PrSpMV: An Efficient Predictable Kernel for SpMV

Gelin Fu, Tian Xia, Shaoru Qu, Zhongpei Luo, Shuyu Li, Pengyu Cheng, Runfan Guo, Yitong Ding, Pengju Ren
Xi'an Jiaotong University, Xi'an, China
{fugelin, qushaoru202, luozhongpei, lsyxary, 2141389737, jimmygrf, dingyitong}@stu.xjtu.edu.cn,
{tian_xia, pengjuren}@xjtu.edu.cn

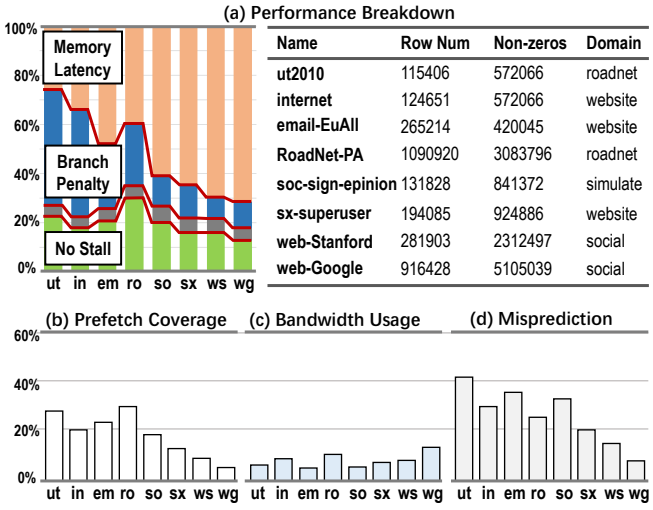
***Abstract*—Sparse Matrix-Vector Multiplication (SpMV) has been widely applied in scientific computation, industry simulation, and intelligent computation domains, which is the critical algorithm in all these applications. Due to the poor data locality, low cache usage, and extremely irregular branch patterns caused by the highly sparse and random distributions, SpMV optimization has become one of the most challenging problems for modern high-performance processors. In this paper, we study the bottlenecks of SpMV on current out-of-order CPUs and propose a novel SpMV kernel named PrSpMV to improve its performance by pursuing high predictability. Specifically, we improve the memory access regularity and locality by creating serialized access patterns so that the data prefetching efficiency and cache usage are optimized. We also improve pipeline efficiency by creating regular branch patterns to make branch prediction more accurate. Experiment results show that using the above optimization approaches, PrSpMV can eliminate nearly all branch mispredictions. Moreover, it can also significantly reduce the average L2 cache miss rate from 57% to 20% via efficiently leveraging hardware prefetchers. By using PrSpMV, stride prefetcher can be boosted with 1.31× speedup and dedicated irregular prefetcher can be improved with 1.40× speedup. Meanwhile, on commercial high-end Intel processors, it achieves 1.32× speedup against some state-of-the-art SpMV kernels.**

***Index Terms*—Sparse Matrix Computation, Data Prefetching, Branch Prediction**

I. Introduction

Sparse Matrix-Vector Multiplication (SpMV) is a widely-used kernel in scientific and industrial applications such as graph operations, high-performance computing, and machine learning [1] [2] [3]. It performs y=$A \times x$ where $x \in \mathcal{R}^{n\times 1}$ and $y \in \mathcal{R}^{m\times 1}$ is a dense vector, and $A \in \mathcal{R}^{m\times n}$ is a sparse matrix where non-zeros are in extremely low density (e.g., $< 0.01\%$). Typically, the SpMV kernel is iteratively performed to fulfill precise numeric simulation, linear algebra solving, or graph analytics, making it the most critical algorithm for many applications. However, current high-end processors usually suffer from poor performance when computing SpMV kernels. This is mainly due to the high sparsity and random distribution of non-zero elements, which results in irregular memory accesses and poor data locality. Furthermore, working sets are usually too large to fit in a conventional cache, leading to frequent cache misses that require long latency to solve. As a result, SpMV has been considered one of the most critical and challenging bottlenecks for modern processors.

This work was supported in part by National Key Research and Development Program of China No.2022YFB4500500, National Natural Science Foundation of China No.62088102, Key Research and Development Program of Shaanxi No.2022ZDLGY01-08, and Fundamental Research Funds for the Central Universities under Grant xtr072022001. The Authors are with the National Key Laboratory of Human-Machine Hybrid Augmented Intelligence, National Engineering Research Center of Visual Information and Applications, and Institute of Artificial Intelligence and Robotics, Xi'an Jiaotong University, Xi'an, Shaanxi, China. The corresponding author is Pengju Ren.



Name	Row Num	Non-zeros	Domain
ut2010	115406	572066	roadnet
internet	124651	572066	website
email-EuAll	265214	420045	website
RoadNet-PA	1090920	3083796	roadnet
soc-sign-epinion	131828	841372	simulate
sx-superuser	194085	924886	website
web-Stanford	281903	2312497	social
web-Google	916428	5105039	social

Fig. 1. Performance analysis of SpMV on intel Xeon CPU.

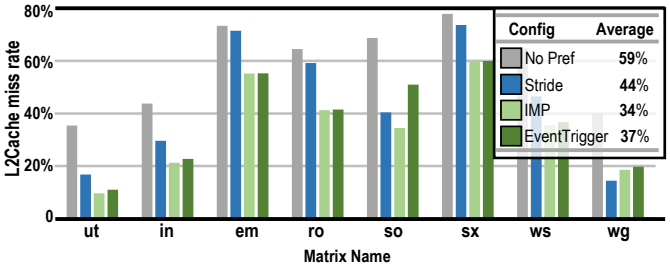


Fig. 2. L2 cache miss rate of prefetchers running SpMV kernel.

Meanwhile, current processors heavily rely on predictability to achieve high performance. Specifically, accurate branch prediction is critical to maintaining an efficient processor pipeline, which requires the branches to have predictable patterns. Moreover, the data prefetching technology is usually used to improve throughput for memory-bound workloads, where the hardware prefetcher predicts the addresses of subsequent memory accesses and fetches them in advance to hide long memory latency. Current commercial processors have trivial data prefetchers such as next-line or stride prefetchers to prefetch stream-in or fixed stride data access patterns. However, SpMV kernel usually has poor predictability for both branches and memory accesses. Fig 1 depicts the runtime breakdown of SpMV kernel's execution on Intel Xeon

CPU on several large-scale sparse matrices. In Fig 1(a), the penalty of branch misprediction and memory access latency contribute the most runtime overhead. Further measurements of Fig 1(b)(c)(d) show that the existing data prefetchers and branch prediction units of Intel processor are working in very low efficiency, indicating the severe bottleneck due to the poor predictability of SpMV. It is also obvious that memory access latency is the most critical bottleneck for SpMV in most cases.

Focusing on this problem, dedicated prefetchers are proposed to improve memory access performance. Yu [4] proposed a hardware prefetcher named IMP, which detects and captures the SpMV irregular memory access pattern during the execution so that the sparse non-zeros and the corresponding elements in the vector can be fetched in advance. Another typical approach is Event-trigger [5], where the irregular access patterns are learned by a customized compiler and then used for hardware prefetching. Though such approaches are effective to alleviate the bottlenecks of SpMV, their gains are still limited. This is because data are still fetched and used with very poor locality, meaning that only very few data of a prefetched cache line are used before the cache line gets evicted or replaced, which results in low data utilization and abundant memory accesses. We evaluate the efficiency of these prefetchers using the metric of cache miss rate and the results in Fig 2 indicate that memory bottleneck still exists even with these prefetchers. Meanwhile, such approaches pay no attention to the branch misprediction, which imposes significant overhead as shown in Fig 1. Furthermore, such dedicated hardware prefetchers pay high complexity to support complicated irregular memory access pattern identification, which undermines the overall optimization benefits.

In this paper, we propose a novel kernel, termed as **Predictable SpMV(PrSpMV)**. It is designed to improve the efficiency of current commercial processors by providing a more predictable SpMV kernel including more regular branch patterns and serialized data accesses so that current branch predictor and stride prefetcher can work efficiently. Moreover, we improve the data locality to optimize the cache usage. As a result, we can leverage a general commercial processor with a commonly-used stride prefetcher to achieve high performance. Our approach can also cooperate with existing irregular hardware prefetchers (e.g., IMP and event-trigger) to achieve even higher performance. Evaluation with extensive experiments shows that our PrSpMV can significantly reduce the average L2 cache miss rate from 57% to 20% via efficiently leveraging hardware prefetchers. By using PrSpMV, stride prefetcher can be boosted with 1.31× speedup and dedicated irregular prefetcher can be improved with 1.40× speedup. Meanwhile, on commercial high-end Intel processors, it achieves 1.32× speedup against some state-of-the-art SpMV kernels. Moreover, PrSpMV has low pre-processing overhead which indicates that it is highly applicable to real-world scenarios.

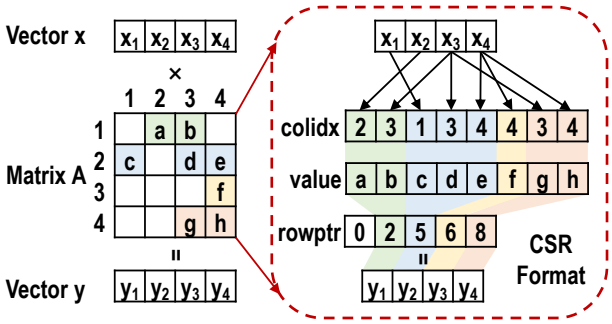


Fig. 3. SpMV algorithm using CSR format.

II. Background

Our work focuses on the execution of SpMV with the Compressed Sparse Rows (CSR) format, which is widely used in many applications. As shown in Fig 3, the CSR format encodes a sparse matrix with three arrays: *value* as the nonzero values, *colidx* as the column indices of nonzero values, and *rowptr* to store the pointers of each row's first element in the *value* and *colidx* array. Algorithm 1 shows the pseudo-code of SpMV based on CSR format, while Fig 4 depicts an example implementation of SpMV kernel on an AArch64 machine. By studying the SpMV codes, we analyze the critical bottlenecks for current processors and existing optimization techniques.

Algorithm 1 Scalar Sparse Matrix-Vector Multiplication

Require: CSR $A[m][n]$, double-float $x[n]$
Ensure: double-float $y[m]$
1: **for** $i = 0$ to $m - 1$ **do**
2: $y[i] \leftarrow 0$
3: **for** $j = A.rowptr[i]$ to $A.rowptr[i + 1] - 1$ **do**
4: $y[i] \leftarrow y[i] + A.nnz[j] \times x[A.colidx[j]]$
5: **end for**
6: **end for**

A. Memory Access

As depicted in Fig 4, the SpMV kernel traverses non-zero values for each row in the sparse matrix using *colidx* to fetch their corresponding elements in *x*, and then uses multiply-and-add to calculate *y*. During this process, accesses on the CSR format (*colidx*,*value*,*rowptr*) and the result vector y are stream-in patterns that can be easily detected and handled by the stride prefetchers that are available in most modern high-end processors. On the other hand, the access of *x[colidx[i]]* is very unfriendly to modern CPU because each non-zero value in the matrix loads from arbitrary positions of *x* which usually has poor locality. Earlier research [6] has shown that more than 95% of cache misses in SpMV are caused by the execution of *x[colidx[i]]*. This irregular access pattern is termed as *indirect access* with basic form of *A[B[i]]*.

To optimize the accesses on x, some work [7] [8] proposed column tiling so that for each tile the addresses of x access are restricted. However, such methods became inefficient on large matrices as the cost of generating final results from multiple

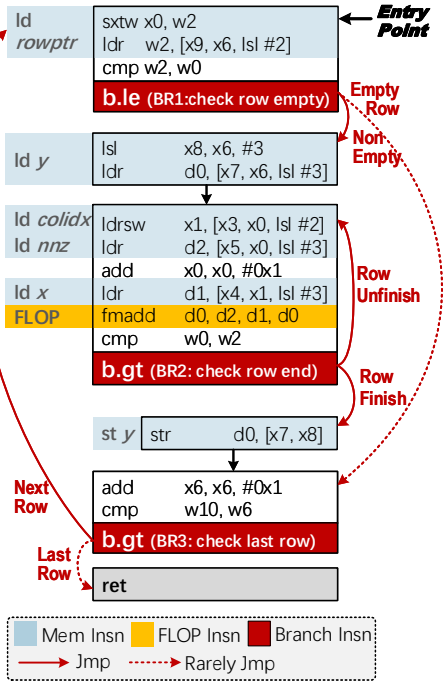


Fig. 4. SpMV instruction flow in ARMv8.

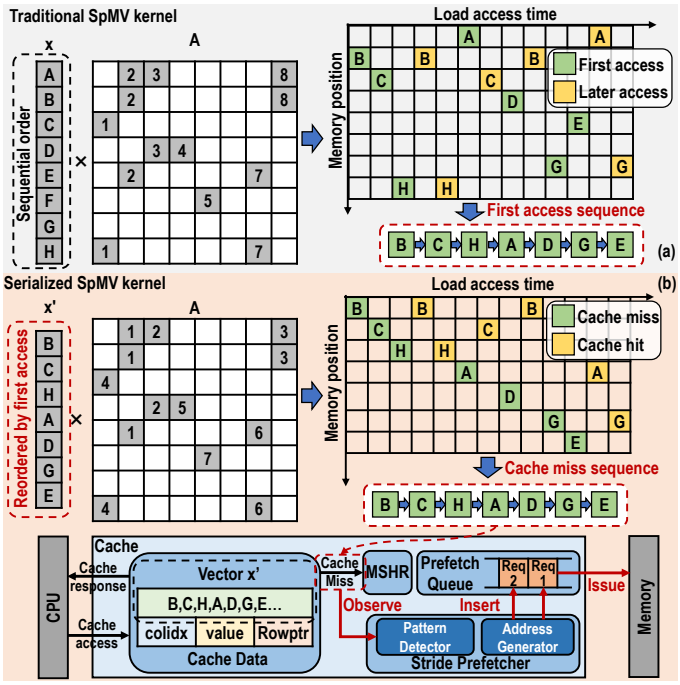


Fig. 5. Memory access pattern comparison between traditional SpMV kernel and our method.

tiles gets high when there are more tiles. Recently, the data prefetching technique has been considered a promising solution. Focusing on the indirect access pattern of *x[colidx[i]]*, customized hardware prefetchers were designed. One state-of-the-art research is IMP [4] which uses hardware units to dynamically detect and prefetch the indirect access pattern during execution. However, due to the disturbance of out-of-order execution and branch mispredictions, IMP usually fails to prefetch all the required x values. An alternative approach is to detect the access pattern during compilation instead of execution. Event-Trigger [5] is a state-of-the-art method of this category. It proposed a customized compiler to analyze the kernel and label the found pattern using customized instructions, which are then used to guide the data prefetching at runtime. This approach can achieve good performance as the access patterns are precisely identified and accurately prefetched. However, such a method requires software to be rebuilt using a modified compiler and executed on the modified processor, which requires too much effort to be practically feasible. Meanwhile, though x values are prefetched, their actual usage is still in poor locality as a cache line is likely to be used very few times, even only once, before it gets evicted from cache. This also undermines the optimization effects.

B. Branch prediction

Modern processor relies on the branch prediction technique to maintain pipeline efficiency. When encountering a branch, the branch predictor speculatively picks one direction based on the history record and continues execution. Wrong branch prediction usually causes great penalty because wrong-fetched instructions need to be flushed from the pipeline and the processor state should be restored to the branch point. While such penalty has been overlooked in existing SpMV studies, it is a critical performance factor as in Fig 1. Specifically, as shown in Fig 4, there are three branches in the SpMV code path noted as *BR1/BR2/BR3*, whose branch conditions depend on different factors. Among them, *BR1* and *BR2* are usually easy to predict as their conditions of empty row and last row are unlikely to happen during the execution. On the other hand, the condition of *BR2* depends on the random length of the row which is unpredictable by its nature, resulting in poor prediction accuracy. Therefore, *BR2* is the most critical branch and incurs most branch mispredictions.

III. Kernel Design

Based on the bottleneck analysis in Section II, in this section we provide an efficient novel SpMV kernel, Predictable SpMV (PrSpMV), to improve SpMV performance by providing more predictable memory access and branch patterns.

A. Vector Access Serialization

Fig 5(a) shows an example of $x[colidx[i]]$ in SpMV. Elements in the input vector x are sequentially placed in memory and matrix loads elements of x from arbitrary positions which have poor cache locality. Assuming stride prefetcher in cache is observing the address of cache miss, it fails to detect a stream-in pattern for vector x because the addresses of cache miss are irregularly changed.

To resolve this problem, we collect the address sequence of the first access for vector x (i.e., the matrix first accesses this element). The vector x is rearranged to the vector x' by the order of address sequence of the first access as shown in Fig 5(b). Assuming that subsequent access to the same element will hit the cache due to temporal locality, now the addresses of cache miss sequence for x' are sequential, so that

the stride prefetcher can build a stream-in pattern for x' and insert prefetch requests into the prefetch queue. Then prefetch queue issues prefetch requests in a FIFO style to memory and fetch data in advance to hide memory latency.

B. Block Partitioning Strategy

We hope that elements of x' won't be fetched from memory more than once. Otherwise, the cache miss sequence observed by prefetcher will be mixed with additional misses caused by evicted elements and it will disturb the process of stream-in pattern identification. However, for many matrices, the vector size is usually larger than cache and inevitably causes earlier fetched cache line to be evicted and re-fetched later. To address this problem, we partition the matrix into multiple row blocks so that each block only access a smaller part of x' which can all fit in the cache, as shown in Fig 6(a). In this way, the vector x are rearranged to x'=$\{x_1$',x_2',...$\}$, where each x_k' represents the vector element array used for each block respectively. We use the size of x_k' to determine the block partitioning.

As different blocks may contain the same elements, the size of overall rearranged vector x' could be larger than the original vector x. Therefore, while using smaller blocks can create smaller x_k' that is easier to fit in the cache, it also tends to generate a larger size of x' as there are likely more repeated elements among blocks. So the size of x_k' should be wisely chosen. We assume that for each block, its vector array x_k' size should not be larger than the cache size. Considering other CSR data structures (*value*,*rowptr*,*colidx* arrays) are also stored in cache, we set the size of x_k' to be half of the cache size according to empirical experiments.

C. Bitmap-Based Row Reorder Strategy

To further reduce the overall size of x', we propose a bitmap-based row reordering strategy to reduce repeated elements among different blocks by adjusting the distribution of non-zeros in different blocks. As shown in Fig 6(b), we divide the matrix into different column regions. For each row, count non-zeros of each region, and mark the region with the most non-zeros as one while others as zero, resulting in a one-hot bitmap label for each row. By rearranging all rows according to a descending label ordering, non-zeros of different blocks are more likely to be placed in different columns and thus resulting in less amount of repeated elements in x'.

D. Branch Optimization

As shown in Fig 4, the branch *BR-2* is the most critical branch during SpMV execution. It causes most of the branch mispredictions because its direction depends on the random length of each row in the matrix which is unpredictable. We propose the method of rearranging the matrix row to alleviate this problem. Specifically, each block is divided into bundles of 2048 consecutive rows. In each bundle, we sort the matrix rows by length. This method ensures that the branch jump pattern of consecutive rows is the same to facilitate branch predictor pattern identification.

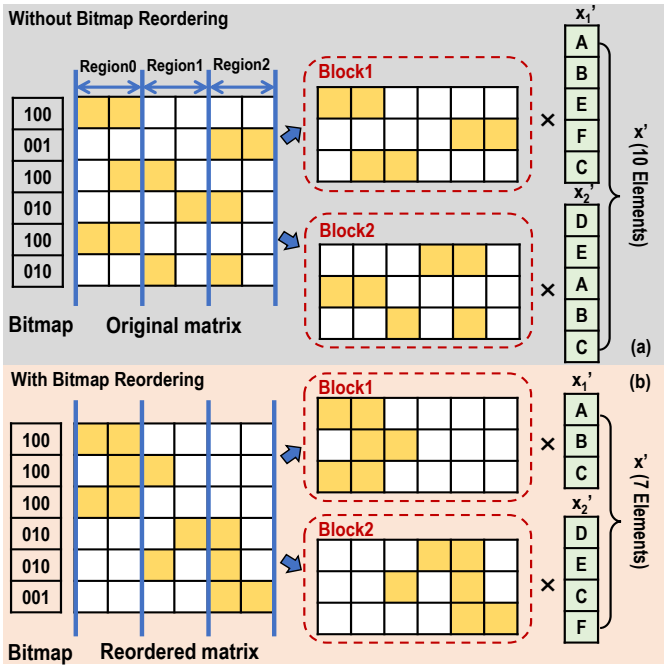


Fig. 6. Example of PrSpMV's block partitioning and bitmap row reorder strategy.

E. Preprocessing and Computation Flow

1) Preprocessing Flow: The preprocessing of **PrSpMV** combines the above optimizations, which includes the following steps as shown in Fig 7(a):

- **Step-1**: Rearrange sparse matrix rows based on bitmap and record mapping function between the original matrix and rearranged matrix as R_1.
- **Step-2**: Partition bitmap-reordered matrix into blocks by the predetermined block size.
- **Step-3**: Partition each block into bundles and rearrange rows in each bundle to improve branch prediction. Record the row mapping function as R_2.
- **Step-4**: Perform serialization for each bundle by modifying the *colidx* value and record this mapping function as S.
- **Step-5**: To support iterative computation, output after each iteration needs to be reorganized into input vector required for the next iteration. PrSpMV records this mapping function for each row i as $W_1[i] = S[R_2[R_1[i]]]$. It is also necessary to restore the output of the last iteration to the original row order. The function for restoration is $W_2[i] = Rev(S[R_2[R_1[i]]])$ where *Rev* represents a function that reverses the key and value of function.

After preprocessing, the sparse matrix is represented as a series of bundles and three mapping functions are recorded including S, W_1 and W_2.

2) Computation Flow: Fig 7(b) depicts the process of iterative SpMV computation using the format generated by the above preprocessing, which includes the following steps:

- **Step-1**: For the first iteration, perform SpMV with vector x_0 and the original sparse matrix and generate otuput

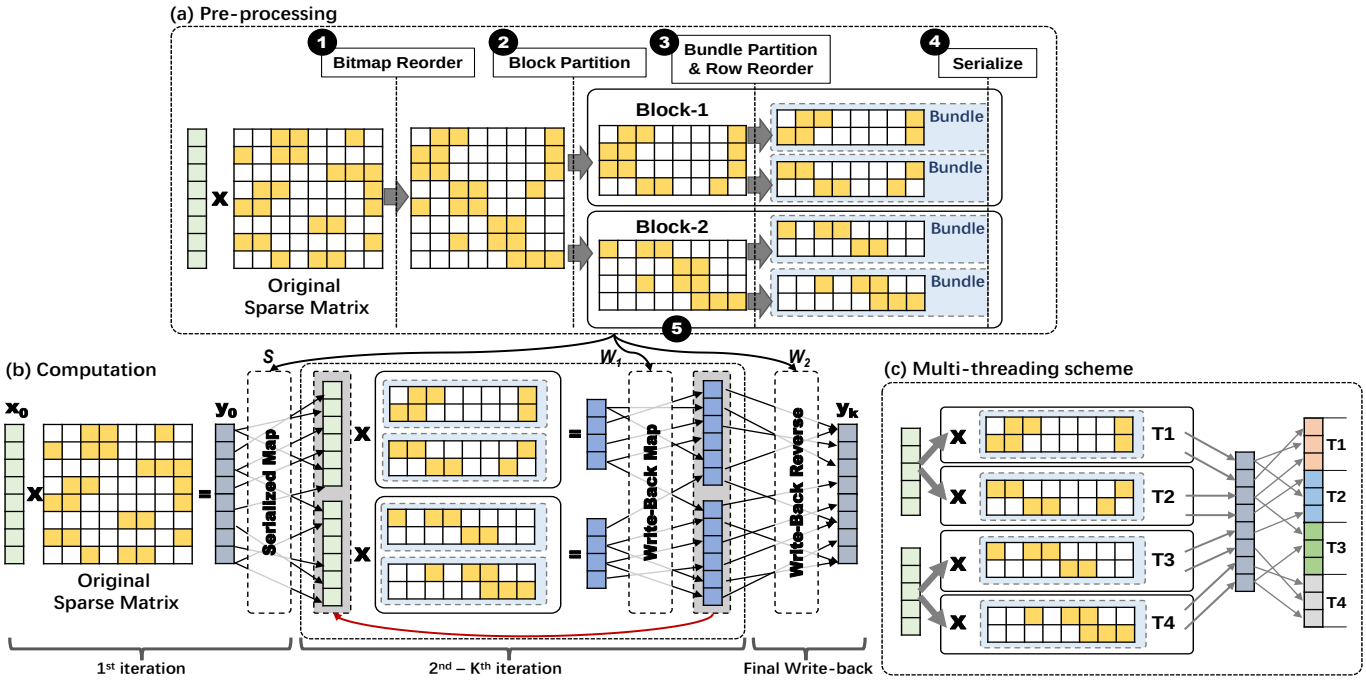


Fig. 7. Preprocessing flow, computation method, and multi-thread implementation of PrSpMV.

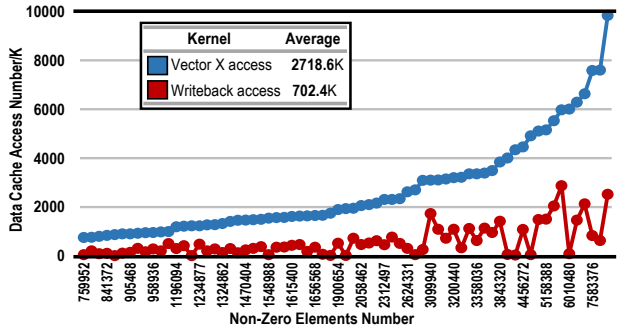


Fig. 8. Comparisons between vector access and write-back access number.

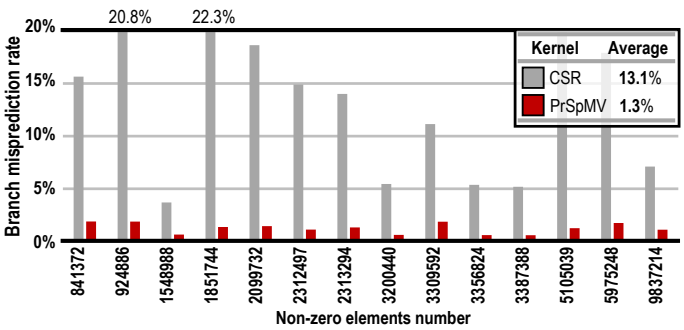


Fig. 9. Branch misprediction rate for CSR and PrSpMV. Lower is Better.

vector y_0. Then transfer vector y_0 into the order of serialized access pattern by applying function S.

- **Step-2**: Perform SpMV with transferred vector and sub-matrix (i.e., each partitioned block of the matrix) individually and write results into dense output vector. Each sub-matrix performs serializing memory access for vector so that stride prefetcher can be triggered and cache locality is improved.
- **Step-3**: After all sub-matrices are computed, elements in result vector are rearranged to the order of the serialized pattern by W_1, and perform SpMV for the next iteration with the converted result vector and matrix.
- **Step-4**: Repeat Step-2 and Step-3 until the result converges.
- **Step-5**: Restore the original order for output vector by function W_2.

It should be noted that PrSpMV can be easily adapted to multi-core systems by assigning each thread with several bundles as Fig 7(c) describes.

F. Write-back Overhead

It should be noted that by serializing accesses on vector and row reordering, the write-back of final SpMV results imposes extra overheads as the row-wise calculation results in y are generated in an out-of-order fashion. Therefore, at the end of SpMV computation, we need to update x' with y with an out-of-order value assignment. Though this process adds extra random memory accesses, we consider it a rational trade-off as the performance benefits gained from optimizing the accesses of SpMV is usually much more profitable. To quantify the trade-off benefits, Fig 8 depicts the number of vector x' access during the SpMV computation and the write-back access after SpMV computation on total of 76 matrices. It can be observed that as the access number of vector x' increases fast with the matrix size, the write-back access numbers are growing slowly, indicating it would be profitable to trade some out-of-order write-back accesses for much more efficient and prefetchable accesses on vector x'.

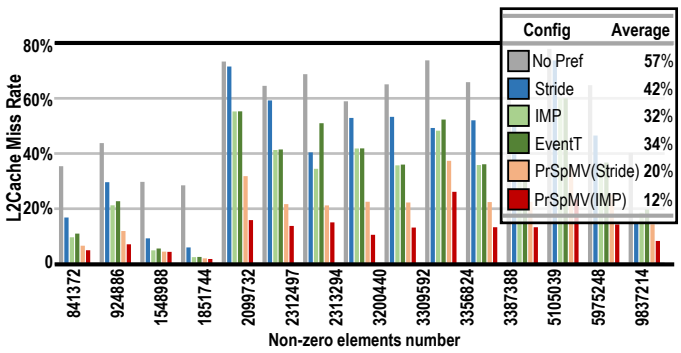


Fig. 10. L2 cache miss rate for different approaches. Lower is Better.

TABLE I
INFORMATION OF EVALUATION PLATFORMS.

Architecture	**AArch64**	**x86**
Platform	GEM5	Intel Xeon Gold 6146
Processor	ARMv8 (O3CPU)	Skylake-X
Frequency	2.5GHz	2.5GHz
L1 D-Cache	32KB	32KB
L2 Cache	1MB	1MB
L3 Cache	/	24.75MB

IV. EVALUATIONS

A. Experiment Methodology

1) Environment setup: We evaluate our method on a simulated AArch64 processor using Gem5 simulator in order to study the influences of factors such as branch prediction, prefetching, and cache. The Gem5 CPU model is O3CPU that can simulate superscalar out-of-order CPUs with high accuracy ($\geq$95%) [9]. Besides, we measure the effects of our method on commercial x86 platforms of Intel Xeon Gold (Skylake) 6146 Processor. For both platforms, Ubuntu 18.04 is used. Table I lists the spec of platform configurations. It should be noted Gem5 simulator is configured to be close to the Intel CPU hardware factors, i.e. the CPU frequency, cache hierarchy, and sizes. Additionally, we use region-of-interest functionality from GEM5 to only profile the core algorithm in order to improve simulation accuracy.

2) Comparison: On GEM5 AArch64 platform, we evaluate PrSpMV in comparison with existing state-of-the-art irregular prefetching techniques that are dedicated to solving SpMV access patterns, including IMP [4] and Event-Trigger [5]. We use the artifacts provided by the authors for evaluating IMP and Event-Trigger. Specifically, we evaluate the effect of PrSpMV when working with commonly-used trivial stride prefetcher residing in the L2 cache as well as with the IMP technique. The baseline of GEM5 AArch64 is a trivial CSR-based scalar SpMV kernel as shown in Fig 4 running without any hardware prefetching unit. For the x86 platform, we set the original CSR SpMV routine from Intel Math Kernel Library [10](MKL version 2020.2) as the baseline. We also compare with several state-of-the-art approaches including CSR5 [11] and CVR [12]. All SpMV algorithms are tested on their available open-source code and are compiled by GCC 11.3 with the -O2 option.

3) Dataset: For the experiment workloads of the SpMV kernel, we use the widely-used SuiteSparse Collection [13] and select groups of various applications and data sources including the Stanford Network Analysis Platform (SNAP) [14], the Discrete Mathematics and Theoretical Computer Science (DMACS) and the Pajek. The tested dataset includes a total number of 78 sparse matrices of wide-ranged shapes with 106K–9M non-zero value and 31K–0.8M rows.

B. Optimization Effect Analysis

This section shows that by providing a more predictable SpMV kernel, PrSpMV is quite effective to optimize memory bottleneck and branch predictions. Our experiments are conducted on 14 representative matrices of different types and sizes. Specific monitors are added to the GEM5 simulator in order to profile the key metrics of cache and branch predictor.

We first evaluate the misprediction rate of the critical branch *BR2* listed in Fig 4. The stage-of-the-art branch predictor TAGE [15] is used. As shown in Fig 9, CPU equipped with this branch predictor has an average 13% failure rate on this branch, because sparse matrices have irregular row lengths and most rows are very short, making it difficult to predict. Meanwhile, by using PrSpMV, most mispredictions are eliminated, resulting in only 1.3% misprediction rate on this branch, which indicates the effectiveness of branch optimization in PrSpMV.

We then evaluate the memory bottleneck by measuring the L2 cache miss rate. As shown in Fig 10, the non-prefetching baseline has a pretty high ($\geq$55% on average) L2 cache miss rate, indicating memory latency is a critical problem for SpMV. Using stride prefetcher reduces the miss rate to an average of 34%, but still remains many unsolved cache misses. However, when using PrSpMV kernel, the stride prefetcher can greatly reduce most cache misses and result in only a 20% cache miss rate, which outperforms specific hardware prefetchers of IMP and Event-Trigger. Moreover, we test using PrSpMV with the IMP hardware prefetcher and show that even more cache misses are eliminated as the write-back sequence of PrSpMV can be efficiently handled by the IMP prefetcher, resulting in an average 12% L2 cache miss rate. Therefore, for both trivial stride prefetcher and dedicated irregular prefetchers, PrSpMV can greatly optimize their memory accesses.

C. SpMV Performance

This section evaluates the overall performance of SpMV kernels in metrics of GFLOPs and speedup against the baseline. All benchmark matrices are tested. The speedup of each approach is calculated as $\frac{t_{base}}{t_{opt}}$, where t_{base} is the runtime of the baseline method and t_{opt} is the runtime of the corresponding compared method.

Fig 11 compares PrSpMV with several prefetching techniques including stride prefetcher and two dedicated irregular prefetchers of IMP and Event-Trigger on the GEM5 AArch64 processor. As PrSpMV creates a more predictable memory access pattern with improved data locality, it can be observed that using PrSpMV can effectively improve the performance of existing prefetchers. Specifically, PrSpMV improves the

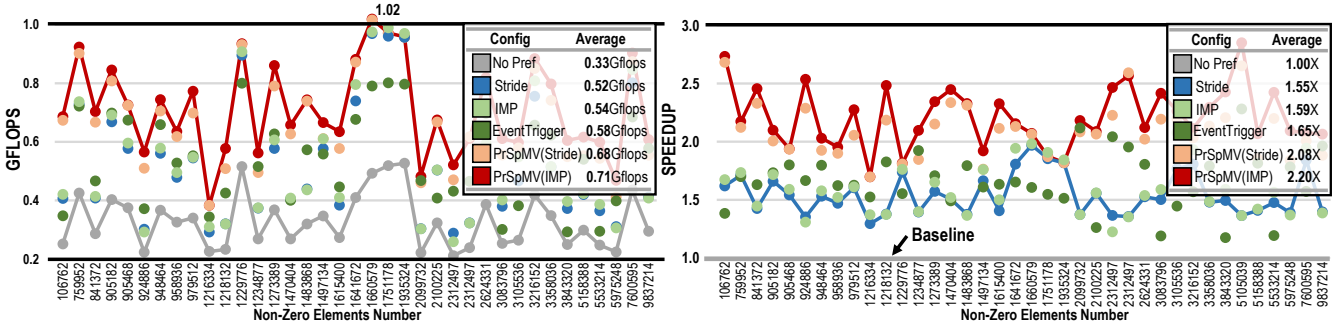


Fig. 11. Performance comparison of non-prefetching baseline, stride prefetcher, IMP, EventTrigger, PrSpMV(Stride)(our method), and PrSpMV(IMP)(our method) on Gem5 AArch64 CPU platform. Higher is better.

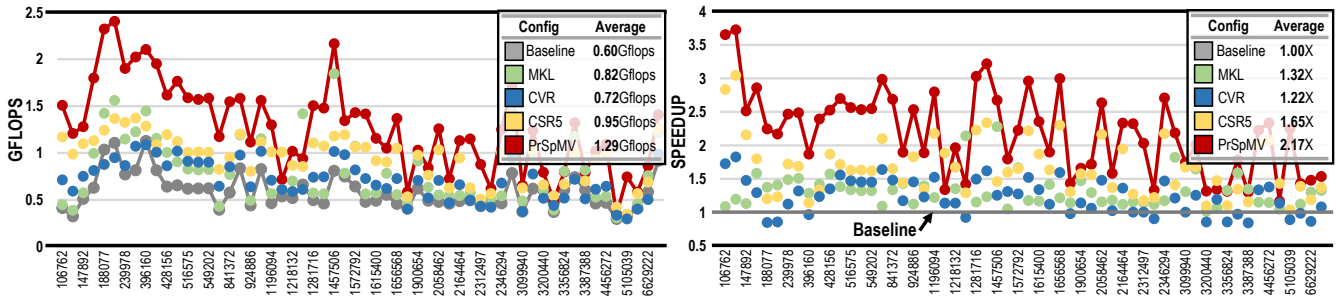


Fig. 12. Performance comparison of CSR baseline, MKL, CVR, CSR5 and PrSpMV(our method) on Intel Xeon CPU platform. Higher is better.

performance of stride prefetcher from average 0.52 GFLOPs to 0.68 GFLOPs, achieving 1.31× speedup, and improves IMP by 1.40×, indicating PrSpMV is an effective optimization for various prefetching techniques. Moreover, using PrSpMV with a stride prefetcher outperforms IMP by 1.31× speedup and Event-Trigger by 1.26× speedup. This means by using our PrSpMV kernel, a general commercial processor with stride prefetching can achieve better performance than dedicated prefetching techniques while not paying the costs of additional hardware and compilation overheads.

For more extensive studies, we also evaluate PrSpMV on current commercial CPUs. Fig 12 compares PrSpMV with multiple advanced SpMV kernels on the Intel processor, including CSR5 [11], CVR [12]. The original Intel Math Kernel Library (MKL) routine is used as baseline, and the optimized MKL routine is also compared. All the compared methods use Intel AVX-512 SIMD ISA to accelerate their computation. We can observe that PrSpMV achieves the best performance on all the tested matrices, and outperforms other approaches with great margins. On average, PrSpMV reaches 2.2× over baseline and 1.32× over the second best method (CSR5), mainly due to the optimization of data prefetching and branch prediction. Such results prove that PrSpMV can effectively improve SpMV performance on modern commercial processors.

D. Discussions

1) Ablation Experiment: As PrSpMV kernel uses several optimization methods, ablation experiments are conducted on 14 representative matrices to analyze the effects of different methods on the overall performance. Therefore, we design the following comparative schemes: (a) *PrSpMV-v1* with vector serialization and block partitioning; (b) *PrSpMV-v2* that improves *PrSpMV-v1* with bitmap row reordering; (c) *PrSpMV-v3* that augments *PrSpMV-v2* with branch optimization. The comparison results of these schemes are shown in Fig 13. It can be observed that the bitmap row reordering method increases performance by 15% because non-zeros are more regularly distributed so that the writeback overhead is reduced. Moreover, branch optimization can further increase 17% performance by removing most misprediction penalty.

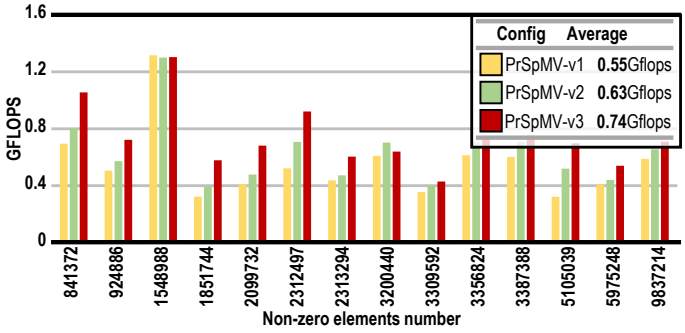


Fig. 13. Software ablation of PrSpMV. Higher is better.

2) Memory Bandwidth Utilization: The effect of PrSpMV can also be analyzed using the memory bandwidth utilization metric, as better prefetching typically leads to higher bandwidth usage. Our GEM5 CPU uses a DDR4_2400_16x4 memory model whose peak memory bandwidth is 14 GB/s. Fig 14 compares bandwidth utilization of different schemes before and after PrSpMV is used. We can observe that compared with the non-prefetching baseline, using stride prefetcher only slightly improves the bandwidth, but is still limited to only 5.8 GB/s as the CPU is blocked by long memory latency and cannot issue enough memory requests. However, when using PrSpMV with stride prefetcher, memory bandwidth is signifi-

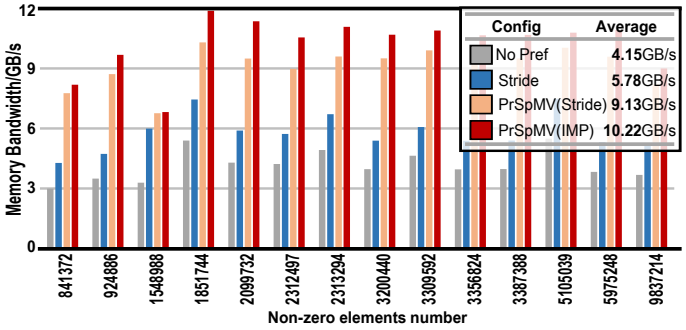


Fig. 14. Memory bandwidth utilization of different methods. Higher is better.

cantly improved by nearly 2×, reaching 9.1 GB/s. Moreover, using PrSpMV with IMP further increases memory bandwidth to 10 GB/s, which is close to the peak bandwidth. This is because IMP helps generate prefetching requests for the write-back memory accesses. This result further proves PrSpMV can be used to optimize existing prefetching techniques.

3) Multi-thread Scalability: Fig 15 shows the performance of single-thread to 8-thread executions on the Intel processor. We turn off the SMT technology of the Intel CPU to ensure the fairness of experiments and ensure all threads run on the same NUMA node by binding computing cores manually. It can be observed that PrSpMV achieves good scalability as the number of cores increases. It should also be noted that on 8-thread execution PrSpMV gains less than 8× speedup compared with single-thread. This is because the system memory bandwidth is likely to be fully saturated as the number of cores increases and the benefits are limited.

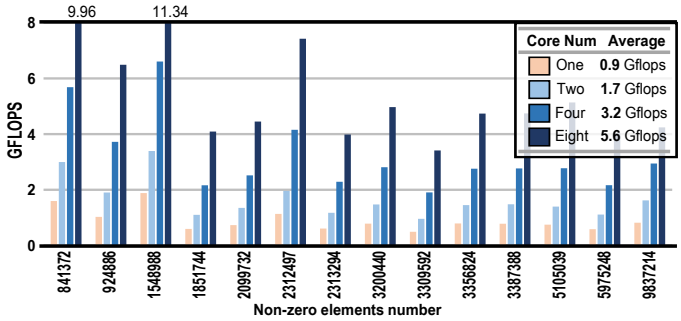


Fig. 15. Performance of PrSpMV with the different number of cores.

4) Preprocessing Costs: We measure the preprocessing costs of all optimization methods on the Intel processor, using the metric $\frac{t_{pre}}{t_{csr}}$, where t_{pre} is 1-thread preprocessing time of the optimization method and t_{csr} is 1-thread one-time SpMV execution time of CSR baseline. As shown in Fig 16, MKL has the most expensive preprocessing time on average 17.8× baseline runtime. CVR has the lightest preprocessing time on average 4.8× baseline runtime. PrSpMV has a modest preprocessing time on average of 11.2× baseline runtime. Considering hundreds of SpMV iterations in real-world scenarios, the preprocessing cost for PrSpMV is extremely low and can be easily amortized.

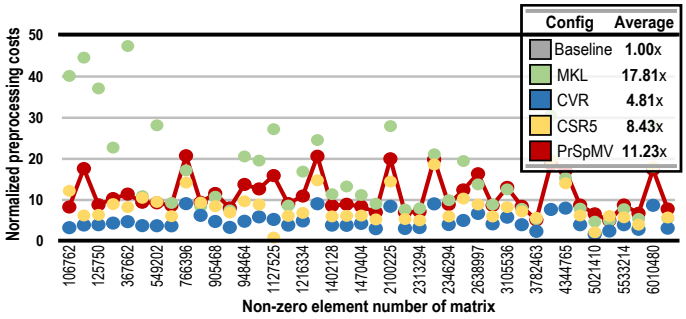


Fig. 16. Evaluation of preprocessing costs of different methods. Lower is better

V. Conclusion

In this paper, we propose PrSpMV, a novel SpMV kernel that is designed to achieve high efficiency of commercial processors. We serialize data accesses and provide a more regular branch pattern so that the current branch predictor and stride prefetcher can work efficiently. Experiment results show that PrSpMV can significantly reduce the average L2 cache miss rate from 57% to 20% via efficiently leveraging prefetchers. Moreover, PrSpMV with stride prefetchers outperforms existing state-of-the-art dedicated irregular hardware prefetchers with CSR format and gains further performance with other irregular prefetchers. The low preprocessing overhead indicates that PrSpMV is applicable to real-world scenarios. The PrSpMV kernel is available at https://github.com/moonstarslo/prSpMV-public.git.

VI. Acknowledgment

The authors thank the anonymous reviewers for their insightful comments and suggestions.

References

[1] J. Dongarra and F. Sullivan, "Guest editors introduction to the top 10 algorithms," *Computing in Science & Engineering*, vol. 2, no. 01, pp. 22–23, 2000.

[2] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, *et al.*, "The landscape of parallel computing research: A view from berkeley," 2006.

[3] S. Venkataraman, E. Bodzsar, I. Roy, A. AuYoung, and R. S. Schreiber, "Presto: distributed machine learning and graph processing with sparse matrices," in *Proceedings of the 8th ACM European Conference on Computer Systems*, pp. 197–210, 2013.

[4] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, "Imp: Indirect memory prefetcher," in *Proceedings of the 48th International Symposium on Microarchitecture*, pp. 178–190, 2015.

[5] S. Ainsworth and T. M. Jones, "An event-triggered programmable prefetcher for irregular workloads," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 578–592, 2018.

[6] T. Xia, G. Fu, C. Li, Z. Luo, L. Zhang, R. Chen, W. Zhao, N. Zheng, and P. Ren, "A comprehensive performance model of sparse matrix-vector multiplication to guide kernel optimization," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 2, pp. 519–534, 2022.

[7] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, "Efficient sparse matrix-vector multiplication on x86-based many-core processors," in *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pp. 273–282, 2013.

[8] W. T. Tang, R. Zhao, M. Lu, Y. Liang, H. P. Huyng, X. Li, and R. S. M. Goh, "Optimizing and auto-tuning scale-free sparse matrix-vector multiplication on intel xeon phi," in *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 136–145, IEEE, 2015.

[9] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.

[10] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, Y. Wang, E. Wang, Q. Zhang, B. Shen, *et al.*, "Intel math kernel library," *High-Performance Computing on the Intel® Xeon Phi™: How to Fully Exploit MIC Architectures*, pp. 167–188, 2014.

[11] W. Liu and B. Vinter, "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.

[12] B. Xie, J. Zhan, X. Liu, W. Gao, Z. Jia, X. He, and L. Zhang, "Cvr: Efficient vectorization of spmv on x86 processors," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pp. 149–162, 2018.

[13] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, pp. 1–25, 2011.

[14] J. Leskovec and A. Krevl, "Snap datasets: Stanford large network dataset collection," 2014.

[15] A. Seznec, "A new case for the tage branch predictor," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 117–127, 2011.