Streaming Sparse Data on Architectures with Vector Extensions using Near Data Processing

Pranathi Vasireddy
University of North Texas
Denton, Texas, USA

Krishna Kavi
University of North Texas
Denton, Texas, USA

Alex Weaver
University of North Texas
Denton, Texas, USA

Gayatri Mehta
University of North Texas
Denton, Texas, USA

ABSTRACT

Most scientific and AI applications rely on data that is usually sparse. Sparsity is sometimes deliberately introduced with pruning of insignificant and redundant connections in the neural network layers. This sparse data can efficiently be represented through various compression formats, thereby, reducing storage footprint and eliminating computations with zero values. However, accessing nonzero values introduces extra computational overhead.

Consider sparse matrix-dense vector multiplication (spMV), where the sparse matrix data is represented in CSR format. The irregular memory loads caused by CSR format are particularly vexing for vectorized computations (e.g., modern CPUs with SVE or AVX extensions, GPUs) because they do not exploit spatial and temporal localities. In this work, we propose a memory-side accelerator known as *MAID* (Memory-side Acceleration for Irregular Data), which is a simple integer unit that performs these irregular memory accesses and streams the aligned data to the primary core. The primary vectorized CPU focuses on only fetching these aligned values and computing multiply-accumulate operations concurrently alongside our MAID accelerator. To accurately evaluate the performance effects of our proposed accelerator in a *CPU-MAID-memory* system, we used a Gem5 extension that permits simulating near data processing (NDP) accelerators close to memory devices at various levels of the memory hierarchy. Our experimental evaluations show that when MAID is placed close to L2 or LLC, minimizes data transfer to L1 cache, eliminates cache conflicts, and thus increases energy savings. We observed performance gains ranging between 2.76 and 3.56 times over in-order SIMD-CPU and an average of 2 times for out-of-order SIMD-CPU for sparse matrix * dense vector (spMV) algorithm. We also evaluated our *MAID* with sparse matrix * sparse vector (spMspV) and observed gains as high as 5.3 times over SIMD baseline.

CCS CONCEPTS

• **Hardware → Hardware accelerators**; • **Computer systems organization → Embedded systems**; **Single instruction, multiple data**; **Reconfigurable computing**; **Heterogeneous (hybrid) systems**.

KEYWORDS

Near-memory computing, SIMD processors, sparse linear algebra, neural networks, edge devices, Gem5

1 INTRODUCTION

Neural networks are being adopted in wide range of domains ranging from day-to-day tasks such as voice-activated assistants to scientific applications such as predictive models, and even in computer architecture designs such as neural network-based branch predictors. Neural networks, however, involve complex computations on large data sets and deploying them on resource-constrained systems and devices can be challenging. On the other hand, in most cases the data sets for many of the applications are sparse. Often the sparsity is further enhanced by sampling and pruning redundant or insignificant connections in weight or activation matrices/tensors. This reduces the arithmetic computational complexity of the neural networks and increases the ability to handle large data sets without losing accuracy significantly. Previous studies have used various sparse data formats to store only the nonzero values that can improve cache utilization and reduced memory footprint. However, these compression formats introduce overheads for accessing nonzero values of the sparse matrices. This is commonly observed in most sparse linear algebraic algorithms including Sparse Matrix-Matrix multiplication (spMM), Sparse Matrix dense Vector multiplication (spMV), Sparse Matrix Sparse Vector multiplication (spMspV), convolutions, transposition and factorization. Domain-specific libraries and frameworks like GraphBLAS [12] for graph applications, PETSc [6] for scientific computations and cuSPARSE [1] for GPU-acceleration provide optimized software for these common algorithms but the underlying metadata processing overhead in accessing nonzero data due to compressed representation of the sparse data still exists. The distribution of nonzero values in input sparse matrices lead to irregular memory accesses, which limit the effectiveness of the compile-time optimizations provided by these libraries. This decompressing overhead shifts the bottleneck of these algorithms from being computationally intensive tasks to memory intensive tasks.

For this reason, memory-side hardware accelerators such as Processing-in-memory (PIM) or Near Data Processing (NDP) have gained attention. Studies have indicated significant performance improvements using these accelerators compared to traditional compute models that rely on frequent data transfers between memory and processing units [4, 30, 40, 41].These studies, however, focus on accelerating the entire algorithm or computational kernel (both memory accessing and arithmetic tasks) which requires the accelerator to be complex and equipped with parallel floating point arithmetic units, which may cause issues with power and energy requirements for near data or in memory processing, and may lead to under utilization of primary CPU cores.

Instead, we propose a NDP accelerator called *MAID* (Memory-side Acceleration for Irregular Data) that only performs computations needed to locate required nonzero values and provide them to primary processing elements for algebraic computations. *MAID* can be implemented either as a small in-order processor with only integer functional units, or customized hardware (ASIC). *MAID* can be programmed to perform "gather" operations for different sparse representations and different access patterns (e.g., row-wise, column-wise or irregular) based on the algorithm. By offloading the memory-intensive tasks to *MAID*, the primary core can focus on computationally-intensive tasks alone, leveraging the full capabilities of vectorized (SIMD) instructions. This creates an overlap between the *MAID* and the primary core resulting in potentially better cache utilization and reduced power consumption.

There are several studies that propose intelligent and programmable data prefetchers, particularly for applications that rely on irregular data structures including linked lists and sparse data representations such as CSR. For example, IMP [39] proposes hardware support for prefetching data items that involve indirect accesses such as *m[v[j]]*, which represent accessing elements of a vector based on the location of nonzero values of matrix rows in sparse matrices using CSR-based format. We would like to point out that while our *MAID* has the effect of prefetching data for processing, *MAID* should not be considered merely as a prefetcher. In general IMP [39] and other prefetchers only aid in prefetching data, including prefetching data that is not needed or used by the processor while *MAID* can be programmed to supply *only needed* data, including intelligence regarding array bounds. *MAID* can be programmed to match nonzero values of a sparse matrix with nonzero values of a sparse vector in sparse matrix - sparse vector (spMspV) computations, which is very difficult to achieve with conventional prefetchers. Unlike other NDP accelerators, *MAID* performs equally well with floating point and quantized data because it solely operates on integer indices of nonzero data and doesn't perform computations on the data itself.

Integrating NDP near main memory of a computing system to benefit from the large storage capacity of the DRAMs can reduce the need for frequent data movement across the memory hierarchy. However, since the decompressed or reformatted data generated by the NDP is not reused, integrating *MAID* into the main memory may not provide significant benefits when compared to placing it closer to the last level cache (LLC). By placing *MAID* in or near a LLC, *MAID* can take advantage of the cache's faster access times and potentially reduce the energy consumption. This close proximity to LLC can reduce cache pollution at L1 cache since *MAID* handles metadata (indexes of nonzero values) and they are not brought into L1 cache. Additionally, the decompressed (or sparse) data processed by *MAID* can bypass the intermediate caches and be pushed directly into the L1 cache of the CPU. This further reduces the data movement across the cache hierarchy and can improve performance due to the lower latency and increased data locality of L1 cache. We feel that different applications and irregular (or sparse) data representations may benefit from *MAID* like devices being placed at L1, LLC or DRAM. The Gem5 [8] based framework developed in our research can permit the study of these alternatives. Gem5 provides a flexible platform for modeling and simulating various configurations of a full-system and permits valuable insights into the performance improvements such as latency and cache misses, data movement reduction, and analyze potential trade-offs associated with introducing *MAID* in the CPU-memory system.

The contributions of this paper include:

- *Our extensions to Gem5 based NDP framework [36] to permit the inclusion of* MAID *(or other memory accelerators) along the memory hierarchy.* In the Gem5-NDP framework [36], the NDP is connected between the processor and the memory, causing every memory access to pass through the NDP, and the NDP has to determine whether the response is intended for the CPU or for itself. This can introduce delays for the CPU if the NDP is occupied with other tasks. To mitigate these unnecessary delays, we established separate connections to the L1 cache from CPU and NDP, eliminating the need for prioritization and preventing unnecessary delays in memory responses from memory to CPU. While our *MAID* can provide aligned data (matching nonzero values) through scratchpad memories if available, our *MAID* can also store the data directly in L1 cache so that the primary core can use conventional addressing to access this data. We further extended the Gem5 cache memories to permit *MAID* to lock some cache lines to prevent eviction of the data generated by *MAID* before being consumed by the primary core. Such a mechanism can be used when the processors are not equipped with scratchpad memories.
- *Our detailed simulations of* MAID *as processing hardware placed along the memory hiearchy near L1, L2/LLC or DRAM memories.* We demonstrated that *MAID* can aid in handling sparse and irregular data for Sparse Matrix * Dense Vector (spMV) and Sparse Matrix * Sparse Vector (spMspV) computations. We analyzed the impact of *MAID* in terms of performance gains with both in-order and out-of-order processor using vectorized (SIMD) instruction extensions. We also evaluated the impact of *MAID* on the number of cache accesses and cache misses. We show when *MAID* is placed at LLC or DRAM, indexing data (or metadata) is not brought into processor's L1 cache since the primary core does not require the metadata.
- *Our results show that modern out-of-order processors will not perform well when dealing with irregular data, particularly if the data needed depends on other factors, as is the case with spMspV computations: the nonzero values of sparse matrices must be matched with the nonzero values of sparse vectors.* While out-of-order execution can hide memory latencies by issuing several load requests, such is not the case with spMspV. We show that using a simple integer *MAID* for aligning data (for both spMV and spMspV) is a better use of resources than using a powerful out-of-order core computing indexes and memory addresses.

2 BACKGROUND

Sparse compression formats. Many real-world data sets are very large but sparse while other data-intensive applications like neural networks introduce sparsity through pruning. Traditional or dense representation of data can result in wasted memory space and computations involving zero values. Several compression representations have been proposed for compact representation of only nonzero data. Compressed Sparse Row (CSR) format is widely employed in many domains and applications. In CSR format, sparse matrix M is represented using three arrays: *M_cols* stores the column indexes of the nonzero values in each row, *M_vals* stores all the nonzero values of M ordered by rows, and *M_rows* stores the starting position of first nonzero element of each row of the matrix M in *M_cols* and *M_vals* arrays as shown in Figure 1.

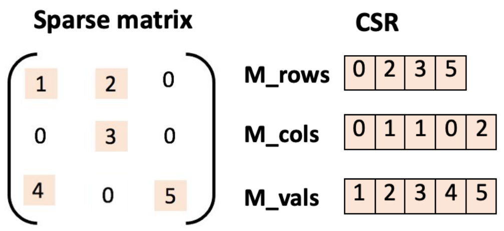


Figure 1: A 3x3 sparse matrix in CSR Format

Sparse Linear Algebra. Two fundamental operations that are commonly used in most neural networks, graph analytics and scientific applications are spMV (sparse matrix - dense vector) and spMspV (sparse matrix - sparse vector) multiplications. spMV algorithm exploits the sparsity of the matrix to reduce the number of multiplications while spMspV exploits the sparsity of both matrix and vector (this can be a component of a sparse matrix - sparse matrix multiplications). Both algorithms compute pairwise multiplications of elements from matching columns (of rows of one matrix), with the corresponding (nonzero) elements of the vector. Consider the spMV algorithm on a sparse matrix M and a dense vector v that produces a resulting dense vector y. Since M is sparse, it is usually represented in one of the above discussed compression formats. We use CSR format here for illustration where the location of each nonzero in the matrix is determined using the *M_cols* array and the *M_rows* array is used to determine the number of nonzeros (*nnzs*) in each row. As shown in Algorithm 1, the algorithm iterates row by row over the matrix M, calculating the *nnzs* for each row and performing multiply-accumulate (MAC) of the nonzeros. The indirect accesses to vector v are introduced to eliminate the computations over non significant data as shown in line 8 of Algorithm 1.

Algorithm 1 CSR Version of *spMV*

```
1:  procedure SPMV($M\_rows$, $M\_cols$, $M\_vals$, n, v)
2:      $s \leftarrow 0$
3:      $k \leftarrow 0$
4:      for $i = 0; i < n; i = i + 1$ do
5:          $nnzs \leftarrow M\_rows[i+1] - M\_rows[i]$
6:          $s \leftarrow 0$
7:          for $j = 0; j < nnzs; j = j + 1$ do
8:              $s \leftarrow s + M\_vals[k+j] * v[M\_cols[k+j]]$
9:          $k \leftarrow k + nnzs$
10:         $y[i] \leftarrow s$
```

In the spMspV algorithm, the vector is also sparse and can be represented in memory by storing only the nonzero vector values (*V_vals*) and their indexes (*V_cols*). However, this results in an additional computation step to check that both the matrix and vector values are nonzeros before each multiply-accumulate (MAC) operation. Using compressed data for spMV and spMspV can significantly reduce the number of MACs compared to a dense algorithm and achieve storage compression. But the overhead of irregular accesses due to the compression format increase the number of load instructions per iteration resulting in achieving lower than expected performance gains [2].

Vectorized or SIMD Instructions. Modern processors like ARM Cortex-M series [32], Intel Skylake [13] and AMD Zen 3 architectures [15] incorporate SIMD or vector instructions through Scalable Vector Extensions (SVE) or Advanced Vector Extensions (AVX) to enable simultaneous execution of the single instruction on multiple data elements, often packed into vector registers. Typically ARM processors provide support for vector registers with varying lengths ranging from 128 to 2048 bits (can be viewed as 2 to 32 SIMD lanes). When performing repetitive operations of linear algebraic computations on dense arrays, SIMD instructions excel at exploiting data parallelism. This leads to faster execution and improved throughput. To further optimize the irregular data handling and reduce power consumption, SIMD instructions utilize predicates to selectively turn on/off the SIMD lanes for a particular computation based on the data pattern. ARM (and x86) processors also support SIMD Scatter/ Gather instructions to improve irregular read/write accesses to non-contiguous memory locations. This may be done efficiently for spMV computations where the column indexes of nonzero matrix values can directly be used to obtain the required vector values. However, it should be noted that the Gather instructions still need to access several cache lines to assemble (or distribute) data needed to pack for SIMD operations. Since these instructions are executed by the primary processing cores, these memory accesses may pollute L1 data caches. Moreover, it is unclear how Gather operations work if the vector is also sparse (for spMspV computations). *Our* MAID *can eliminate many of these deficiencies of Gather instructions. We will show that since* MAID *is a separate unit, all memory accesses needed for "Gather" are not brought into CPU L1 data caches. We will also show that* MAID *is also very effective in "Gathering" data when both matrix and vector are sparse.*

3 FRAMEWORK

In this work, we propose a small in-order core or integer-only ASIC memory-side accelerator called *MAID* that operates near the LLC[1]. Since the main objective of *MAID* is to handle memory index operations for the primary core, it does not require floating-point capabilities like a full-fledged processor. By solely focusing on this data gathering task, *MAID* can effectively work with even modern out-of-order processors with vector (SIMD) instructions, thereby enhancing overall performance of applications using irregular data. Due to its reduced hardware requirements, as the number of cores

[1]We explored *MAID* placed near L1 cache as well as near DRAM. Our experiments show that *MAID* for spMV and spMspV computations works best when placed near LLC.

increases, we can achieve scalability by equipping each processor with its own *MAID*. We evaluated the performance of our *MAID* accelerator alongside both in-order and out-of-order (O3) superscalar primary cores for spMV and spMspV computations.

3.1 MAID Design

MAID is designed as a programmable hardware (either as a simple core or as a coarse-grained reconfigurable hardware) to support different algorithms with only integer units and a small local memory, working alongside and concurrently with the primary CPU cores, aligning data needed for the SIMD (or vectorized) execution. The aligned values are communicated to the CPU using buffers, scratchpad memory or L1 data cache (L1 Dcache). *MAID* fills the buffer with aligned values and sets a flag associated with the buffer, notifying the primary CPU that the aligned data is ready for consumption. The CPU can then start reading the values from the buffer using vector load instructions and process these values using vector arithmetic operations. Concurrently, *MAID* aligns the next set of values and updates the second buffer and so on. Once the CPU consumes all the values in a buffer it resets the buffer flag, releasing the buffer to *MAID*. In spMV algorithm, *MAID* only provides the aligned vector values for the CPU (*V_vals[.]*) as shown in Algorithm 2. CPU fetches the nonzero matrix values (*M_vals*) and the *V_vals* provided by *MAID* using vector loads as shown in Algorithm 3. In the case of spMspV, where the sparse matrix and sparse vector are represented using CSR format, *MAID* has to find the matching indexes for nonzero values in sparse matrix and nonzero values of sparse vector. Only when both of them are nonzeros, *MAID* fetches the corresponding matrix and vector values and supplies them to the CPU through buffers.

Algorithm 2 MAID code for CSR Version of *spMV*

```
1: procedure spMV MAID ($M\_rows$, $M\_cols$, v, num_rows)
2:     $k \leftarrow 0$
3:     for $i = 0; i < num\_rows; i = i + 1$ do
4:         $nnzs \leftarrow M\_rows[i+1] - M\_rows[i]$
5:         for $j = 0; j < nnzs; j = j + 1$ do
6:             $V\_vals[k] \leftarrow v[M\_cols[k]]$
7:         $k \leftarrow k + 1$
```

Algorithm 3 CPU code for CSR Version of *spMV*

```
1: procedure spMV CPU ($M\_rows$, $M\_vals$, $V\_vals$, num_rows)
2:     $k \leftarrow 0$
3:     for $i = 0; i < num\_rows; i = i + 1$ do
4:         $nnzs \leftarrow M\_rows[i+1] - M\_rows[i]$
5:         $y[i] \leftarrow 0$
6:         for $j = 0; j < nnzs; j = j + 1$ do
7:             $y[i] \leftarrow y[i] + M\_vals[k+j] * V\_vals[k+j]$
8:         $k \leftarrow k + 1$
```

In our previous works [2, 3, 34, 37], we proposed hardware architecture for our *MAID*-like accelerator called HHT (Hardware Helper Thread) to perform indexing, similar to this research. However, the focus there was on micro controllers where the primary core is a very simple in-order RISC V-like processors; which makes it easier for the helper thread to meet the demands of the primary core. The hardware helper was placed close to the processor, possibly at L1 cache (if one is available). The study reported here assumes the *MAID* hardware similar to that of HHT [2, 3, 34, 37] but is much broader in its scope and explores near data processing hardware that can be placed at any level in the memory hierarchy. Additionally, we explore the use of the hardware accelerator with modern out-of-order superscalar processor. We also show the benefits of our programmable accelerator for both spMV (dense vector) and spMspV (sparse vector) computations.

3.2 Gem5-NDP Framework

We used a cycle accurate Gem5-NDP simulator [36] in Gem5 System Emulation (SE) mode to perform detailed architectural exploration and performance analysis of our *MAID* accelerator in a CPU-memory system. Gem5-NDP permits a NDP device to be connected to any memory level (including L1, L2, LLC caches or DRAM) in the system through a high-level Python configuration script. NDP is connected in between CPU and memory hierarchy and CPU uses Programmable IO interface that consists of register banks to manage NDP device. Hence NDP should always be connected to the CPU through the dcache_port on the CPU as shown in Figure 2a. NDP device in this framework has 3 ports namely; cpu_side port that connects to the CPU through its dcache_port, mem_side port which connects to the CPU's L1 Dcache and dma_port which is used to connect NDP to various memory levels. NDP uses DMA port to perform memory read and write operations and the mem_side port to establish a connection between CPU and the memory. As a result, all memory requests of CPU are routed through NDP. This introduces additional latency for the CPU to receive the responses from memory and prioritize these responses in real hardware. To avoid this, we created an additional port called ndp_port on CPU to facilitate direct access to the memory for both CPU and NDP. This is achieved by assigning a small specified address range for the ndp_port to distinguish from the dcache_port on CPU side [2]. During simulation, any read or write operations within this address range will trigger functions in Gem5 to update NDP registers emulating memory mapped device functions. So, in the modified framework, NDP is connected to CPU through ndp_port as shown in Figure 2b instead of dcache_port like in original framework. This additional port is only active to initialize the function to be executed and supply base addresses for the sparse matrix and vector arrays in CSR format through registers. Once initialization is complete, this port is inactive. *MAID* needs compressed format data for irregular value alignment and it directly reads this information from the LLC rather than using the L1 cache to avoid unnecessary cache pollution. *MAID* in spMV and spMspV algorithms aligns values exclusively for the CPU consumption. Therefore, it is unnecessary to write these temporary values into LLC all the time that has higher latency. If there is a scratchpad memory associated with the CPU then *MAID* can write directly into this memory. If there is no scratchpad, *MAID* can write directly into the L1 cache of the CPU as shown in Figure 2b using the mem_side port. Since this data is temporary and needs

[2]The NDP memory map is defined by the primary core so that these addresses do not have to be hard-coded.

to be available in L1 cache only until CPU utilizes it, we tested with locking few cache lines of the CPU's L1 cache based on the buffer size that is being used to communicate between CPU and *MAID*. This converts the few locked cache lines of L1 cache into a temporary scratchpad for the CPU. This may result in a small increase in overall cache misses in L1 but unlikely to impact the overall performance. -

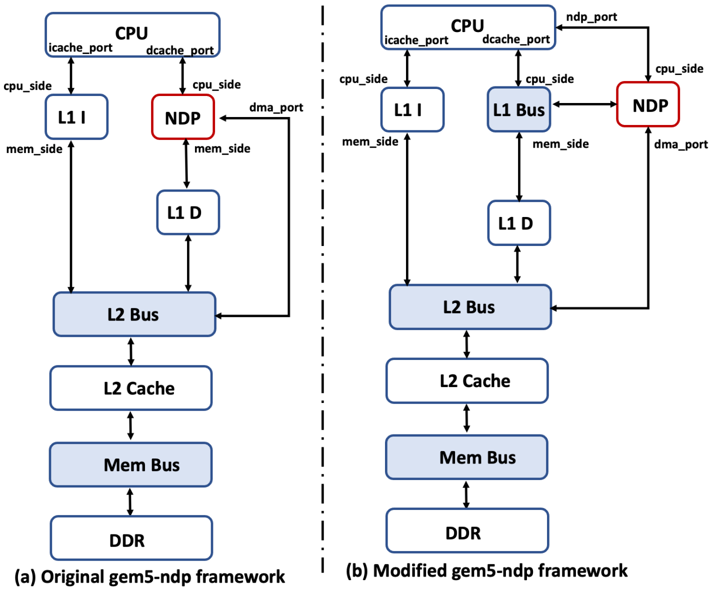


Figure 2: Gem5-ndp framework

4 EXPERIMENTAL SETUP

We evaluated the performance benefits of our *MAID* accelerator in a CPU - memory hierarchy using Gem5-NDP as described in Section 3.2. Table 1 describes the system configuration used in our experiments. *MAID* performs only memory addressing computations and hence is designed as an integer programmable accelerator that can be used with both spMV and spMspV (as well as others) algorithms and can be tuned for different workloads using initialization registers. *MAID* is also configured with its own small local memory of 8KiB with a latency of 5 cycles similar to that of primary CPU's L1 cache to efficiently hold 512 nonzero vector values and 512 column indexes of a sparse matrix. This minimizes the constant retrieval of data from LLC for each value, even if the *MAID* is near the LLC. We used randomly generated sparse matrices with varying sparsities ranging from 10% to 90% in steps of 10% and presented the results using 512x512 matrix in Section 5. This approach allows for a better understanding of the effects of sparsity, akin to a controlled experiment, as real-world matrices often fall within this range. For spMspV, we assumed that the vector has same level of sparsity as the matrix in each workload. *MAID* can run larger matrix workloads by utilizing the Block CSR (BCSR) format [9], which involves dividing the matrix into 512x512 sub-matrices. This can be seamlessly integrated with *MAID* without necessitating any modifications to the existing configuration.We collected total execution times, L1 cache misses, LLC cache misses and CPU idle or wait cycles to evaluate the effectiveness of of *MAID* alongside both in-order and out-of-order primary core using TimingSimpleCPU and DerivO3CPU models for primary core in Gem5 System Emulation (SE) mode.

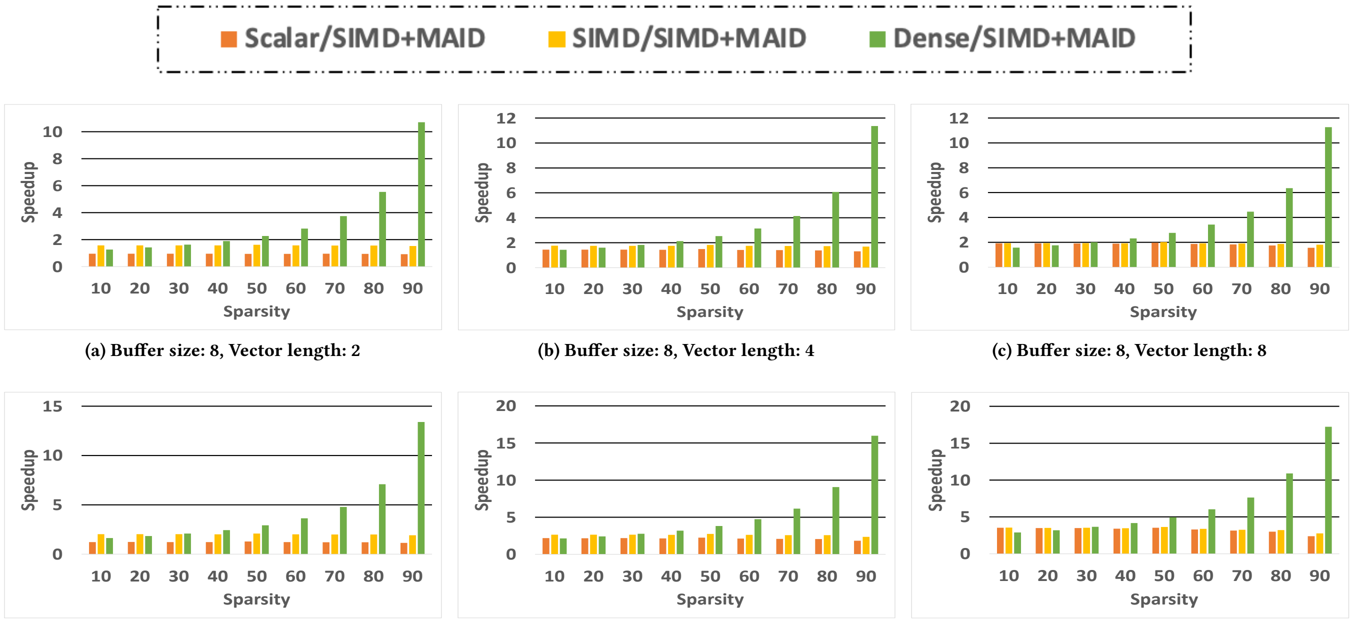
Table 1: System Configuration

Processor	Values
Core	ARMv8 ISA with SIMD Extensions Frequency = 2 GHz Vector width (VL) = 2, 4, 8 Elements Element Size = 64 bit In-order and Out-of-order
MAID	N=2 Buffers Buffer size = 8 to 128 Elements Element size = 64 bit Local Memory = 8KiB bytes, 5 cycles
Cache Configuration	Cache line size = 64 bytes
L1 ICache	32KiB, 2 way, 5 cycles
L1 DCache	32KiB, 4 way, 5 cycles
L2 Cache	256KiB, 16 way, 14 cycles
Main memory	DDR3-1600-8x8, 2GB

Buffer Sizes and Vector Lengths: In this contribution, the size of the buffer indicates the number of vector values corresponding to nonzero values in a row of the sparse matrix for spMV computations and the number of matching nonzero matrix and vector values when both the matrix and vector are sparse (for spMspV computations). The buffer size in our results indicate the number of elements (each being 64 bits) assembled by *MAID*. On the other hand, the vector lengths refer to the width of vectorized units (or SIMD lanes). For example, vector length of 2 means that the processor is using 128 bit (2-wide) vector registers. There is a tradeoff between buffer sizes and vector lengths: larger buffer sizes may lead to longer CPU wait times since CPU must wait until the buffer is filled by *MAID* and the values are released for consumption by CPU. However, shorter vector lengths with larger buffer sizes mean that CPU will consume the values in the buffer over several iterations of vector operations. For example, when buffer size is 8 (or 512 bits) and vector length is 2, CPU takes four iterations to consume all 8 buffer values. We use two buffers (i.e., double buffering) to achieve overlap between CPU and *MAID*. On the other hand, if the buffer size is too small compared to the vector length, it results in increasing CPU wait cycles as all SIMD lanes have to be filled before CPU executes the vector instruction. In our studies, we have opted for a buffer size that is at least equal to the vector length. In future, we plan to explore the use of multiple circular buffers instead of larger buffer sizes.

5 RESULTS

In this study, we evaluated the performance improvements attained by adding our *MAID* accelerator to a SIMD processor against 3 different baselines of CPU alone handling both data gathering and computational tasks. These baselines include (a) Scalar CPU (with no vectorized instructions), (b) Sparse SIMD CPU (vectorized instructions on sparse data) where the CPU performs all computations including indirect memory addressing and (c) Dense SIMD CPU that represents both zero and nonzero values, possibly wasting computations on zero values. We present the *MAID* accelerator speedup



(a) Buffer size: 8, Vector length: 2

(b) Buffer size: 8, Vector length: 4

(c) Buffer size: 8, Vector length: 8

(d) Buffer size: 128, Vector length: 2

(e) Buffer size: 128, Vector length: 4

(f) Buffer size: 128, Vector length: 8

Figure 3: Performance analysis for 512x512 spMV on In-order core: Scalar, SIMD-only sparse and SIMD-only dense baselines vs SIMD with MAID acceleration for various buffer sizes and vector lengths.

gains against both in-order and out-of-order baseline CPUs. Since our goal is to offload the irregular data gathering task to *MAID* and overlap this work with CPU computations, we analyzed CPU idle cycles while waiting for the data from *MAID*. The results presented in the later sections use a 512x512 sparse matrix represented in CSR format with varying degrees of sparsity (10% to 90%) for both spMV and spMspV algorithms. Since SIMD processors can support various vector register widths, we experimented with vector lengths ranging 128, 256 and 512 bits which correspond to 2, 4 and 8 values respectively (each data items is a 64-bit long integer). Data type (integer or floating point) is not a concern for *MAID* since it does not directly perform computations on the data and only packs the data in a buffer for the primary core (the minimum size of the buffer is the vector width).

5.1 *MAID* aiding In-order processors on spMV workload

In this section, the primary core is assumed to be an in-order CPU and *MAID* as a single stand-alone integer functional unit with a small dedicated local memory. This local memory of size 8KiB is used to store column indexes of the row being processed and dense vector values. *MAID* aligns only one value at a time and the latency for aligning each value is dependent on the local memory access latency and instruction latency (for example, obtaining a matching vector value in spMV applications involves 2 memory accesses: one for column index and one for vector value). It should be noted that when the processor is in-order, without the aid of *MAID*, scalar CPU works better than SIMD CPU (i.e., with vector instructions) for irregular memory accesses as observed in Figure 3 since SIMD heavily relies on gathering sequential data. Irregular accesses create load imbalances since all SIMD lanes have to be filled with data causing additional overhead in data alignment. Also, at higher sparsities irregular memory accesses are spread across multiple cache lines and as vector length increases, the cache utilization decreases as there can be only be a single hit per cache line and to fill all SIMD lanes, different cache lines need to be fetched. Since scalar processor works with only one value at a time, cache misses are reduced and as well as the processing time. Only at lower sparsities and higher vector lengths, we observe similar gains for both scalar and SIMD CPU baselines. As the number of nonzeros decreases (sparsity increases), scalar baseline performs better, even at higher vector lengths since cache utilization decreases. *MAID* accesses one vector data element at a time corresponding to the nonzero column index of the sparse matrix. But *MAID* assembles these values for CPU and works concurrently with CPU which uses the values supplied by *MAID* to perform necessary computations such as Multiply-Accumulate (MAC) operations. This leads to performance gains when compared to SIMD CPU baseline (without *MAID*). Also since the aligned values are supplied to the SIMD CPU, use of *MAID* can reduce primary CPU's L1 cache misses. Since dense SIMD CPU ignores data sparsity, it performs all 512x512 MAC operations. At lower sparsities (e.g., 10%) the number of wasted zero value multiplications is minimal and acceptable, but as the sparsity increases, dense formulation incurs too many wasted zero value multiplications. On the other hand, sparse data representations require indexing overheads for locating vector values corresponding to the location of nonzero values of the matrix. These overhead computations can be excessive at low sparsities. We observe that SIMD CPU with *MAID* using CSR data representations can achieve nearly 10x to 17x speedup over dense (CPU only) formulations at 90% sparsities.

As vector register widths (or SIMD lanes) increase, speedup increased with SIMD + *MAID* (SIMD CPU aided by *MAID*) over scalar when *MAID* uses buffer sizes of 8 (see Figure 3a, 3b, 3c). *MAID* aligns 8 vector values and writes them into L1 cache of the primary CPU. If the vector register width is smaller than the provided data (or buffer size) then CPU can access the data to satisfy multiple SIMD computations. In the meantime this data may be evicted from L1 by other instructions. But as vector length increases, most of the data provided will be utilized in a single or very few SIMD cycles. Thus it is necessary to set the buffer sizes based on the vector widths. At buffer size of 8 and vector length of 2, SIMD+*MAID* worked similar to a scalar processor with no gains. But as vector length increases, speedup increased as shown in Table 2. This table summarizes the results shown in Figure 3. Since *MAID* writes into L1 of CPU, and all loads are regular L1 accesses, SIMD+*MAID* gains over SIMD CPU-only baseline which may have to bring multiple cache lines to fill the vector registers. Speedup also increases as the buffer size increases. The speedup becomes more prominent when the vector length increases as more values are processed at once. Since dense CPU works irrespective of sparsity in data, all loads are sequential similar to what is achieved with the *MAID* for a SIMD CPU. However, *MAID* reduces the number of computations as sparsity increases (eliminating zero values) but the execution times for dense baseline do not change. As buffer size increases to 128 elements, gains increase to almost 4x with vector length as 8 compared to scalar processor and vector processor as shown in Figure 3.

To summarize, *MAID*, in conjunction with an in-order SIMD CPU, demonstrates the potential for scalability through varying vector lengths, resulting in performance gains. Increasing buffer size effectively mitigates CPU and *MAID* wait times and enhances the overlapped execution of CPU and *MAID*, increasing the overall performance. These results lead us to believe that it may be possible to consider *MAID* like accelerators with SMP cores inside GPUs, whereby the *MAID* aligns sparse data for consumption by the threads of an SMP, since these threads are in-order processors. We will explore these ideas in our future research.

5.2 *MAID* aiding Out-of-order processors on spMV workload

In this section, we analyzed the performance of *MAID* alongside an out-of-order primary core. In in-order processing, the CPU's pipeline stalls until all the vector register values are available for the SIMD instruction. The out-of-order processor hides this latency by reordering instructions. Thus out-of-order cores with SIMD instructions outperform those without SIMD instructions (i.e., scalar cores) as observed in Figure 4. For out-of-order processing, we observe from Figures 4a, 4b and 4c, no speedup when compared to the dense baseline at lower sparsities for buffer size of 8. At lower sparsities, the number of zero computations are insignificant and thus it may be better to use dense representations of matrices, avoiding the need for indexing using CSR formats which may cause irregular memory accesses. Once *MAID* starts providing more values (using a larger buffer size) and CPU uses larger vector lengths then we observe slight gains over dense baseline. These gains are more prominent with larger vector lengths than shorter vector lengths since shorter vector lengths (but larger buffer sizes) may cause *MAID* to wait for CPU to consume all the values in a buffer. As sparsity increases, dense baseline performs poorly compared to all other configurations because of the increasing number of wasted computations on zero values. When the buffer size and vector length are both 8, all values produced by *MAID* are consumed by the CPU in a single iteration. Figures 4a, 4b, 4c show that with buffer size set to 8, performance decreases as the vector lengths increase from 2 to 8. On the other hand, when buffer size is set to 128, CPU has more data provided and speedups are higher as shown in Figures 4d,4e and 4f.

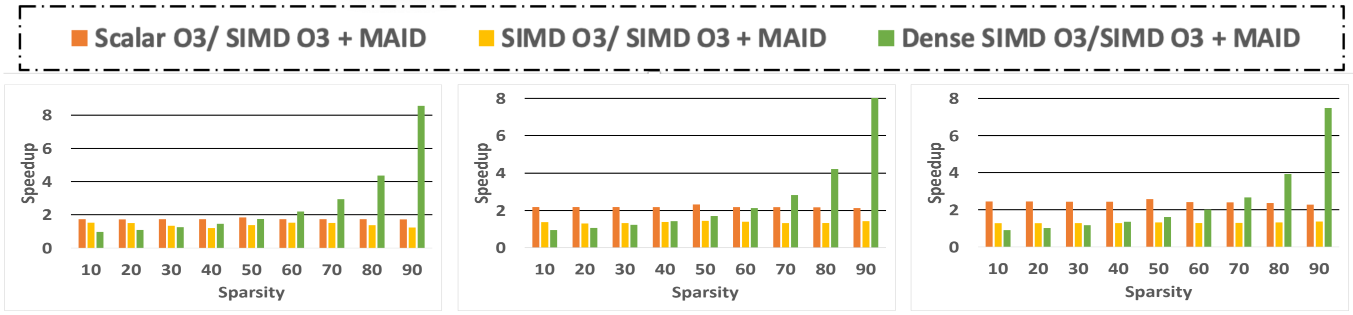
In summary, out-of-order CPU cores effectively conceals latency caused by irregular memory accesses since the processor can issue and execute instructions out of order and in the case of processors with vector instructions, the performance scales with the vector widths, and achieves favorable gains against dense baseline at lower sparsities. The performance gains in the out-of-order setup can be further improved by adding *MAID*, particularly with larger buffer sizes and wider vector registers as shown in Table 3. These gains are higher as the sparsity in the matrix increases. Rather than increasing the buffer size, a more effective strategy would be to consider each large buffer as multiple smaller buffers, with each buffer size set equal to the vector length, similar to circular buffer schemes. This approach reduces the wait time of the CPU associated with larger buffer sizes, and CPU and *MAID* can achiever greater overlapped executions with multiple buffers.

Table 2: Speedup against In-order baselines on a 512x512 matrix with 10% to 90% sparsity.

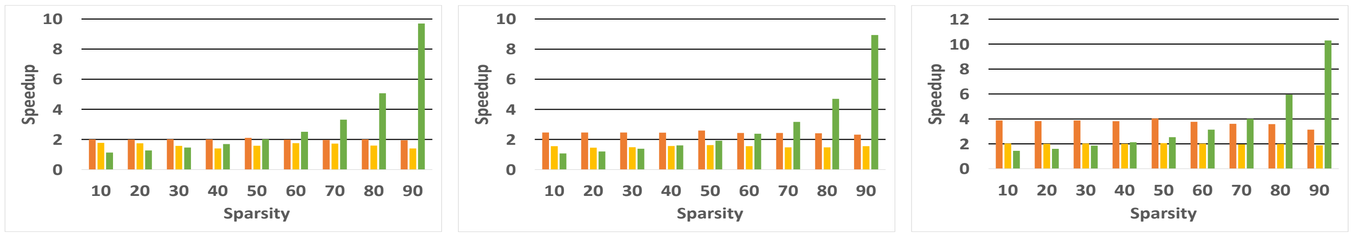
	Buffer size: 8			**Buffer size: 128**		
Vector length	**2**	**4**	**8**	**2**	**4**	**8**
Scalar CPU	0.95x to 0.92x	1.44x to 1.3x	1.92x to 1.56x	1.23x to 1.15x	2.18x to 1.83x	3.53x to 2.39x
SIMD CPU	1.57x to 1.53x	1.75x to 1.68x	1.94x to 1.80x	2.03x to 1.91x	2.65x to 2.36x	3.56x to 2.76x
Dense CPU	1.27x to 10.70x	1.42x to 11.36x	1.56x to 11.26x	1.63x to 13.40x	2.16x to 15.97x	2.89x to 17.23x

Table 3: Speedup against Out-of-order baselines on a 512x512 matrix with 10% to 90% sparsity.

	Buffer size: 8			**Buffer size: 128**		
Vector length	**2**	**4**	**8**	**2**	**4**	**8**
Scalar CPU	1.73x to 1.72x	2.18x to 2.13x	2.45x to 2.00x	1.95x to 1.15x	2.46x to 2.32x	3.88x to 3.14x
SIMD CPU	1.53x to 1.23x	1.37x to 1.42x	1.28x to 1.37x	1.77x to 1.39x	1.55x to 1.47x	2.02x to 1.88x
Dense CPU	0.98x to 8.56x	0.95x to 8.18x	0.90x to 7.48x	1.13x to 9.69x	1.07x to 8.94x	1.44x to 10.29x

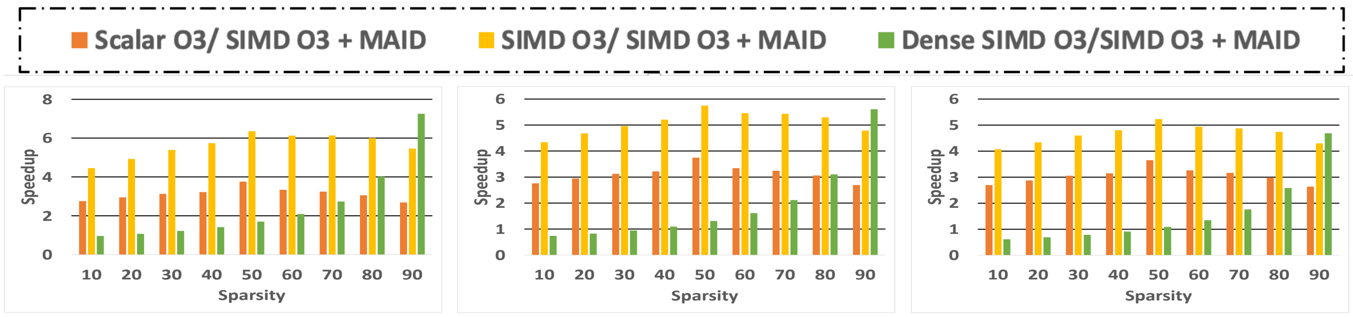


(a) Buffer size: 8, Vector length: 2 (b) Buffer size: 8, Vector length: 4 (c) Buffer size: 8, Vector length: 8



(d) Buffer size: 128, Vector length: 2 (e) Buffer size: 128, Vector length: 4 (f) Buffer size: 128, Vector length: 8

Figure 4: Performance analysis for 512x512 spMV on Out-of-order core: Scalar, SIMD-only sparse and SIMD-only dense baselines vs SIMD with MAID acceleration for various buffer sizes and vector lengths.



(a) Buffer size: 16, Vector length: 2 (b) Buffer size: 16, Vector length: 4 (c) Buffer size: 16, Vector length: 8

Figure 5: Performance analysis for 512x512 spMspV on Out-of-order core: Scalar, SIMD-only sparse and SIMD-only dense baselines vs SIMD with MAID acceleration for varying vector lengths and buffer size of 16.

5.3 *MAID* aiding Out-of-order processors on spMspV workload

In this section we evaluate the effectiveness of *MAID* when both the matrix and vector are sparse. The vector is represented using two arrays instead of a single dense array as in spMV: *V_cols* array to hold the positions of nonzeros within the vector while the *V_vals* array stores the corresponding nonzero values. For this set of experiments, we used the same sparsity levels for both matrix and vector. The primary core employed was an O3 SIMD CPU, as we had previously observed with spMV algorithm that even O3 scalar CPU can effectively hide latency for irregular memory fetching compared to a in-order SIMD CPU. The local memory of *MAID* now stores not only the vector values and the column indexes of a single row from the sparse matrix (as in spMV computations) but also the vector indexes and matrix values for that row. *MAID* performs index matching between the matrix column indexes and vector indexes to determine the presence of a corresponding nonzero value in the vector. *MAID* fetches matrix and vector nonzero values only if there is a match. We doubled the buffer size for spMspV (buffer size of 16 in Figure 5) compared to the spMV since *MAID* supplies matching nonzero matrix and vector values: the buffer is divided into two halves, with the first half storing aligned vector values and the second half storing the aligned matrix values. The primary core uses vector load instructions to obtain aligned matrix and vector values from the buffer and performs MAC operations. To optimize performance and minimize the delays in providing the primary core with data, we sometimes supply zero values for vector (when it becomes necessary to fetch additional data to assure index matching). This may cause some wasted zero computations, but this can be avoided by providing the CPU with masks to indicate the number of zeros that are included in the buffers. It should be noted that the zeros are appended after packing all the aligned values within the buffer size. *MAID* still aligns only one value at a time and the latency for aligning each value is dependent on the local memory access latencies and instruction latencies for index matching and accessing arrays. Figure 5 shows the performance improvements of *MAID* aided out-of-order SIMD CPU against Scalar O3 CPU, SIMD O3 CPU and Dense SIMD O3 CPU baselines at a buffer size of 16 and vector lengths of 2, 4 and 8 for varying sparsities. As can be seen in these figures, as vector length increases, all values supplied by *MAID* are effectively utilized by the CPU in a single iteration. Consequently, the overhead of identifying matching indexes becomes more substantial at lower sparsities when vector length increases. This leads to a lack of speedup against the dense baseline until the sparsity reaches 50% for a vector length of 8. However at lower vector lengths, *MAID* can maintain pace with the processor resulting in marginal gains at 20% for vector length of 2. SIMD+*MAID* consistently outperforms Scalar and SIMD CPU only baselines since memory-intensive pattern matching and irregular data gathering tasks are offloaded to *MAID* which works in parallel with the CPU. Scalar baseline performs better than SIMD CPU-only baseline because SIMD instructions are incapable of executing pattern matching on indexes, and they struggle with data irregularities and dependencies. With *MAID*, on an average, we observe speedups of almost 4x for all sparsities over SIMD CPU-only baseline and 3x over Scalar baseline. In both Scalar and SIMD CPU-only baselines, as sparsity increases, performance improves upto 50% sparsity and then there is a slight decline. This decline occurs at higher sparsities since there are fewer nonzero values in both matirx and vector: as stated above, we sometimes pad buffers with zero values which can lead to higher number of wasted zero computations. The zero computations can be avoided by providing masks to indicate the presence of zero values and the CPU can deactivate SIMD lanes with zero values (and we will explore this in our future work).

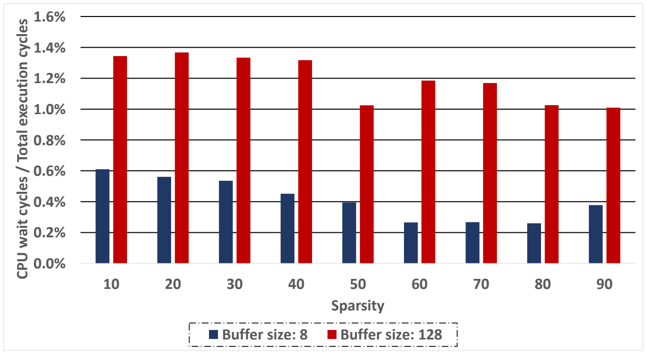


Figure 6: CPU Wait cycles in out-of-order SIMD+ *MAID* at vector length of 8 for spMV algorithm

As the vector length increases, wait cycles also increase, as all supplied data is rapidly consumed by the CPU. Since *MAID* performs more computations to match nonzero matrix and vector values, causing longer delays in filling and providing buffers to the CPU, we observe CPU wait cycles of above 7% compared to only 1% with spMV as shown in Figures 6 and 7. The use of zero padding actually reduced the wait cycles to some extent and the wait cycles will be even higher otherwise in spMspV.

In summary, even when both the matrix and the vector are sparse, *MAID* can be beneficial with an out-of-order CPUs using vector

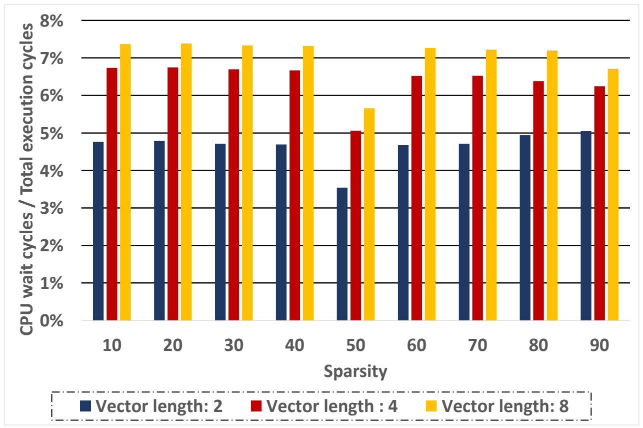


Figure 7: CPU Wait cycles in out-of-order SIMD+ *MAID* for spMspV algorithm at varying vector lengths and buffer size of 16.

instructions. Without *MAID* an out-of-order CPU faces difficulties in hiding latencies since it must assemble matching values for the SIMD instructions and the irregularities caused by the presence of sparsities with both the matrix and vector can lead to several memory accesses and cache misses. However *MAID* helps in mitigating the latency in fetching values by operating in parallel with CPU and supplying values as needed. CPU may be waiting longer for *MAID* to provide aligned data , but performance gains result from the overlapped execution of *MAID* with the CPU. SIMD+*MAID* can outperform Scalar, SIMD CPU-only and dense baselines (using out-of-order CPUs), particularly at higher sparsities. For lower sparsities with longer vector lengths, the process of matching indexes incurs a significant overhead in comparison to conducting a dense multiplication.

5.4 Memory utilization

Our goal is to offload the irregular memory computations to *MAID* and work in parallel with CPU. In spMV algorithm, each MAC operation involves a total of 3 cache accesses: one for fetching the column index, one for the associated vector value, and one for the matrix value instead of 2 load accesses typically seen in a dense matrix-dense vector algorithm. However, with *MAID* aligning vector values for the CPU, we effectively decrease the total cache accesses required by the CPU for a single MAC operation to 2; fetching only matrix and vector values, while *MAID* handles the irregular index matching. By providing these values as needed by the CPU, we manage to reduce CPU cache misses. In other words, the scalar CPU and SIMD CPU baselines experience higher cache misses compared to the SIMD+*MAID* configuration. Given the differences in the total number of cache accesses for different configurations, we recognize that the traditional cache miss rate is not a fair or accurate way to compare the effective use of caches. Instead we opted to employ cache misses per MAC operation as our metric for comparing cache utilization. Figures 8 and 9 illustrate the L1 Dcache and L2/LLC cache misses for out-of-order configurations using a vector length of 8 with 2 different buffer sizes of 8 and

128. It can be seen that changing buffer size or vector length does not impact LLC misses. This is attributed to the fact that all 512 vector values, along with a maximum of 512 column indexes, are already present in *MAID*'s local memory. As a result, there is no change in LLC misses in Figure 9, regardless of buffer size. As *MAID* operates in parallel and provides data to the CPU through buffers in L1 Dcache as needed, L1 misses are reduced as observed in Figures 8. With larger buffer sizes, more data is supplied by *MAID* even before the CPU needs it, leading to a marginal rise in cache misses in comparison to smaller buffer sizes. This situation may lead to unnecessary L1 cache evictions, subsequently contributing to higher cache misses. However, with *MAID* working alongside the SIMD CPU and is connected to the LLC, it can retrieve the column index and vector data directly from the LLC without storing them into the L1 cache. This leads to a decrease in the overall number of cache misses with our *MAID* alongside SIMD configuration for spMV algorithm. Similarly in spMspV algorithm, using *MAID* in conjunction with the SIMD core reduces the L1 Dcache misses on average by 7.2 times and L2 cache misses by 9 times for all sparsities reducing the number of cache misses per MAC as shown in Figures 10 and 11 respectively. The inclusion of zeros in the buffer during spMspV as described previously makes the number of cache misses per MAC similar for both spMV and spMspV algorithms (treating the sparse vector as if it is dense). The reduced cache access and misses when *MAID* is used suggests that the use of local memory with *MAID* (8KiB in our experiments) should not be viewed as additional hardware: it may be possible to use smaller L1 caches for the primary core and use the saved space for *MAID* local memory.

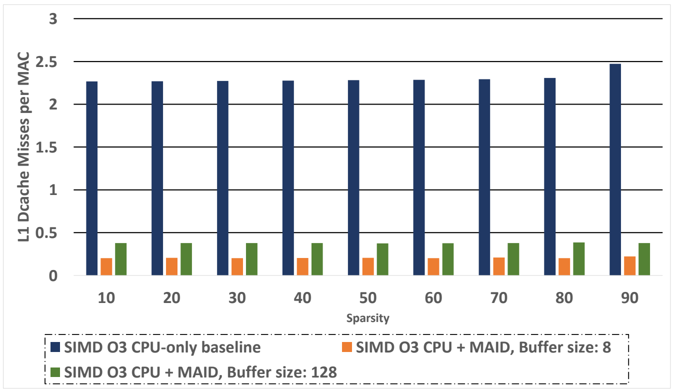


Figure 8: Number of L1 Dcache misses of Out-of-order SIMD CPU baseline over SIMD +*MAID* for spMV algorithm at vector length of 8 with buffer sizes 8 and 128.

6 RELATED WORK

Sparse accelerators. Much work has been done on compression for DNNs, but the unique challenges posed by quantized data and SIMD processing have not been adequately addressed. dCSR [33] proposes to address the issue of indexing bit-width by altering CSR format to store an index *delta* rather than the column index itself. This *delta* is calculated as the difference between the actual column index of a nonzero value and the average index for a matrix row.

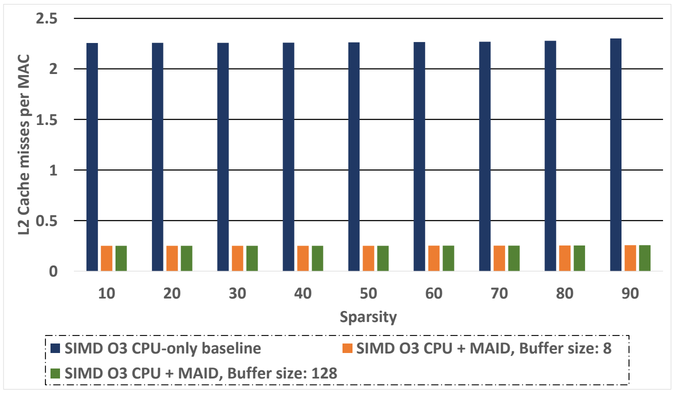


Figure 9: Number of LLC misses of Out-of-order SIMD CPU baseline over SIMD +*MAID* for spMV algorithm at vector length of 8 with buffer sizes 8 and 128.

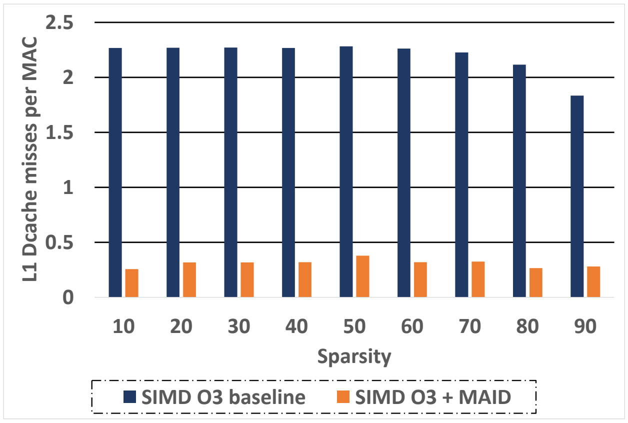


Figure 10: Number of L1 Dcache misses of Out-of-order SIMD CPU baseline over SIMD +*MAID* for spMspV algorithm at vector length of 8 with buffer size 16.

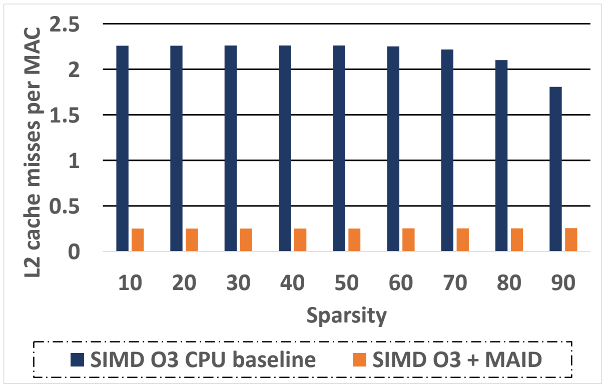


Figure 11: Number of LLC misses of Out-of-order SIMD CPU baseline over SIMD +*MAID* for spMspV algorithm at vector length of 8 with buffer size 16.

This format tackles the same issue of compressing the bit width of sparse column indexes, but is much more complicated to implement in a streaming run time. Our solution is simpler and more flexible because 1) the compression algorithm does not require advance knowledge of the matrix layout as in dCSR and 2) our storage format is agnostic to the initial shape of the dense matrix.

Some [7, 23] propose Processing-in-Memory approach to rearrange sparse data either to align nonzero values or expand sparse date to dense data. While our approach is somewhat similar to these, we overlap the data alignment with computations and we propose our near data processing along cache memories. Moreover, we rely on programmable cores for our data alignment which allows the alignment to be based on different sparse data formats and access patterns (e.g., row-wise or column-wise). There were some preliminary results presented for near data processing for embedded and micro controller systems based on RISC V cores [3, 34, 37].

Prefetchers. There are several studies on data prefetching into on-chip cache to hide the memory latency problem. Most of the early works [16, 19–21] are sequential stride-based prefetchers. Some more recent prefetchers can even be programmed or learn to handle irregular accesses [39], similar to those those used by Apple M1 systems. However, in most cases prefetchers are unaware of program specific structure bounds (or array bounds) and this can be exploited by security attacks as reported in [35]. Our *MAID* is programmed not only to understand access patterns but also with memory bounds for different structures, limiting the ability of out of bounds attacks.

Helper Threads. While there have been many prior studies in terms of *decoupling* or *off-loading* memory access operation (consider an early decoupling work reported in [31]), *MAID* is a flexible hardware which can be programmed to process application specific metadata processing. Helper threads (particularly software threads) have been used to aid primary threads with some operations (for example see [22]). Such software techniques may not lead to performance gains if the threads are scheduled on different cores requiring cache coherency related overheads. We use separate hardware unit specifically for indexing operations, placed near memory and hence eliminate cache coherency issues and compiler optimization that leads to performance loss in software threads.

Accelerators for Machine Learning. Interest in DNN based accelerators have seen a rise in recent years, leading to many specialized hardware accelerators, too many for us to include here. Many of these specialized accelerators based on either dataflow or tensor/systolic arrays that lack flexibility or reconfigurability [11, 18, 24, 25, 27, 28, 42]. These accelerators either rely on very specialized sparse data representations or implement specific DNN algorithms. Our *MAID* only aids in memory-side operations and does not perform actual computations. In this contribution we focused on quantized data used in TinyML applications. Our *MAID* is programmable and can be used with different compression formats and for different DNN algorithms.

Several works focus on accelerating sparse matrix-dense vector multiplication (*spMV*) operations [5, 10, 17, 29, 38], We only proposed to aid in index computations for any sparse data based algorithm instead of accelerating the actual computations. While [7] and [2] are somewhat similar to our work, they expand sparse data into dense data so that the primary cores can rely on simple algorithms; our *MAID* only provided required or matching nonzero values needed for the computation, still keeping the primary core computations simple.

Processing in Memory Simulation Tools. There are very few usable tools for prototyping processing in memory or near memory designs [14, 36]. There are FPGA based PIM prototyping tools [26]. We extended [36] since the tool is based on widely used full system simulator Gem5 [8]. Our framework permits programmable PIMs or custom hardware PIMs.

7 CONCLUSION

In this work, we introduced *MAID*, a memory-side programmable accelerator designed to operate in conjunction with a vectorized in-order or out-of-order CPU. The primary aim is to expedite computations involving both sparse matrix-dense vector and sparse matrix-sparse vector algorithms using CSR format and can be extended to other algorithms and compression formats. We offloaded the irregular data gathering task to our accelerator, which functions in parallel with the processor, and aligns them for the CPU. This alignment enhances SIMD compatibility, thereby reducing cache misses and enhancing performance. This approach is particularly advantageous when applied to the spMspV algorithm, given that both arrays exhibit sparsity and involves index matching before fetching matrix and vector values. We evaluated the performance of our accelerator in the CPU-memory system by modifying the Gem5 extension to enable parallel execution of both CPU and *MAID* while communicating through CPU's L1 data cache. In the future, we would like to explore multi-ported local memory for *MAID* to access multiple column indexes and vector values so that *MAID* can meet the demands of wider SIMD registers or even GPU threads.

Acknowledgements This research is supported in part by a SRC grant AIHW Task 2943 and a NSF grant #1828105. The authors acknowledge Paolo D'Alberto (AMD), Abhishek Kumar Jain (AMD), Brandon Potter (AMD), Marko Scrbak (AMD), Mahzabeen Islam (AMD), Mike Ignatowski (AMD), Lizy K. John (UT Austin) and Fernando Mosquera (UNT) for their valuable suggestions.

REFERENCES

[1] 2018. The API reference guide for cuSPARSE, the CUDA sparse matrixlibrary.(v8.0 ed.). (2018).

[2] Shashank Adavally, Nagendra Gulur, Krishna Kavi, Alex Weaver, Pranoy Dutta, and Benjamin Wang. 2020. ExPress: Simultaneously Achieving Storage, Execution and Energy Efficiencies in Moderately Sparse Matrix Computations. In *The International Symposium on Memory Systems*. 46–60.

[3] S. Adavally, A. Weaver, P. Vasireddy, K. Kavi, G. Mehta, and N. Gulur. 2022. Heterogeneous architecture for sparse data processing. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE Computer Society, Los Alamitos, CA, USA, 6–15. https://doi.org/10.1109/IPDPSW55747.2022.00012

[4] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi. 2015. A scalable processing-in-memory accelerator for parallel graph processing. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. 105–117. https://doi.org/10.1145/2749469.2750386

[5] Ariful Azad and Aydin Buluç. 2017. A Work-Efficient Parallel Sparse Matrix-Sparse Vector Multiplication Algorithm. In *2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2017, Orlando, FL, USA, May 29 - June 2, 2017*. IEEE Computer Society, 688–697. https://doi.org/10.1109/IPDPS.2017.76

[6] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. 1997. Efficient Management of Parallelism in Object Oriented Numerical Software Libraries. In *Modern Software Tools in Scientific Computing*, E. Arge, A. M. Bruaset, and H. P. Langtangen (Eds.). Birkhäuser Press, 163–202.

[7] Adrián Barredo, Jonathan C Beard, and Miquel Moretó. 2019. Poster: Spidre: Accelerating sparse memory access patterns. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 483–484.

[8] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH computer architecture news* 39, 2 (2011), 1–7.

[9] Aydin Buluç, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson. 2009. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *SPAA 2009: Proceedings of the 21st Annual ACM Symposium on Parallelism in Algorithms and Architectures, Calgary, Alberta, Canada, August 11-13, 2009*, Friedhelm Meyer auf der Heide and Michael A. Bender (Eds.). ACM, 233–244. https://doi.org/10.1145/1583991.1584053

[10] Aydin Buluç and John R. Gilbert. 2012. Parallel Sparse Matrix-Matrix Multiplication and Indexing: Implementation and Experiments. *SIAM J. Scientific Computing* 34 (2012).

[11] Pengcheng Dai, Jianlei Yang, Xucheng Ye, Xingzhou Cheng, Junyu Luo, Linghao Song, Yiran Chen, and Weisheng Zhao. 2020. SparseTrain: Exploiting Dataflow Sparsity for Efficient Convolutional Neural Networks Training. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1109/DAC18072.2020.9218710

[12] Timothy A. Davis. 2019. Algorithm 1000: SuiteSparse:GraphBLAS: Graph Algorithms in the Language of Sparse Linear Algebra. *ACM Trans. Math. Softw.* 45, 4, Article 44 (dec 2019), 25 pages. https://doi.org/10.1145/3322125

[13] Jack Doweck, Wen-Fu Kao, Allen Kuan-yu Lu, Julius Mandelblat, Anirudha Rahatekar, Lihu Rappoport, Efraim Rotem, Ahmad Yasin, and Adi Yoaz. 2017. Inside 6th-Generation Intel Core: New Microarchitecture Code-Named Skylake. *IEEE Micro* 37, 2 (Mar 2017), 52–62. https://doi.org/10.1109/MM.2017.38

[14] Bruno E. Forlin, Paulo C. Santos, Augusto E. Becker, Marco A.Z. Alves, and Luigi Carro. 2022. Sim2PIM: A complete simulation framework for Processing-in-Memory. *Journal of Systems Architecture* 128 (2022), 102528. https://doi.org/10.1016/j.sysarc.2022.102528

[15] Mark Evers, Leslie Barnes, and Mike Clark. 2022. The AMD Next-Generation "Zen 3" Core. *IEEE Micro* 42, 3 (2022), 7–12. https://doi.org/10.1109/MM.2022.3152788

[16] John W. C. Fu, Janak H. Patel, and Bob L. Janssens. 1992. Stride Directed Prefetching in Scalar Processors. In *Proceedings of the 25th Annual International Symposium on Microarchitecture* (Portland, Oregon, USA) *(MICRO 25)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 102–110. http://dl.acm.org/citation.cfm?id=144953.145006

[17] Joseph L. Greathouse and Mayank Daga. 2014. Efficient Sparse Matrix-vector Multiplication on GPUs Using the CSR Storage Format. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (New Orleans, Louisana) *(SC '14)*. IEEE Press, Piscataway, NJ, USA, 769–780. https://doi.org/10.1109/SC.2014.68

[18] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: Efficient Inference Engine on Compressed Deep Neural Network. *CoRR* abs/1602.01528 (2016). arXiv:1602.01528 http://arxiv.org/abs/1602.01528

[19] Mahzabeen Islam, Soumik Banerjee, Mitesh Meswani, and Krishna Kavi. 2016. Prefetching As a Potentially Effective Technique for Hybrid Memory Optimization. In *Proceedings of the Second International Symposium on Memory Systems* (Alexandria, VA, USA) *(MEMSYS '16)*. ACM, New York, NY, USA, 220–231. https://doi.org/10.1145/2989081.2989129

[20] D. Joseph and D. Grunwald. 1999. Prefetching using Markov predictors. *IEEE Trans. Comput.* 48, 2 (Feb 1999), 121–133. https://doi.org/10.1109/12.752653

[21] N. P. Jouppi. 1990. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *[1990] Proceedings. The 17th Annual International Symposium on Computer Architecture*. 364–373. https://doi.org/10.1109/ISCA.1990.134547

[22] Jaejin Lee, Changhee Jung, Daeseob Lim, and Yan Solihin. 2008. Prefetching with helper threads for loosely coupled multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems* 20, 9 (2008), 1309–1324.

[23] Scott Lloyd and Maya Gokhale. 2015. In-memory data rearrangement for irregular, data-intensive computing. *Computer* 48, 8 (2015), 18–25.

[24] Mostafa Mahmoud, Isak Edo, Ali Hadi Zadeh, Omar Mohamed Awad, Gennady Pekhimenko, Jorge Albericio, and Andreas Moshovos. 2020. TensorDash: Exploiting Sparsity to Accelerate Deep Neural Network Training. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 781–795. https://doi.org/10.1109/MICRO50266.2020.00069

[25] T. Moreau, T. chen, L. Vega, J. Roesch, E. yan, L. Zheng, J. Fromm, Z. Jiang, L. Ceze, C. Guestrin, and A. Krishnamurthy. 2019. A hardware-software blueprint for flexible deep learning specialization. *IEEE Micro* (Sept/Oct 2019). https://doi.org/10.1109/MM.2019.2928962

[26] Ataberk Olgun, Juan Gómez Luna, Konstantinos Kanellopoulos, Behzad Salami, Hasan Hassan, Oguz Ergin, and Onur Mutlu. 2022. PiDRAM: A Holistic End-to-End FPGA-Based Framework for Processing-in-DRAM. *ACM Trans. Archit. Code Optim.* 20, 1, Article 8 (nov 2022), 31 pages. https://doi.org/10.1145/3563697

[27] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. 2017. SCNN: An Accelerator for Compressed-Sparse Convolutional Neural Networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA '17)*. Association for Computing Machinery, New York, NY, USA, 27–40. https://doi.org/10.1145/3079856.3080254

[28] E. Qin, A. Samajdar, H. Kwon, S. Srinivasan, D. Das, B. Kaul, and T. Krishna. 2020. SIGMA: A sparse and irregular GEMM accelerator with flexible interconnects for DNN training. (Feb 2020).

[29] Fazle Sadi, Joe Sweeney, Tze Meng Low, James C. Hoe, Larry T. Pileggi, and Franz Franchetti. 2019. Efficient SpMV Operation for Large and Highly Sparse Matrices using Scalable Multi-way Merge Parallelization. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*. ACM, 347–358. https://doi.org/10.1145/3352460.3358330

[30] Charles Shelor and Krishna M. Kavi. 2017. Dataflow based Near Data Computing Achieves Excellent Energy Efficiency. In *HEART*.

[31] James E Smith. 1982. Decoupled access/execute computer architectures. *ACM SIGARCH Computer Architecture News* 10, 3 (1982), 112–119.

[32] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanaël Prémillieu, Alastair Reid, Alejandro Rico, and Paul Walker. 2018. The ARM Scalable Vector Extension. *CoRR* abs/1803.06185 (2018). arXiv:1803.06185 http://arxiv.org/abs/1803.06185

[33] Elias Trommer, Bernd Waschneck, and Akash Kumar. 2021. dCSR: A Memory-Efficient Sparse Matrix Representation for Parallel Neural Network Inference. arXiv:2111.12345 [cs.DS]

[34] Pranathi Vasireddy, Krishna Kavi, and Gayatri Mehta. 2022. Sparse-T: Hardware accelerator thread for unstructured sparse data processing. *International Conference on Computer-Aided Design (ICCAD '22)* (2022). https://doi.org/10.1145/3508352.3549441

[35] J. Sanchez Vicarte, M. Flanders, R. Paccagnella, G. Garrett-Grossman, A. Morrison, C. Fletcher, and D. Kohlbrenner. 2022. Augury: Using data memory-dependent prefetchers to leak data at rest. In *2022 2022 IEEE Symposium on Security and Privacy (SP) (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 1518–1518. https://doi.org/10.1109/SP46214.2022.00089

[36] Joao Vieira, Nuno Roma, Gabriel Falcao, and Pedro Tomás. 2022. gem5-ndp: Near-Data Processing Architecture Simulation From Low Level Caches to DRAM. In *2022 IEEE 34th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 41–50.

[37] Alex Weaver, Krishna Kavi, Pranathi Vasireddy, and Gayatri Mehta. 2022. Memory-Side Acceleration and Sparse Compression for Quantized Packed Convolutions. In *2022 IEEE 34th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. 81–90. https://doi.org/10.1109/SBAC-PAD55451.2022.00019

[38] Leonid Yavits and Ran Ginosar. 2017. Sparse Matrix Multiplication on CAM Based Accelerator. *CoRR* abs/1705.09937 (2017). arXiv:1705.09937 http://arxiv.org/abs/1705.09937

[39] Xiangyao Yu, Christopher Hughes, Nadathur Satish, and Srinivas Devadas. 2015. IMP: Indirect memory prefetcher. In *Proceedings of the 48th International Symposium on Microarchitecture*. IEEE, 178–190.

[40] Dong Ping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joseph Greathouse, Mitesh Meswani, Mark Nutter, and Mike Ignatowski. 2013. A New Perspective on Processing-in-memory Architecture Design. In *Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness* (Seattle, Washington) *(MSPC '13)*. ACM, New York, NY, USA, Article 7, 3 pages. https://doi.org/10.1145/2492408.2492418

[41] Dong Ping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joseph L. Greathouse, Lifan Xu, and Mike Ignatowski. 2014. TOP-PIM: throughput-oriented programmable processing in memory. In *HPDC*.

[42] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. 2016. Cambricon-X: An accelerator for sparse neural networks. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12. https://doi.org/10.1109/MICRO.2016.7783723