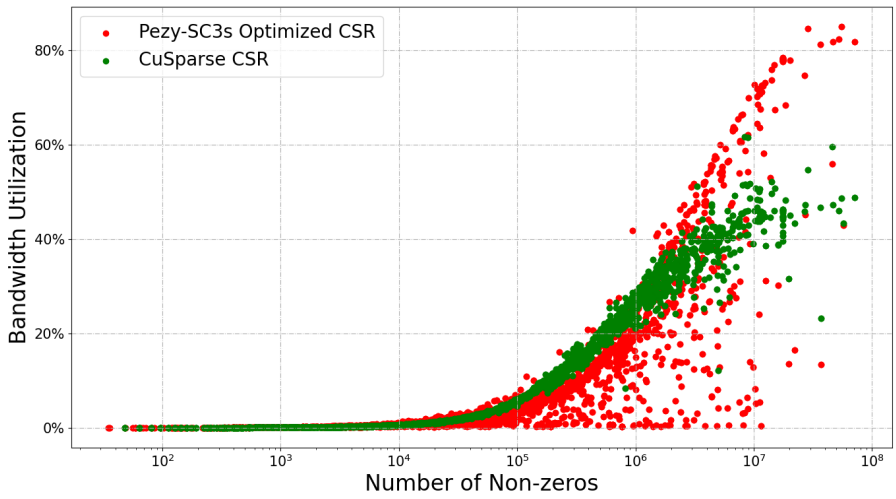

Fig. 12. Bandwidth Utilization On Pezy-SC3s and Nvidia 3090ti GPU

method achieves an average bandwidth utilization of 35.22% on large-scale matrices ($nnz \geq 10^6$), reaching 62.54% on very large-scale matrices ($nnz \geq 10^7$). This result surpasses the bandwidth utilization achieved by Nvidia CuSparse libraries on very large-scale matrices ($nnz \geq 10^7$), which is 36.17% and 45.56%,

respectively. The analysis reveals that our method achieves better bandwidth utilization compared to CuSparse when the matrix contains a sufficient number of non-zero elements. However, our bandwidth utilization slightly lags behind CuSparse for matrices with fewer non-zero elements ($nnz \leq 10^7$). As shown in Fig. 12, the performance of our method is similar to CuSparse on average, but the performance variance is higher (some points are close to the X-axis in Fig. 12). This can be attributed to our workload balance approach, which involves assigning at least one entire row at a time. As a result, we can not obtain a complete workload balance, making it unsuitable for matrices with specific sparse patterns. Consequently, this leads to relatively significant performance variance across different matrices.

4.4 SpeedUp

We also calculated the speedup ratio of the optimized SpMV over the CSR-Based SpMV for the 2092 matrices on Pezy-SC3s. Assuming the average time of the optimized SpMV is denoted as $time_{opt}$, and the average time of the CSR-Based SpMV is denoted as $time_{baseline}$, the speedup ratio ($SpeedUp$) can be expressed as:

$$SpeedUp = \frac{time_{baseline}}{time_{opt}} \tag{4}$$

As shown in Fig. 13, our optimized SpMV achieves a maximum speedup ratio

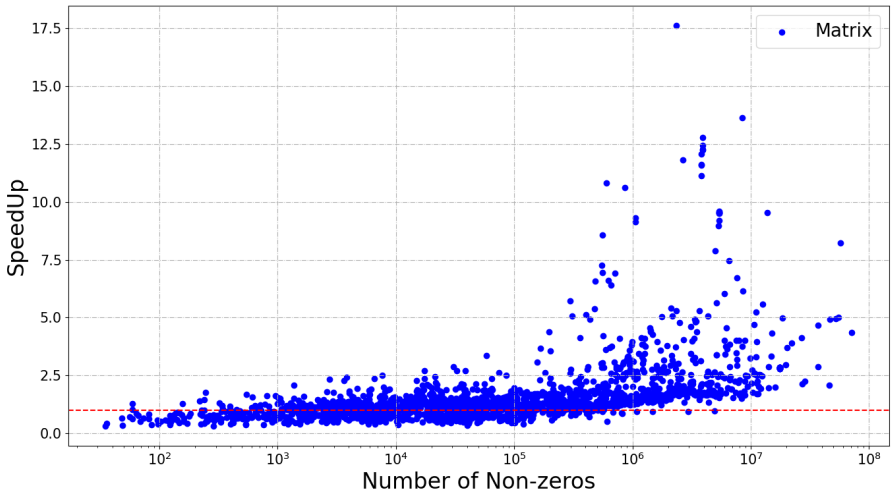

Fig. 13. SpeedUp Rate

of x17.63 compared to the CSR-Based SpMV. CSR-Based SpMV also performs

better when the data volume is very small ($nnz \leq 10^4$). However, for practical applications, larger matrices that impose significant time overhead are more common. Our optimized SpMV demonstrates increasingly noticeable benefits as the number of non-zero elements reaches approximately 10^6, and the performance improvement continues to grow with a higher number of non-zero elements.

5 Conclusion

In summary, we have optimized the CSR-Based SpMV on a new MIMD platform Pezy-SC3s. Our optimization approach involves matrix chunking at the cache level, with the best performance achieved when the number of chunks is set to 2^6. Fast matrix partitioning with less than two CSR-Based SpMV average time in most cases. Additionally, we utilize the Atomic Cache, a high-performance component provided by Pezy-SC3s, to schedule thread workload distribution in CSR-Based SpMV. This workload scheduling ensures that as many threads as possible are actively working simultaneously, avoiding thread waiting and computing resource waste, thereby addressing the workload imbalance problem. Finally, we apply 128-bit SIMD instructions to enhance parallelism. We test our method on 2092 matrices from the SuiteSparse collection, with the final results showing a maximum speedup ratio of x17.63 compared to the CSR-Based SpMV. Besides, we significantly improve bandwidth utilization. A comparison with Nvidia's highly optimized sparse library CuSparse on the 3090ti GPU indicates that our optimization method achieves an average bandwidth utilization of 35.22% on large-scale matrices ($nnz \geq 10^6$), reaching 62.54% on very large-scale matrices ($nnz \geq 10^7$). This result surpasses the bandwidth utilization achieved by Nvidia CuSparse libraries on very large-scale matrices ($nnz \geq 10^7$), which is 36.17% and 45.56%, respectively. These results highlight that our optimization method can effectively utilize the bandwidth and computing resources of Pezy-SC3s.

References

1. Ashari, A., Sedaghati, N., Eisenlohr, J., Parthasarathy, S., Sadayappan, P.: Fast sparse matrix-vector multiplication on gpus for graph applications. IEEE
2. Ashari, A., Sedaghati, N., Eisenlohr, J., Sadayappan, P.: An efficient two-dimensional blocking strategy for sparse matrix-vector multiplication on GPUs. An efficient two-dimensional blocking strategy for sparse matrix-vector multiplication on GPUs (2014)
3. Bell, N., Garland, M.: Implementing sparse matrix-vector multiplication on throughput-oriented processors. In: Conference on High Performance Computing Networking (2009)
4. Bolz, J., Farmer, I., Grinspun, E., Schrder, P.: Sparse matrix solvers on the gpu: conjugate gradients and multigrid. ACM (2003)
5. Davis, T.A., Hu, Y.: The university of florida sparse matrix collection. ACM Transactions on Mathematical Software (TOMS) **38**(1), 1–25 (2011)

6. Heller, M., Oberhuber, T.: Adaptive row-grouped csr format for storing of sparse matrices on gpu. Computer Science (2012)
7. Kepner, J., Bade, D., Buluc, A., Gilbert, J., Mattson, T., Meyerhenke, H.: Graphs, matrices, and the graphblas: Seven good reasons. arXiv e-prints (2015)
8. Kepner, J., Gilbert, J.: Graph algorithms in the language of linear algebra. Opencoursesfree Org **10.1137/1.9780898719918**, 315–337 (2011)
9. Khairoutdinov, M.F., Randall, D.A.: A cloud resolving model as a cloud parameterization in the ncar community climate system model: Preliminary results. Geophysical Research Letters **28**(18), 3617–3620 (2001)
10. Krotkiewski, M., Dabrowski, M.: Parallel symmetric sparse matrix-vector product on scalar multi-core cpus. PARALLEL COMPUTING -AMSTERDAM- (2010)
11. Langr, D., Tvrdik, P.: Evaluation criteria for sparse matrix storage formats. IEEE Transactions on parallel and distributed systems **27**(2), 428–440 (2015)
12. Li, C., Xia, T., Zhao, W., Zheng, N., Ren, P.: Spv8: Pursuing optimal vectorization and regular computation pattern in spmv. In: 2021 58th ACM/IEEE Design Automation Conference (DAC). pp. 661–666. IEEE (2021)
13. Liang, Y., Tang, W.T., Zhao, R., Lu, M., Huynh, H.P., Goh, R.S.M.: Scale-free sparse matrix-vector multiplication on many-core architectures. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **36**(12), 2106–2119 (2017). https://doi.org/10.1109/TCAD.2017.2681072, https://doi.org/10.1109/TCAD.2017.2681072
14. Liu, W., Vinter, B.: Csr5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In: Proceedings of the 29th ACM on International Conference on Supercomputing. pp. 339–350 (2015)
15. Maggioni, M., Bergerwolf, T.Y.: Adell: An adaptive warp-balancing ell format for efficient sparse matrix-vector multiplication on gpus. In: IEEE Computer Society. pp. 11–20 (2013)
16. Merrill, D., Garland, M.: Merge-based parallel sparse matrix-vector multiplication. In: SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (2017)
17. Merrill, D., Garland, M.: Merge-based sparse matrix-vector multiplication (spmv) using the csr storage format. Acm Sigplan Notices **51**(8), 1–2 (2016)
18. Mohri, M.: Semiring frameworks and algorithms for shortest-distance problems. Journal of Automata, Languages and Combinatorics (2002)
19. Mu, S., Xiao, L., Li, R., Zhang, Z., Tao, Y., Zhu, M., Deng, Y.: Gpu accelerated sparse matrix-vector multiplication and sparse matrix-transpose vector multiplication. Concurrency and computation: practice and experience (2015)
20. Ravishankar, M., Dathathri, R., Elango, V., Pouchet, L.N., Ramanujam, J., Rountev, A., Sadayappan, P.: Distributed memory code generation for mixed irregular/regular computations. In: Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 65–75 (2015)
21. Shalf, J., Dosanjh, S.S., Morrison, J.: Exascale computing technology challenges. In: High Performance Computing for Computational Science - VECPAR 2010 - 9th International conference, Berkeley, CA, USA, June 22-25, 2010, Revised Selected Papers (2010)
22. STREAM: Sustainable memory bandwidth in high performance computers. http://www.cs.virginia.edu/stream/
23. Sundaram, N., Satish, N.R., Patwary, M., Dulloor, S.R., Vadlamudi, S.G., Das, D., Dubey, P.: Graphmat: High performance graph analytics made productive. arXiv e-prints (2015)

24. Venkat, A., Hall, M., Strout, M.: Loop and data transformations for sparse matrix code. ACM SIGPLAN Notices **50**(6), 521–532 (2015)
25. Vázquez, F., Fernández, J., Garzón, E.: A new approach for sparse matrix vector product on nvidia gpus. Concurrency & Computation Practice & Experience **23**(8), 815–826 (2011)
26. Vázquez, F., Ortega, G., Fernández, J., Garzón, E.: Improving the performance of the sparse matrix vector product with gpus. In: IEEE International Conference on Computer & Information Technology (2010)
27. Wang, E., Zhang, Q., Shen, B., Zhang, G., Lu, X., Wu, Q., Wang, Y., Wang, E., Zhang, Q., Shen, B., et al.: Intel math kernel library. High-Performance Computing on the Intel® Xeon Phi™: How to Fully Exploit MIC Architectures pp. 167–188 (2014)
28. Wang, Y., Pan, Y., Davidson, A., Wu, Y., Yang, C., Wang, L., Osama, M., Yuan, C., Liu, W., Riffel, A.T.: Gunrock: Gpu graph analytics (2017)
29. Wright, M.: The opportunities and challenges of exascale computing (2010)
30. Xie, B., Zhan, J., Liu, X., Gao, W., Jia, Z., He, X., Zhang, L.: Cvr: Efficient vectorization of spmv on x86 processors. In: Proceedings of the 2018 International Symposium on Code Generation and Optimization. pp. 149–162 (2018)
31. Yan, S., Yan, S., Li, C., Li, C., Zhang, Y., Zhang, Y., Zhou, H., Zhou, H.: Yaspmv: Yet another spmv framework on gpus. ACM SIGPLAN Notices (2014)
32. Zheng, C., Gu, S., Gu, T.X., Yang, B., Liu, X.P.: Biell: A bisection ellpack-based storage format for optimizing spmv on gpus. Journal of Parallel and Distributed Computing **74**(7), 2639–2647 (2014)