MILES: Instruction-Level Simulation of Dense and Sparse Matrix Engines in CPUs with A Unified Performance Model

Xiaoyu Hao, Sen Zhang, Junshi Chen, and Hong An

***Abstract*—Matrix extensions (MEs) are increasingly adopted in modern CPUs to accelerate dense and sparse matrix multiplication. However, evaluating CPU MEs is challenging because matrix instructions involve both arithmetic operations and memory access, and their performance depends on the interaction among the CPU, matrix accelerator (MA), and memory hierarchy. Existing NPU simulators either lack instruction-level simulation, provide only limited CPU modeling, or require significant effort to customize accelerator behavior.**

This paper presents MILES, an instruction-level simulation framework for dense and sparse matrix engines in CPUs. MILES is a trace-driven and cycle-level simulator. To model MAs efficiently, MILES introduces a memory-centric performance model, MCPM, which is a unified performance model for both dense and sparse MAs. By detailed memory simulation and a simplified model for arithmetic operations, MCPM achieves high accuracy while maintaining high simulation speed. To support irregular sparse accelerators, MILES further proposes MCAR, a memory-centric accelerator representation that describes accelerator behavior through external descriptions.

Experimental results show that MILES achieves geometric mean errors of 13.9% and 7.38% for OS and WS systolic arrays, respectively, compared with BOOM+Gemmini RTL simulation, while providing a $432\times$ simulation speedup. For sparse accelerators based on Gustavson, inner-product, and outer-product dataflows, MILES achieves errors of 9.7%, 3.7%, and 4.5%, respectively, compared with sstStonne, while obtaining a $20\times$ simulation speedup. These results demonstrate that MILES provides an accurate, efficient, and flexible framework for CPU–accelerator co-simulation of CPU MEs.

***Index Terms*—CPU, Matrix Extension, Accelerator, Simulator, Matrix Multiplication.**

I. Introduction

In recent years, with the rapid development of artificial intelligence and scientific computing, matrix extensions (MEs) are becoming an important part of mainstream instruction set architectures (ISAs) [1]–[6]. Compared with scalar and vector instructions, matrix instructions involve both arithmetic operations and memory access at a coarser granularity. Their performance depends not only on the matrix accelerator itself but also on the CPU and the memory system. Therefore, in the early design stage, it is crucial to use architectural simulators for design space exploration and performance evaluation. However, building a simulator for MEs faces three challenges.

Xiaoyu Hao, Sen Zhang, Junshi Chen, and Hong An are with the School of Computer Science and Technology, University of Secience and Technology of China, Hefei, China. (Email: haoxiaoyu@mail.ustc.edu.cn)

First, modeling of sparsity. The primary goal of ME is to accelerate General Matrix Multiplication (GEMM). Meanwhile, leveraging sparsity to further improve the computational efficiency of matrix accelerators (MAs) has been widely studied [7]–[10]. However, sparse MAs exhibit indirect memory access behavior that depends on input data, which necessitates a different modeling approach from that used for GEMM accelerators. Second, co-modeling of CPU, MA, and the memory system. The execution of matrix instructions can be blocked by general-purpose instructions due to data or structural hazards, which is known as the configuration wall [11]. As MAs have various memory behaviours and share memory hierarchy levels, such as caches and DRAM with CPUs, modeling of memory system is also required. Third, the conventional instruction-level cycle-by-cycle simulation approach, while achieving high accuracy, suffers from slow simulation speed. Additionally, matrix instructions involve both arithmetic operations and memory access, making it difficult to simplify them into a fixed-latency model as is commonly used for scalar instructions, further increasing the time overhead of detailed simulation.

Recently, many Neural Processing Unit (NPU) simulators have been proposed, but they still exhibit the following three shortcomings when addressing the above challenges: (1) Most works focus only on evaluating the performance of the MA itself, lacking the capability for co-modeling with the CPU. Currently, only PytorchSim [12] and SMAUG [13] provide CPU simulation functionality. (2) There is a lack of instruction-level simulation (ILS) methods. Due to the absence of compiler support, some works [14]–[16] use custom input formats and implement cycle-level simulators driven by computation and memory access events, but they cannot represent computation in the form of instructions, leading to significant errors when simulating CPU matrix instructions. Although PytorchSim supports matrix instructions in Gem5 [17] and Spike [18], it only collects computation latency from them as input to its performance model, rather than performing instruction-level co-simulation of the CPU and the accelerator. SMAUG triggers accelerator execution via system calls rather than instructions. (3) There is a lack of flexibility in defining accelerator behavior. Most existing approaches hard-code the accelerator behavior inside the simulator, and introducing a new architecture or modifying existing behavior often requires additional development effort. Only a few simulators [19], [20], including SMAUG, support customizing accelerator behavior through external descriptions, i.e., C kernels.

To address these limitations, we propose MILES [1], an ILS framework for dense and sparse matrix engines in CPUs. MILES performs trace-driven, cycle-level co-simulation of the CPU, MA, and memory hierarchy. It uses LLVM Intermediate Representation (IR) as the instruction representation, allowing matrix instructions and general-purpose instructions to be executed in a unified instruction stream. When the CPU model encounters a matrix instruction, the instruction is dispatched to a coprocessor model, which invokes a memory request generator (MRG) to produce memory requests according to the target MA behavior. MRG is governed by a memory-centric performance model (MCPM), which provides a unified performance model for both GEMM and sparse-sparse matrix multiplication (SpMSpM).

The key idea of MILES is to model matrix engines from the perspective of memory behavior rather than fully simulating every arithmetic operation inside the accelerator. MILES abstracts the execution of different matrix engines into three common stages: stationary (STA), streaming (STR), and writeback (WB). For GEMM accelerators, MILES directly generates regular memory requests from instruction operands, such as DRAM addresses, scratchpad addresses, and matrix dimensions. For sparse matrix engines, whose memory accesses depend on input data and indirect indices, MILES introduces a memory-centric accelerator representation (MCAR) that describes accelerator behavior using C kernels, annotated address streams, and synchronization primitives. This design enables MILES to support both GEMM engines based on systolic array (SA) and representative SpMSpM engines within the same simulation framework.

This paper makes the following contributions:

- We propose a unified performance model, MCPM, that captures both GEMM and SpMSpM accelerators through common execution stages and fine-grained memory behavior, enabling accurate and efficient cycle-level simulation by simplifying the model of arithmetic operations.
- We introduce MCAR that allows users to define new MA behaviors using C kernels, annotated address streams, and synchronization primitives.
- We present MILES, an instruction-level co-simulation framework that integrates CPU, MA, and memory-system modeling. MILES supports both regular GEMM accelerators and irregular SpMSpM accelerators whose memory accesses depend on input data.
- We demonstrate MILES on OS/WS SAs and representative SpMSpM accelerators, achieving low simulation error and significant speedup over RTL-level and a detailed accelerator simulator sstSTONNE [21].

II. Background and Motivation

A. The Importance of CPU–Accelerator Co-design

Matrix extensions have been widely adopted in general-purpose ISAs. These extensions follow a similar design philosophy: they introduce custom hardware matrix registers, such as AMX Tile registers [1] and SME ZA two-dimensional registers [3], to provide the data-level parallelism required by GEMM. At the instruction level, such extensions can directly address source and destination operands using register numbers, as exemplified by the AMX `TDPBF16PS` instruction and the SME `FMOPA` instruction.

In addition to register-based matrix extensions, another class of implementations does not rely on architectural matrix registers, such as Gemmini [22]. Gemmini is integrated with the CPU as a coprocessor and uses scratchpad memory (SPM) as its on-chip storage. In such designs, matrix computation instructions must explicitly encode parameters such as DRAM/SPM addresses, data sizes, offsets, et al. This leads to two issues. First, due to the limited instruction width, a single matrix operation often needs to be decomposed into multiple instructions. Furthermore, constrained by the matrix size that a single matrix instruction can process, large target matrices must be handled by repeatedly invoking matrix instructions inside loops, thereby introducing additional loop-control overhead. Second, the corresponding addresses and parameters must be generated by additional scalar instructions, which may block the normal issue and execution of matrix instructions. This problem has recently been referred to as the configuration wall [11]. Therefore, for this class of designs, it is important to introduce ILS approaches during instruction design and evaluation, so as to accurately capture the impact of the configuration wall.

Figure 1 shows the cycle breakdown of GEMM when Gemmini is integrated with Rocket [23] and BOOM [24], respectively. Gemmini uses the default 16×16 SA configuration. The experiment is based on RTL simulation, and the execution time of Gemmini and the CPU is measured separately using performance counters.

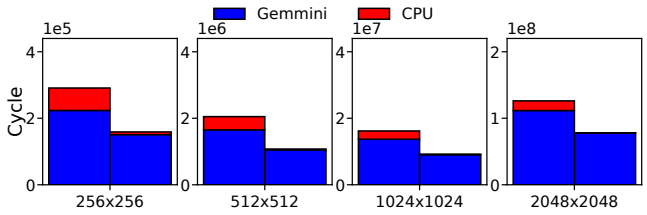


Fig. 1: Execution cycles of Gemmini integrated with Rocket and BOOM

The results show that using BOOM not only reduces the execution time of the CPU itself, but also reduces the computation time of Gemmini. This indicates that a more powerful CPU can effectively shorten the time during which matrix instructions are blocked. Taking a 512×512 matrix as an example, the CPU and Gemmini execute approximately 1.47 million instructions in total, while Gemmini executes only about 32 thousand GEMM instructions. Even when data movement instructions, such as `mvin` and `mvout`, are included, matrix instructions still account for only a small fraction of the total instruction count. As we discussed before, this large number of scalar instructions thereby delays the dispatch of matrix instructions to Gemmini. For example, matrix instructions rely on division (`div`) and remainder (`rem`)

[1]MILES is named after the combinations of words: **M**atrix, **I**nstruction-**L**evel, **E**ngine, and **S**imulator

TABLE I: Limitations of current NPU simulators

	Instruction-Level Co-Simultion	Accurate CPU Model	Sparsity	Customizable Behavior
PytorchSim [12]	✗	△	✓	✗
sstStonne [21]	✗	✗	✓	✗
SCALESimV3 [14]	✗	✗	✓	✗
mNPUSim [15]	✗	✗	✗	✗
ONNXim [16]	✗	✗	✗	✗
SMAUG [13]	✗	✓	✗	✓
MILES	✓	✓	✓	✓

△: Partial Support

instructions to compute DRAM or SPM addresses. These instructions have long latencies, for instance, integer division on BOOM requires 65 cycles and cannot be fully pipelined, which severely limits the configuration bandwidth of the accelerator [11]. Therefore, CPU–accelerator co-modeling is crucial for improving overall system performance.

B. Limitations of Existing NPU Simulators

Although NPU simulators have been extensively studied, they still have several limitations when evaluating CPU MEs, as shown in Table I. As discussed previously, ILS of the CPU and accelerator is essential for ME design. However, many NPU simulators, including ONNXim [16], mNPUSim [15], SCALE-Sim [14], [25], and sstStonne [21], lack CPU simulation capability, making them unsuitable for evaluating MEs. Since modeling the performance of out-of-order (OOO) CPU cores is highly complex, these works cannot achieve accurate simulation without proper instruction representation and an OOO core performance model.

Currently, only PytorchSim [12] and SMAUG [13] provide CPU simulation capability based on Gem5 [17]. Although PytorchSim can execute matrix instructions using Gem5, its performance model only runs one loop iteration on Gem5 to obtain computational latency as input to the NPU performance model, rather than performing co-simulations. In addition, PytorchSim's RISC-V instruction extension flow relies on multiple toolchains, including Gem5 [17], Spike [18], and MLIR [26], making it costly to extend new instructions. SMAUG provides the ability to co-simulate CPU and accelerator, but it launches accelerators through a device control interface, i.e., `ioctl`. This process involves address translation and differs significantly from directly controlling accelerators through instructions. Meanwhile, although SMAUG supports SAs, its array size cannot exceed 8×8, and it does not support sparsity modeling.

Among the above works, only SMAUG supports customizing accelerator behavior with external descriptions. Simulators such as Gem5-Aladdin [19] and Gem5-SALAM [20] require users to fully specify both the accelerator's computation and memory-access behavior, and simulate every computation and memory operation in detail. In contrast, the MCAR method proposed in this work requires only that users define the key memory-access behavior, which is then modeled by MCPM. This design significantly simplifies computation representation and reduces simulation overhead.

C. Toward a Unified Execution Abstraction for Matrix Accelerator

The key to matrix multiplication accelerator design is data reuse. Since matrix multiplication involves memory accesses to two input matrices and one output matrix, directly accessing memory for every computation would make memory bandwidth a major performance bottleneck. Therefore, designing appropriate dataflows to reuse data inside the hardware is critical for improving accelerator performance. For dense matrix multiplication (GEMM), SAs are widely used to build matrix accelerators, such as Google TPU [27] and Gemmini [22]. Depending on which data are reused, SAs commonly adopt representative dataflows such as output stationary (OS) and weight stationary (WS). For sparse matrix multiplication (SpMSpM), since the input matrices are stored in compressed formats, the computation must also handle irregular memory accesses caused by nonzero indices. Existing SpMSpM accelerators commonly adopt representative dataflows such as inner product (IP) [7], [8], outer product (OP) [28], [29], and Gustavson-based dataflow [9], [30].

Although SpMSpM accelerators differ in sparse formats and dataflows, their execution can be commonly interpreted from a vector-level perspective. Figure 2 illustrates this abstraction. The input matrices are stored in compressed formats, such as CSR or CSC. During execution, the accelerator first selects a stationary vector, like a_1, from compressed matrix A, which is reused across multiple computations. This vector can often be obtained as a contiguous segment from the value array according to the corresponding pointer and index arrays. The streaming stage then gathers related data vectors from compressed matrix B. These streaming vectors, denoted as b_1 to b_N, are fed into the accelerator over time. Unlike dense GEMM accelerators, the elements of a streaming vector in SpMSpM may be non-contiguous in memory because they are selected through indirect addressing, which is input-dependent and can introduce irregular memory accesses. When streaming vectors are successively injected, the accelerator typically performs computation using a spatial reduction datapath, such as a multiplier-and-adder tree, to generate partial sums. Dense GEMM accelerators, such as SAs, also exhibit similar vector-level behavior, but their memory access patterns are more regular. In WS SAs, weight vectors are preloaded and kept stationary inside the array, while input vectors are continuously streamed to trigger computation. In OS SAs, input and weight vectors are streamed into the array, while partial sums are kept and accumulated inside the array before being written back. These behaviors can also be interpreted under the same vector-level abstraction.

Based on these observations, the execution of representative matrix accelerators can be broadly interpreted as three stages: stationary (STA), streaming (STR), and writeback (WB). The STA stage loads data that will be reused during computation, the STR stage continuously feeds input data at the vector granularity, and the WB stage drains results or partial sums to the memory system. This common execution pattern motivates a unified performance model that can support both regular GEMM accelerators and irregular SpMSpM accelerators.

TABLE II: Matrix accelerators modeled by MILES

Accelerator	On-Chip Memory	Stationary	Streaming
WS SA	SPM	Matrix B is preloaded row-wise	Matrix A is streamed row-wise
OS SA	SPM	–	Matrix A and matrix B are streamed column-wise and row-wise, respectively
Gust	Cache	Nonzero elements of one sparse row of A: $[A_{i,k_1}, A_{i,k_2}, \ldots, A_{i,k_m}]$, where k_r is the column index of a nonzero element in the i-th row of A	Nonzero elements in rows of B selected by the column indices k_r from the stationary vector form streaming input vectors: $\mathbf{b}^{(t)} = [B_{k_r,j_{r,t}}]_{r\in\mathcal{R}_t}$. Here, $j_{r,t}$ is the column index of the t-th nonzero element in the k_r-th row of B, $\mathcal{R}_t = \{r \mid t \le n_r\}$ is the set of row indices that still contain valid nonzero elements in the t-th streaming vector, and n_r is the number of nonzero elements in the k_r-th row of B
Sigma	Cache	Nonzero elements of one or multiple sparse rows of A	Columns of B are streamed one by one, where the nonzero elements are indexed by the nonzero columns of all stationary rows in A
OP	Cache	Nonzero elements of one or multiple sparse columns of A with column indices $k_1, \ldots, k_q$	The input vector is: $\mathbf{b}^{(t)} = [b_1^{(t)}\mathbf{1}_{m_1}; \ldots; b_q^{(t)}\mathbf{1}_{m_q}]$, where m_r is the number of nonzero elements in the k_r-th column, $b_r^{(t)} = B_{k_r,j_{r,t}}$ is the t-th nonzero element taken from the k_r-th row of B, $j_{r,t}$ is its column index, and $\mathbf{1}_m$ denotes an all-ones vector of length m

In the Gust and OP accelerators, the superscript (t) denotes the t-th streaming input vector.

In this work, we focus on three representative SpMSpM accelerators, including the Gustavson-based Flexagon accelerator (Gust), the outer-product-based Flexagon accelerator (OP), and the inner-product-based Sigma accelerator, as well as two GEMM accelerators, including WS and OS SAs. Table II summarizes their on-chip memory structures and their vector-level STA and STR behaviors. Although these accelerators differ in dataflow and memory access regularity, their execution can be described using the same abstraction: STA defines the data vectors reused by the accelerator, while STR defines the vectors continuously supplied to trigger computation. However, the WB behaviors vary among accelerators. For WS SAs, Gust, and Sigma, partial sums can be written back during the STR stage. In contrast, OS SAs accumulate partial sums inside the array and write back the final outputs after STR. OP first generates partial sums through outer-product computation, and then merges these partial sums before being written back to the memory system.

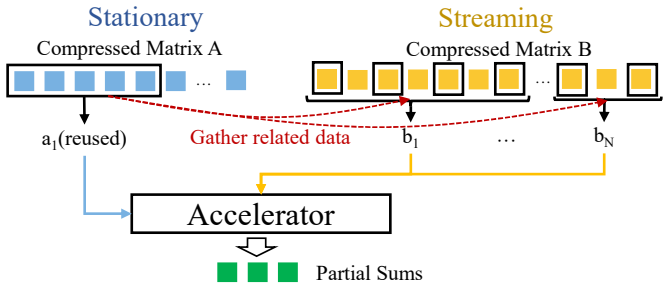


Fig. 2: Vector-level behavior of representative SpMSpM accelerators

III. MILES Overview

MILES is a performance modeling framework for CPU matrix extensions. Built upon CISim, it integrates a coprocessor model to simulate matrix instructions. MILES adopts a trace-driven, cycle-level simulation methodology and uses LLVM IR as its ISA. It currently supports performance models for various GEMM and SpMSpM accelerators, including OS and WS SAs, Sigma, OP, and Gust accelerators. MILES first runs the workload using the CPU model, and when a matrix instruction is encountered, it is dispatched to the coprocessor model for simulation, and the coprocessor notifies the CPU side after the instruction completes.

To support flexible customization of accelerator behavior, MILES provides a memory-centric accelerator representation method, namely MCAR. As shown in Fig. 3, the execution flow of a custom matrix instruction based on MCAR consists of three stages. First, in the user-defined C kernel, the primitives provided by MCAR are used to describe the core memory access behavior of the accelerator, mark different address streams, and specify the corresponding matrix instruction through a configuration file. Second, MILES uses the LLVM toolchain to instrument the program according to the MCAR annotations. During execution, it collects the memory addresses accessed by the binary, stores them into separate files based on their address stream labels, and records the execution order of these address streams. Third, the simulator replays these addresses based on the MCPM, thereby modeling and evaluating the accelerator behavior.

IV. Memory-Centric Performance Model

As discussed in Section II-C, representative GEMM and SpMSpM accelerators adopt different dataflows, storage formats, and on-chip memory structures. Despite these differences, their execution can be interpreted under a common

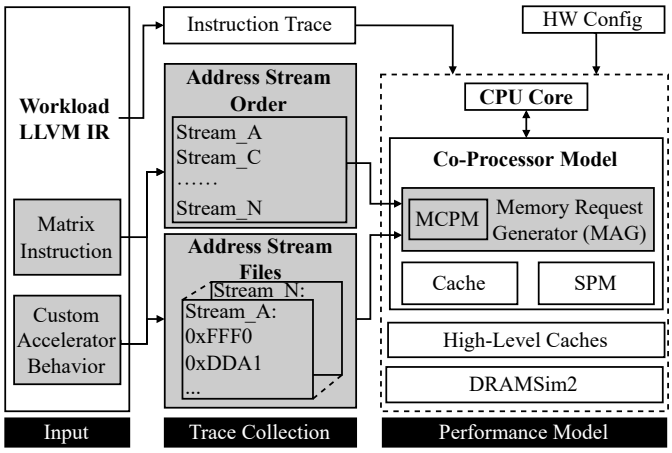


Fig. 3: The workflow of MILES

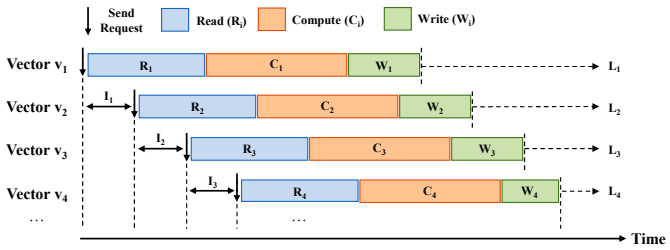


Fig. 4: MCPM for matrix accelerator

vector-level abstraction. To build an accurate and rapid cycle-level performance model, it is necessary to determine when each vector-level memory request is issued, when the requested data becomes available, how the data participates in computation, and when the generated results or partial sums are written back. Therefore, this work proposes a Memory-Centric Performance Model (MCPM) to support cycle-level simulation. MCPM improves simulation speed while maintaining accuracy by modeling memory access behavior in detail and simplifying the modeling of arithmetic operations. This design enables MCPM to support both regular GEMM accelerators and irregular SpMSpM accelerators with high simulation efficiency.

A. Vector-Level Performance Model

Figure 4 shows the unified abstraction of MCPM for the execution of different matrix accelerators. For a MatMul containing n data-vector requests, let v_i denote the i-th data vector being processed. MCPM defines four latency parameters for each data vector v_i: I_i denotes the issue interval between adjacent data-vector requests, R_i denotes the read latency of the i-th data vector, C_i denotes the latency required for this data vector to participate in computation, and W_i denotes the writeback latency of the result or partial sum generated after the computation of this data vector.

Here, both R_i and W_i are used to describe the data transfer process between the memory structure and the compute logic during accelerator execution. For example, when the accelerator uses SPM as on-chip storage, R_i only represents the latency of reading data from the SPM to the compute logic, and does not include the time required to move data from DRAM to SPM. The latter is modeled by explicit data movement instructions and DMA operations.

This work denotes the issue time of the i-th data-vector request as shown in Equation 1. Here, $I_0 = 0$ indicates that the first request is issued immediately when the computation starts. For $k \geq 1$, I_k represents the issue interval between the k-th data-vector request and the $(k+1)$-th data-vector request.

$$T_i = \sum_{k=0}^{i-1} I_k, \quad i \geq 1 \tag{1}$$

Based on the above definitions, the time L_i experienced by the i-th data vector from starting computation to generating its results is given by Equation 2. Here, T_i represents the elapsed time before the i-th data vector request is issued, while R_i, C_i, and W_i correspond to the read, computation, and writeback of this data vector, respectively. For the stationary stage, a data vector is typically only read into the compute logic and temporarily stored, and therefore does not incur writeback behavior, i.e., $W_i = 0$. Its computation latency C_i mainly corresponds to the propagation time of data in the on-chip network or compute logic. A computation process completes only after all data vectors have finished reading, computation, and writeback. Therefore, the completion time of a MatMul can be represented by the completion time L_n of the last data vector.

$$L_i = T_i + R_i + C_i + W_i = \sum_{k=0}^{i-1} I_k + R_i + C_i + W_i, \quad i \geq 1 \tag{2}$$

The above equations provide the theoretical basis for constructing the cycle-level performance model. Among these parameters, I_i, R_i, and W_i are determined by events that actually occur during simulation, while C_i is simplified by MCPM according to the accelerators' computational datapath. This avoids full cycle-level simulation of every compute unit inside the compute logic, thereby improving simulation speed while maintaining simulation accuracy. The following introduces how each component is modeled.

B. Modeling Writeback Request Trigger Time

In Equation 2, W_i represents the time required to send the result of the i-th data vector to the memory system after its computation is completed. However, the trigger time of writeback requests differs across accelerators, and writeback does not always strictly start after computation completion. For example, some accelerators need to wait until the output data are fully generated before starting writeback, while others continuously generate partial sums and trigger writeback during the streaming stage. Therefore, before computing W_i, it is necessary to determine when writeback requests are issued during accelerator execution. Table III summarizes the writeback request trigger times of the accelerators studied in this work.

TABLE III: Writeback behaviors of representative GEMM and SpMSpM accelerators

Name	Stage	Actual Memory Access Time
OS SA	WB	$2d-1$ cycles after all inputs are readys
WS SA	STR	$2d-1$ cycles after each input vector is ready
Gust	STR	After each vector completes computation
OP	WB	After all computation completes and each partial sum is reduced
Sigma	STR	After each vector completes computation

The writeback behaviors of different accelerators can be mainly divided into two categories. The first category has a relatively independent writeback stage, where writeback requests are usually issued only after the output data are fully generated. For example, an OS SA needs to wait until partial sums propagate to the bottom of the array before generating the corresponding writeback requests. Since OS SAs typically support overlapping computation and writeback through hardware double buffering, the first output vector is generated exactly after the last input vector completes computation. For the OP accelerator, partial sums need to be merged after all computations are completed, and the merged results are then written back to the memory system. Since each partial sum must be obtained through the reduction network, MCPM simplifies the latency of the partial-sum merging process to the same as the depth of the reduction network, adds this latency to C_i, and does not model the merging of each partial sum individually.

The second category does not have an independent hardware writeback stage, and its result generation process is coupled with the streaming stage. For example, WS SA, Gust accelerator, and Sigma accelerator continuously generate results or partial sums during the streaming stage. For a WS SA, once an input vector is ready, the corresponding output vector can be generated after the propagation latency. Assuming that the array size is $d \times d$, this propagation latency can be expressed as $2d-1$ cycles, which is exactly the time for the computation of this vector. For Gust and Sigma accelerators, MCPM does not model the writeback time of every partial sum generated during the streaming stage. Instead, it assumes that partial sums are first written into an intermediate memory (IM), and are then uniformly written back to the memory system from the IM after the streaming stage finishes. By introducing this virtual writeback stage, MCPM preserves the memory access overhead of writeback while avoiding the complexity of precisely modeling the generation and writeback time of each partial sum.

C. Element-Level Performance Modeling

To obtain R_i, C_i, and W_i in Equation 2, MCPM further decomposes each data vector into multiple data elements, models their memory access, computation, and writeback time at the element level, and then aggregates them to obtain the vector-level latency.

Since SpMSpM accelerators usually compress matrices using sparse formats, different elements within the same data vector may have non-contiguous memory addresses and different memory access times. Therefore, as shown in Figure 5a, the read latency of the i-th data vector is determined by the element whose read completes last:

$$R_i = \max_{x \in D_i} r(x) \tag{3}$$

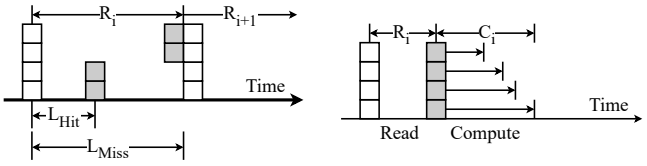
where D_i denotes the set of data elements contained in the i-th data vector, x denotes one data element in this set, and $r(x)$ denotes the time required to complete the read request for element x. This time needs to be obtained through cycle-level simulation while considering the state of the memory system. Although SAs have regular memory access behaviour, the above equation can also be applied to SAs through element-level accesses to model factors that affect performance, such as bank conflicts.

Similarly, different elements within the same data vector may also have different propagation times in the compute units. Taking an SA as an example, input data needs to be injected into the array in a skewed manner through synchronization FIFOs. For data input from the left side, the computation time increases row by row from top to bottom; for data input from the top, the computation time increases column by column from left to right. As shown in Figure 5b, the vector-level computation latency is determined by the element whose computation completes last, and can be expressed as:

$$C_i = \max_{x \in D_i} c(x) \tag{4}$$

Here, $c(x)$ denotes the propagation and computation time of data element x in the compute logic. Although different elements within the same data vector may require different computation times, since MCPM adopts a synchronous execution model, the vector-level computation latency C_i can be simplified as the time required by the element that finishes computation the latest. For the OS and WS SAs modelled in this work, if the array size is $d \times d$, the latency C_i required for a data vector to complete propagation and computation is $2d-1$. For the SpMSpM accelerators modelled in this work, their compute logic mainly consists of a distribution network and a reduction network. Therefore, the vector-level computation latency is jointly determined by these two components. MCPM assumes that the distribution network adopts an N-input, N-output non-blocking topology [8], [10], which is responsible for sending input data to the multiplication units. In this case, its latency C_i^{dist} can be represented as $2 \times log(N) + 1$. The reduction network is responsible for reducing multiplication results into partial sums. Since the final result needs to be output from the root node of the multiplier-adder tree, MCPM uses the depth of the multiplier-adder tree to estimate the reduction latency, i.e., $C_i^{\text{red}} = log(M_n) + 1$. Here, M_n denotes the number of multiplication units. In summary, the computation latency of the i-th data vector in the SpMSpM accelerator can be represented as: $C_i = C_i^{\text{dist}} + C_i^{\text{red}}$.

The writeback latency can also be aggregated from element-level writeback behavior. Let O_i denote the set of data elements that need to be written back for the i-th data vector, and let $w(x)$ denote the time required for element x to be



(a) Memory access (b) Computation

Fig. 5: Fine-grained memory access and computation latency

successfully sent to the memory system. Then, the writeback latency of the i-th data vector is:

$$W_i = \max_{x \in O_i} w(x) \tag{5}$$

Unlike the read process, a writeback request can be considered complete once the data is sent to the memory system, without waiting for the entire memory access process to finish. However, the writeback process may incur additional stalls due to structural hazards such as cache port contention or SPM bank conflicts, thereby affecting W_i.

D. Modeling Memory Request Intervals

In addition to R_i, C_i, and W_i, the issue interval I_i between data-vector requests is also an important factor affecting accelerator performance. I_i reflects the impact of the dataflow, storage structure, and control logic on the memory access behavior of the accelerator. MCPM adopts a synchronous request model for read memory accesses, where the $(i+1)$-th data-vector request needs to wait until the i-th data vector has been read before it can be issued. Therefore, the request issue interval can be approximated as:

$$I_i = R_i, \quad i \geq 1 \tag{6}$$

For accelerators that use SPM, if SPM accesses can be pipelined and no bank conflict occurs, the issue interval between consecutive data-vector requests can be one cycle. In practice, when multiple accesses contend for the same bank, subsequent requests need to wait until bank arbitration is completed before they can be issued. In this case, the effective issue interval between adjacent data-vector requests is no longer fixed to one cycle, and can be represented as:

$$I_i = 1 + B_i, \quad i \geq 1 \tag{7}$$

where B_i denotes the additional stall cycles caused by bank conflicts, port contention, or other structural hazards. MCPM can describe the impact of both cache-based and SPM-based structural conflicts on memory access behavior via I_i.

E. Modeling Accelerator Pipeline Parallelism

The above model describes the issue, read, computation, and writeback latencies of data-vector requests within a single computation process. However, in real MAs, different stages are often not executed in a completely serial manner. Since MAs usually adopt dataflow-based execution, once the data required by the current stage has been read and entered the compute logic, the accelerator can start the next stage or the next matrix instruction, while the data of the current stage may continue computing and propagating through the compute array or reduction network. Therefore, when modeling pipeline parallelism, MCPM does not require the current data vector to complete all computation and writeback before starting subsequent data accesses. Instead, after the i-th data vector has been read and enters the compute logic, i.e., after R_i, MCPM allows the read request for the $(i+1)$-th data vector to be issued.

V. Memory-Centric Accelerator Representation

Unlike the regular memory access pattern of SAs, the memory access behavior of SpMSpM accelerators depends on indirect addressing, and memory addresses cannot be directly generated using only a base address, strides, and sizes. To address this issue, MILES proposes a Memory-Centric Accelerator Representation (MCAR) based on MCPM. MCAR defines the core execution stages of an accelerator in the form of C kernel functions, and characterizes their memory access behavior by marking address streams. Unlike other methods that target broader domains and provide external accelerator descriptions, such as Gem5-Aladdin [19] and Gem5-SALAM [20], MCAR only needs to describe the key memory behavior of an accelerator, and can achieve fast performance simulation with the support of MCPM. Based on MCAR, MILES implements three accelerators, namely Gust, OP, and Sigma.

The following uses the Gust accelerator as an example to introduce how MCAR is used in Figure 6. The implementations of the OP and Sigma accelerators are detailed in Appendix VIII. Figure 6a shows the baseline implementation of Gustavson SpMSpM, which contains three address streams: `A->values`, `B->values`, and `C->values`. Among them, `A->values` serves as the stationary matrix, while `B->values` serves as the streaming matrix. Figure 6b defines an instruction that completes the computation for one row of matrix A at a time. By repeatedly invoking this instruction M times, the complete matrix multiplication can be performed. Figure 6c shows how MCAR defines the behavior of this instruction. The behavior of this instruction can be roughly described as follows. The stationary stage reads one row of matrix A stored in CSR format. The streaming input stage locates the corresponding rows in matrix B according to the column indices of the nonzero elements in the current row of matrix A, and reads a group of nonzero elements from these rows at the same offset in each iteration, forming streaming data vectors and triggering computation. After streaming, the writeback stage writes results back to one row of matrix C. Since the number of nonzero elements in a stationary row of matrix A may exceed the number of multipliers, the row of matrix A needs to be processed in blocks in an actual implementation. The blocking code is omitted here for brevity.

The code uses the `MARK_STREAM` macro to mark address streams. The blue-marked `STREAM_A_val` indicates that the row specified by variable `i` is loaded into the accelerator and kept stationary. The yellow-marked `STREAM_B_val` models the process of streaming matrix B into the accelerator. The green-marked `STREAM_C_val` corresponds to the

```
for (int i = 0; i < M; ++i) {
  for (int ap = A->rowptr[i]; ap < A->rowptr[i + 1]; ++ap) {
    int k = A->colidx[ap];
    for (int bp = B->rowptr[k]; bp < B->rowptr[k + 1]; ++bp) {
      int j = B->colidx[bp];
      C->values[i][j] += A->values[ap] * B->values[bp];
```

(a)

```
for (int i = 0; i < M; ++i)
  gustavson_spMspM(A, B, C, i, M, N, K); // A gust instruction that computes a row
```

(b)

```
int8_t A_row[K];
int32_t C_row[N];
// STA Stage
for (int ap = A->rowptr[i]; ap < A->rowptr[i + 1]; ++ap)
  A_row[ap] = MARK_STREAM(&A->values[ap], "STREAM_A_val");
MILES_wait_load(); // Synchronization to wait loads to finish

// STR Stage
int offset = 0;
while (1) {
  int any = 0;
  for (unsigned int ap = A->rowptr[i]; ap < A->rowptr[i + 1]; ++ap) {
    int bp = B->rowptr[A->colidx[ap]] + offset;
    if (bp >= B->rowptr[A->colidx[ap] + 1]) break;
    int j = B->colidx[bp];
    C_row[j] += A_row[ap] * MARK_STREAM(B->values[bp], "STREAM_B_val");
    any = 1;
  }
  if (!any) break;
  offset++;
  MILES_wait_load(); }
MILES_wait_load_delay();

//WB Stage
for (int n = 0; n < N; n++) {
  if(C_rows[n] == 0) continue;
  MARK_STREAM(&C->values[nnz++], "STREAM_C_val") = v; }
MILES_wait_store();// Synchronization to wait stores to finish
MILES_ci_finish();// Indication of instruction finishes
```

(c)

Fig. 6: (a) C implementation of the Gustavson algorithm; (b) the `gustavson_spMspM` instruction; (c) accelerator behavior representation based on MCAR

writeback stage after computation completes. `MARK_STREAM` wraps the `__builtin_annotation` function and is used to mark memory instructions. When MILES instruments `gustavson_spMspM`, it only focuses on the marked `load` or `store` instructions, while all other unmarked memory instructions are ignored. After instrumentation, this function is removed. During trace collection, whenever MILES encounters a marked instruction, it writes the corresponding address to a file named after the corresponding address stream label and records this label in the address stream order file.

The function `MILES_wait_load` is a primitive provided by MILES, which instructs the simulator to wait until all `load` instructions have completed before issuing subsequent memory requests. `MILES_wait_store` waits until the write requests associated with all `store` instructions have been issued. `MILES_ci_finish` indicates that the current instruction has finished execution and must be placed at the end of the function. In addition, MILES also provides the `MILES_wait_load_delay` primitive, which waits for an additional `delay` cycles after all read requests have completed before allowing subsequent memory requests to be generated. This primitive is used to describe cases where writeback requests cannot be issued until the related computation has completed. When MILES encounters the above primitives during trace collection, it writes the corresponding markers into the address stream order file, and the simulator implements the corresponding functionality for these markers during replay.

It should be noted that MCPM models input and output data using vectors as the basic unit, whereas MCAR does not directly represent vector-level requests. Instead, MCAR jointly represents them using multiple discrete memory requests marked by `MARK_STREAM`. Each marked `load` or `store` instruction corresponds to one data element in a vector, while vector boundaries are defined by synchronization primitives such as `MILES_wait_load` and `MILES_wait_store`. In this way, MCAR decomposes the vector-level memory accesses in MCPM into finer-grained element-level memory behavior, while preserving the characteristics of data parallelism.

The collected address streams must be consistent with the actual behavior of the target accelerator. The baseline version shown in Figure 6a cannot be directly used for address stream collection, because `C->values` is frequently read and written. Directly marking and collecting these addresses would introduce a large number of redundant memory requests in MILES, resulting in memory access behavior that is inconsistent with that of a real accelerator. In addition, the example in Figure 6 does not consider the constraints of instruction bit width and format. In real scenarios, the same functionality often needs to be completed cooperatively by multiple instructions. Therefore, MILES provides an instruction `MILES_fake` with no functionality, which allows users to insert several such instructions before and after a custom instruction to emulate the execution effect of multiple instructions.

VI. Implementation of the Cycle-Level Simulation Framework

MILES extends CISim with a coprocessor model. When the CPU executes a matrix instruction, it sends the instruction to the coprocessor. The coprocessor then invokes the memory request generator (MRG) according to the instruction type and configuration file to generate the corresponding data-vector read and write requests. Subsequently, MILES tracks the issue, stalling, and completion of these requests across different storage structures through cycle-level simulation and combines this process with MCPM to model the accelerator's performance.

A. Coprocessor Microarchitecture

The coprocessor adopts a microarchitecture model similar to Gemmini [22], as shown in Figure 7. Instructions from the CPU are first buffered in the command queue (CmdQ) and then enter the reservation station. The reservation station contains three queues for different types of instructions: the execute queue, the load queue, and the store queue. Compute

instructions enter the execute queue, while data movement instructions from DRAM to SPM and from SPM to DRAM enter the load queue and the store queue, respectively. The ROB detects data hazards by checking whether the SPM address ranges accessed by different instructions overlap. Once an instruction is ready, it enters the corresponding execute, load, or store controller. The three controllers are allowed to execute out of order with respect to each other, while instructions within each controller are processed in order.

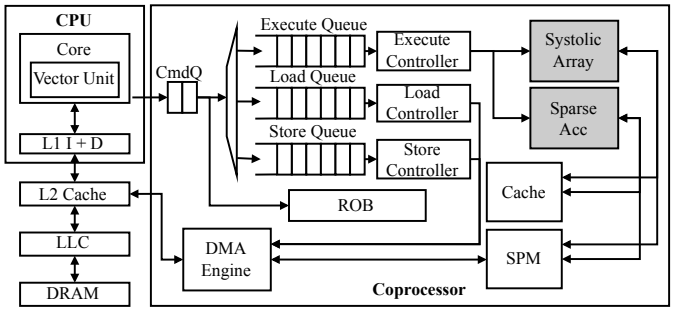


Fig. 7: Coprocessor microarchitecture model

B. Configuration File for MCRA

MILES requires a configuration file for MCAR. Figure 8 shows the basic format of this configuration file. In this file, `MILES_functions` specifies the functions used to represent custom instructions, `MILES_traces` provides the trace files corresponding to different address streams in the form of key-value pairs, `order_file` specifies the filename of the address stream order file, and `mem_type` is used to set the memory structure type used by each address stream. In this example, all address streams use the cache.

```
cosim_functions: ["gustavson_SpMSpM"]
stream_traces:
    STREAM_A_val: stream_STREAM_A_val.txt
    STREAM_B_val: stream_STREAM_B_val.txt
    STREAM_C_val: stream_STREAM_C_val.txt
order_file: stream_order.txt
mem_type:
    STREAM_A_val: cache # or spm
    STREAM_B_val: cache
    STREAM_C_val: cache
stream_dep_args:
    gustavson_SpMSpM: ["spmsrc", "spmdst", "size", "size", "no", ...]
```

Fig. 8: Configuration file format for custom instruction and accelerator behavior definition

When address streams use the SPM, data needs to be moved explicitly between DRAM and the SPM through data movement instructions. Therefore, MILES needs to construct data dependencies between custom instructions and data movement instructions, as well as dependencies among different custom instructions. MILES currently supports two types of data movement instructions, `mvin` and `mvout`. The `mvin` instruction moves data from DRAM to the SPM, while the `mvout` instruction writes data from the SPM or accumulator back to DRAM.

To support dependency construction, MILES introduces the `stream_dep_args` field to specify the type of each argument of the function that represents a matrix instruction. Specifically, `spmsrc` indicates that the argument corresponds to a data address read from the SPM, `spmdst` indicates that the argument corresponds to a data address written to the SPM, and `size` indicates the size of the corresponding read or written data. Input arguments that are irrelevant to dependency construction are marked as `no`. MILES associates addresses with data sizes according to their argument order. Therefore, the number of `size` entries must be consistent with the total number of `spmsrc` and `spmdst` entries. When an instruction enters the ROB, MILES constructs data dependencies based on the range of SPM addresses.

C. Matrix Accelerator Simulation Based on MCPM

The implementation of MCPM in MILES relies on the memory request generator (MRG). In the model shown in Section IV-A, I_i is not a predefined fixed parameter. Instead, it is jointly determined during simulation by synchronization primitives, accelerator state, and memory system state. Specifically, the MRG determines whether subsequent requests can continue to be generated according to the synchronization primitives in the address stream order file, and determines the actual issue time of each data-vector request by considering system states such as cache port availability, SPM bank conflicts, and request queue occupancy. R_i and W_i are obtained through cycle-level simulation of the memory system, while C_i is simplified and determined by MCPM according to the accelerators' computing logic. The performance models of both GEMM and SpMSpM accelerators are built based on MCPM, and their main difference lies in how memory requests are generated. When the coprocessor model receives a matrix instruction from the CPU, it first determines the instruction type according to the configuration file and then invokes the corresponding memory request generation process. For SA instructions, the MRG directly generates read and write requests according to parameters encoded in the instruction, such as DRAM or SPM addresses and data block sizes. For SpMSpM accelerator instructions, the memory request generation process is defined by MCAR.

MILES implements OS and WS SA instructions similar to those in Gemmini [22]. Specifically, `preload` is used in WS mode to preload data from the SPM into the SA; `matmul` and `matmul_out` are used to execute GEMM, where the latter is dedicated to OS dataflow and writes the partial sums temporarily stored in the array into the accumulator after GEMM completes. The SA model in MILES assumes that input data can only be read from the SPM, and partial sums can only be written to the accumulator. Data movement between DRAM and the SPM, as well as between the accumulator and DRAM, is performed by the `mvin` and `mvout` instructions, respectively.

For accelerator instructions represented by MCAR, the MRG first reads the address stream order file specified in the configuration file to determine the issue order of different address streams. It then reads addresses from each

address stream file according to this order and generates the corresponding memory requests. In the address stream order file, the value -2 corresponds to `MILES_wait_load`, which indicates that subsequent memory request generation should be paused until all previous read requests have completed. The value -4 corresponds to `MILES_wait_load_delay`, which indicates that after all previous read requests have completed, the MRG should continue to stall for `delay` cycles before resuming subsequent memory request generation. The value -3 corresponds to `MILES_wait_store`, which indicates that subsequent memory request generation should be paused until all previous write requests have completed. The value -1 corresponds to `MILES_ci_finish`, indicating that the memory request generation process of the current custom instruction has ended, and the MRG will not generate any new memory requests.

When the data of a read request returns, MILES marks the request as memory-access completed. At this point, the corresponding data vector has satisfied the conditions for subsequent memory request generation or synchronization primitive checking. However, memory-access completion does not mean that the corresponding matrix instruction has completed. According to MCPM, after completing reading a data vector, it still needs to go through the computational latency C_i, and may further incur the writeback latency W_i. Therefore, MILES maintains both the memory-access completion time and the computation completion time for each data vector. The former is used to drive subsequent memory request generation and synchronization primitive checking, while the latter is used to indicate the completion time of the computation of each data vector. For `MILES_wait_load_delay`, the stall cycles are counted after all data becomes ready, and are generally set to the same number of cycles as the reduction latency C_i^{red}. Only when all data vectors associated with a matrix instruction have completed reading, computation, and writeback will the instruction be marked as completed.

VII. Experiment

A. Validation Against BOOM+Gemmini

1) Experimental Methodology: We validate the modeling accuracy of MILES for a system consisting of a CPU and an SA by comparing it with RTL simulation results from BOOM [24]+Gemmini [22]. The hardware configuration used in the experiments is shown in Table IV. The CPU adopts the Large BOOM configuration, while Gemmini uses a 16×16 SA, a 256KB SPM, and a 64KB accumulator. The memory system includes two levels of cache and 2GB DDR3 memory.

TABLE IV: Hardware configurations

Large BOOM	8-width fetch; 3-width pipeline; 96-entry ROB; 24-entry LDQ/STQ; 72-entry IQ; 3 ALUs; 1 LD/ST; 1 BRU; 1 FPU; 1 iDIV; 1 iMUL; 1 fDIV
Gemmini	16×16 SA; 256KB SPM; 64KB accumulator
L1 ICache	32KB / 8-way / 2-cycle hit latency
L1 DCache	32KB / 8-way / 4-cycle hit latency
L2 Cache	512KB / 8-way / 10-cycle hit latency
DRAM	2GB DDR3

The benchmark programs used in the experiments are shown in Table V, covering multiple DNN models, GEMM kernels, and Conv2D kernels. Among them, G1 to G4 correspond to four GEMM configurations with input sizes of 256, 512, 1024, and 2048, respectively. C1 to C4 correspond to four two-dimensional convolution configurations, whose input feature map sizes are 7×7, 14×14, 28×28, and 56×56, respectively. The number of input and output channels is the same for each configuration, namely 512, 256, 128, and 64, respectively, and the kernel size is fixed to 3×3. The experiments are based on the DNN implementations in the Gemmini repository. We replace the Gemmini instructions with instructions of MILES, and then compile the programs into LLVM IR and collect traces so that they can be simulated in MILES.

TABLE V: DNN and operator configurations

Category	Name	Configuration
DNN	ResNet50	—
	MobileNet	—
	BERT	Small, Base, Large
GEMM	G1	256×256
	G2	512×512
	G3	1024×1024
	G4	2048×2048
Conv2D	C1	512 channels, 7×7 input feature map
	C2	256 channels, 14×14 input feature map
	C3	128 channels, 28×28 input feature map
	C4	64 channels, 56×56 input feature map

2) Experimental Results: Figure 9 normalizes the total execution cycles simulated by MILES to the execution cycles obtained from RTL simulation. The execution cycles represent the end-to-end runtime of the program on the CPU and MA. A normalized value smaller than 1 indicates that MILES estimates fewer cycles than the RTL simulation. The results show that the geometric mean errors of MILES for the OS and WS SA models are 13.9% and 7.38%, respectively. Since Gemmini does not provide an OS SA implementation for convolution, the corresponding validation results are not shown. The main sources of error are that MILES currently does not support address translation and does not support strided data movement, and the reported error also includes the simulation error from the CPU side. In addition, as shown in Figure 10a, the average simulation speed of MILES reaches $432\times$ that of the 8-thread RTL simulator. Taking G4, which has the largest matrix size, as an example, RTL simulation requires 16 hours, while MILES only takes 440 seconds. This demonstrates that the MCPM-based cycle-level simulator can provide reliable performance evaluation results within a very short time.

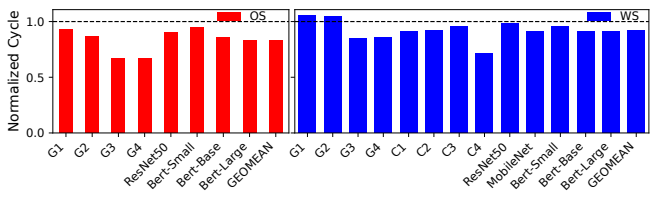
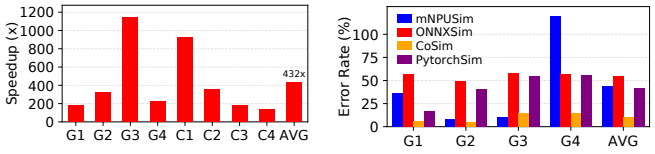


Fig. 9: Comparison between normalized MILES and RTL simulation cycles (left: OS; right: WS)

We further compare the accuracy of MILES with ONNXim [16], mNPUSim [15], and PytorchSim [12]. All simulators use a WS SA, and workloads are the G1 to G4. The results are shown in Figure 10b. The errors of the three simulators are 55.1%, 43.8%, and 42%, respectively, while the error of MILES is only 9%. In addition to the lack of instruction-level OOO CPU modeling, the significant difference in the number of simulated instructions is also an important reason for the large simulation errors of these simulators.



(a) (b)

Fig. 10: (a) Simulation speed comparison between MILES and the 8-thread RTL simulator; (b) accuracy comparison among MILES, ONNXim, mNPUSim, and PytorchSim

Figure 11 shows the number of instructions executed by different simulators during simulation. Taking the G4 matrix multiplication workload (2048×2048) as an example, ONNXim and PytorchSim model a similar number of instructions, about 2.2 million. mNPUSim models about 14 million events to drive the simulation, while the RTL simulation actually executes about 89 million RISC-V instructions. Although matrix computation accounts for most of the execution time, these additional scalar instructions still affect overall performance. In contrast, MILES executes about 53 million LLVM IR, which is closer to the real instruction count. The ISA difference between LLVM IR and RISC-V is the main reason for the mismatch between the two instruction counts, and is also an important source of error between MILES and RTL simulation.

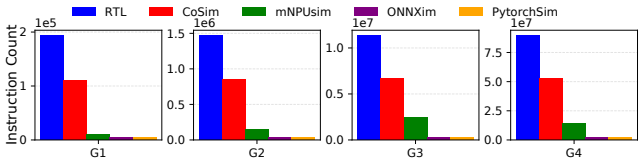


Fig. 11: Comparison of the number of simulated and RISC-V instructions

B. Validation Against sstStonne

1) Experimental Methodology: This section validates the accuracy of the SpMSpM accelerator models implemented based on MCPM and MCAR by comparing the results of MILES with those of the sstStonne simulator. The evaluated accelerators include Sigma, OP, and Gust.

The main configuration parameters of the hardware accelerator and memory system in sstStonne are as follows. All accelerators are configured with 128 multipliers, and both distribution and reduction bandwidth are set to 128. The memory system adopts a two-level cache hierarchy. The L1 Cache has a capacity of 32KB, is 8-way set associative, and has a hit latency of 4 cycles. The L2 Cache has a capacity of 512KB, is 8-way set associative, and has a hit latency of 10 cycles. The main memory adopts the SimpleDRAM model, runs at 500 MHz, and has an access latency of 160 ns. All experiments use 32-bit integer precision.

The sparse matrices used in the experiments are shown in Table VI. These matrices are selected from representative layers in different DNN models [10], where matrix A and matrix B serve as the stationary matrix and the streaming matrix, respectively.

TABLE VI: Sparse matrix dimensions and sparsity

Name	M, N, K	Sparsity of Matrix A (%)	Sparsity of Matrix B (%)
$SQ5$	64, 2916, 16	68	11
$SQ11$	128, 729, 32	70	10
$R4$	256, 3136, 64	88	9
$R6$	64, 2916, 576	89	53
$S\text{-}R3$	64, 5329, 576	89	46
$V0$	128, 12100, 576	90	61
$MB215$	128, 8, 512	50	10
$V7$	512, 144, 4608	90	94
$A2$	384, 121, 1728	70	54

2) Experimental Results: Figure 12 shows the performance comparison between sstStonne and MILES on different sparse matrices. For each test, the two bars on the left and right correspond to the results of sstStonne and MILES, respectively, and the performance error is annotated above the bars. MILES achieves performance errors of 9.7%, 3.7%, and 4.5% on the Gust, Sigma, and OP accelerators, respectively. These results indicate that the MCPM model is reliable. It can be applied not only to SAs with regular memory access behavior, but also to SpMSpM accelerators with irregular memory accesses, providing a practical basis for extending MCPM to more accelerator types in the future.

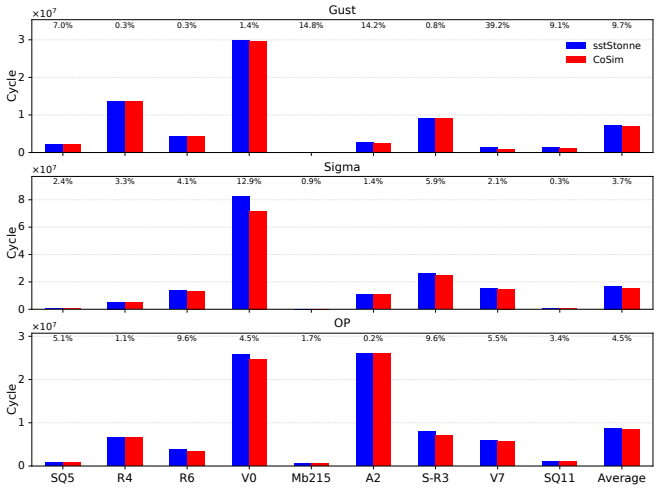


Fig. 12: Cycle count comparison between MILES and sstStonne

This section further compares the simulation speed of the two simulators. As shown in Figure 13, when simulating

the Gust accelerator, the average simulation speed of MILES reaches 20× that of sstStonne. sstStonne accurately models the on-chip interconnect and the internal behavior of PEs, while the MCPM model in MILES relies on detailed memory access modeling and significantly reduces simulation time by simplifying the model of computation, thereby achieving a clear speed advantage.

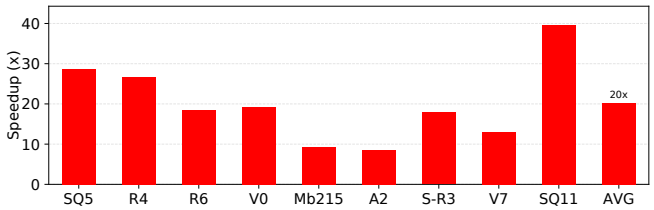


Fig. 13: Simulation speedup of MILES and sstSTONNE

MCPM is a memory-centric accelerator performance model. Therefore, validating its ability to characterize accelerator memory behavior is critical for evaluating the reliability of MILES. Figure 14 compares the L1 and L2 Cache hit rates of MILES and sstStonne on different accelerators. On the Gust, Sigma, and OP accelerators, the L1 Cache hit rate errors of MILES are 0.08%, 0.04%, and 0.017%, respectively, while the L2 Cache hit rate errors are 8.4%, 0.45%, and 8.0%, respectively. The matrices used in the experiments are relatively small, and the streaming stage repeatedly accesses the same matrix, so most data accesses can be served by the L1 Cache. In contrast, the data reuse in the L2 Cache is very low, resulting in many cases with extremely low L2 Cache hit rates. These results show that, with the support of MCPM and MCAR, MILES can model memory access behavior with high reliability.

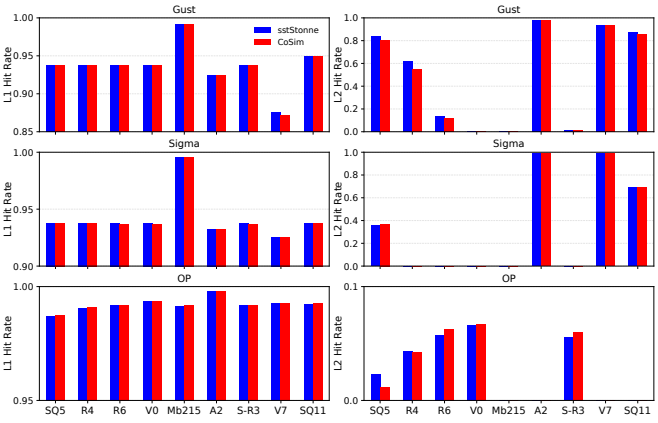


Fig. 14: Comparison of L1 and L2 cache hit rates between MILES and sstStonne

REFERENCES

[1] Intel, "Intel 64 and ia-32 architectures software developer's manual," https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[2] J. P. de Carvalho, J. E. Moreira, and J. N. Amaral, "Compiling for the ibm matrix engine for enterprise workloads," *IEEE Micro*, vol. 42, no. 5, pp. 34–40, 2022.

[3] ARM, "Arm sme," https://developer.arm.com/documentation/109246/0101/SME-Overview/SME-and-SME2.

[4] RISC-V International, "Risc-v integrated matrix extension," https://github.com/riscv-admin/integrated-matrix-extension.

[5] ——, "Risc-v vector-matrix extension," https://riscv.atlassian.net/wiki/spaces/VMEX/pages/554991628/Vector-Matrix+Extension+VME+-+PoW.

[6] ——, "Risc-v attached matrix extension," https://github.com/riscv-admin/attached-matrix-extension.

[7] K. Hegde, H. Asghari-Moghaddam, M. Pellauer, N. Crago, A. Jaleel, E. Solomonik, J. Emer, and C. W. Fletcher, "Extensor: An accelerator for sparse tensor algebra," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 319–333.

[8] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, "Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Ieee, 2020, pp. 58–70.

[9] G. Zhang, N. Attaluri, J. S. Emer, and D. Sanchez, "Gamma: Leveraging gustavson's algorithm to accelerate sparse matrix multiplication," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 687–701.

[10] F. Muñoz-Martínez, R. Garg, M. Pellauer, J. L. Abellán, M. E. Acacio, and T. Krishna, "Flexagon: A multi-dataflow sparse-sparse matrix multiplication accelerator for efficient dnn processing," in *Proceedings of the 28th Acm International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2023, pp. 252–265.

[11] J. Van Delm, A. Lydike, J. Dumoulin, J. Crols, X. Yi, R. Antonio, J. Woodruff, T. Grosser, and M. Verhelst, "The configuration wall: Characterization and elimination of accelerator configuration overhead," in *Proceedings of the 31st Acm International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, 2026, pp. 265–280.

[12] W. Yang, Y. Shin, O. Woo, G. Park, H. Ham, J. Kang, J. Park, and G. Kim, "Pytorchsim: A comprehensive, fast, and accurate npu simulation framework," in *Proceedings of the 58th Ieee/acm International Symposium on Microarchitecture*, 2025, pp. 1363–1380.

[13] S. Xi, Y. Yao, K. Bhardwaj, P. Whatmough, G.-Y. Wei, and D. Brooks, "Smaug: End-to-end full-stack simulation infrastructure for deep learning workloads," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 17, no. 4, pp. 1–26, 2020.

[14] R. Raj, S. Banerjee, N. Chandra, Z. Wan, J. Tong, A. Samajdhar, and T. Krishna, "Scale-sim v3: A modular cycle-accurate systolic accelerator simulator for end-to-end system analysis," in *2025 Ieee International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Ieee, 2025, pp. 186–200.

[15] S. Hwang, S. Lee, J. Kim, H. Kim, and J. Huh, "mnpusim: Evaluating the effect of sharing resources in multi-core npus," in *2023 Ieee International Symposium on Workload Characterization (IISWC)*. Ieee, 2023, pp. 167–179.

[16] H. Ham, W. Yang, Y. Shin, O. Woo, G. Heo, S. Lee, J. Park, and G. Kim, "Onnxim: A fast, cycle-level multi-core npu simulator," *IEEE Computer Architecture Letters*, 2024.

[17] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj *et al.*, "The gem5 simulator: Version 20.0+," *arXiv preprint arXiv:2007.03152*, 2020.

[18] R. Organization, "Spike risc-v isa simulator," https://github.com/riscv-software-src/riscv-isa-sim.

[19] Y. S. Shao, S. L. Xi, V. Srinivasan, G.-Y. Wei, and D. Brooks, "Co-designing accelerators and soc interfaces using gem5-aladdin," in *2016 49th Annual Ieee/acm International Symposium on Microarchitecture (MICRO)*. Ieee, 2016, pp. 1–12.

[20] S. Rogers, J. Slycord, M. Baharani, and H. Tabkhi, "gem5-salam: A system architecture for llvm-based accelerator modeling," in *2020 53rd Annual Ieee/acm International Symposium on Microarchitecture (MICRO)*. Ieee, 2020, pp. 471–482.

[21] F. Muñoz-Martínez, J. L. Abellán, M. E. Acacio, and T. Krishna, "Stonne: Enabling cycle-level microarchitectural simulation for dnn inference accelerators," in *2021 Ieee International Symposium on Workload Characterization (IISWC)*. Ieee, 2021, pp. 201–213.

[22] H. Genc, S. Kim, A. Amid, A. Haj-Ali, V. Iyer, P. Prakash, J. Zhao, D. Grubb, H. Liew, H. Mao *et al.*, "Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration," in *2021 58th*

Acm/Ieee Design Automation Conference (DAC). Ieee, 2021, pp. 769–774.

[23] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz *et al.*, "The rocket chip generator," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, vol. 4, pp. 6–2, 2016.

[24] J. Zhao, B. Korpan, A. Gonzalez, and K. Asanovic, "Sonicboom: The 3rd generation berkeley out-of-order machine," in *Fourth Workshop on Computer Architecture Research with Risc-v*, vol. 5. International Symposium on Computer Architecture Valencia, 2020, pp. 1–7.

[25] A. Samajdar, Y. Zhu, P. Whatmough, M. Mattina, and T. Krishna, "Scale-sim: Systolic cnn accelerator simulator," *arXiv preprint arXiv:1811.02883*, 2018.

[26] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "Mlir: Scaling compiler infrastructure for domain specific computation," in *2021 Ieee/acm International Symposium on Code Generation and Optimization (CGO)*. Ieee, 2021, pp. 2–14.

[27] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017, pp. 1–12.

[28] S. Pal, J. Beaumont, D.-H. Park, A. Amarnath, S. Feng, C. Chakrabarti, H.-S. Kim, D. Blaauw, T. Mudge, and R. Dreslinski, "Outerspace: An outer product based sparse matrix multiplication accelerator," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Ieee, 2018, pp. 724–736.

[29] Z. Zhang, H. Wang, S. Han, and W. J. Dally, "Sparch: Efficient architecture for sparse matrix multiplication," in *2020 Ieee International Symposium on High Performance Computer Architecture (HPCA)*. Ieee, 2020, pp. 261–274.

[30] N. Srivastava, H. Jin, J. Liu, D. Albonesi, and Z. Zhang, "Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product," in *2020 53rd Annual Ieee/acm International Symposium on Microarchitecture (MICRO)*. Ieee, 2020, pp. 766–780.

VIII. Appendix

This appendix presents how MCAR is used to represent the behavior of the OP and Sigma accelerators. To highlight the core idea of MCAR, the algorithms simplify the implementation in MILES and retain only the key memory access and synchronization behavior.

A. Representation for OP Accelerator

Algorithm 1 shows the OP accelerator [10] represented using MCAR. Lines 4–7 correspond to the stationary stage of the accelerator. The STA reads a vector `local_A_vals` of length n from memory, which contains all nonzero elements in the k-th column of matrix A. Line 8 performs synchronization through the `MILES_wait_load` function, indicating that the streaming stage can start only after all data in this vector is ready. Lines 10–21 describe the memory access behavior of the STR stage. The loop in line 12 repeatedly reads $B[k][j]$ to construct a vector of length n, which is then used to perform the outer product with `local_A_vals`.

The synchronization in line 19 ensures that reading the next vector only after the vector composed of $B[k][j]$ is ready, while the synchronization in line 17 ensures that the writeback stage starts only after all previous computations have completed. Lines 23–26 represent the writeback stage of the OP accelerator. For ease of description, Algorithm 1 assumes that the OP accelerator has an unlimited number of multipliers, so it can read all nonzero elements in the k-th column of matrix A at a time. The actual implementation in MILES further considers hardware resource constraints: for an OP accelerator with X multipliers, a vector of length X is fetched from `A.colptr` each time, the column containing each element is recorded, and the corresponding rows of matrix B are then accessed according to the column indices.

For the OP accelerator, the stall time of the `MILES_wait_load_delay` primitive is set to $(C_i^{dist} + C_i^{red}) + C_i^{red}$. This delay consists of two parts: the first part is the computation time required for the input data to generate partial sums after entering the compute logic, and the second part is the reduction time required for the partial sums to generate the final result through the reduction network. Its latency can be approximated as the same as the reduction time of computation.

Algorithm 1 MCAR for OP accelerator

Require: Matrix A (CSC), Matrix B (CSR)
Ensure: Matrix C
1: **for** $k = 0$ **to** $K - 1$ **do**
2: $aStart \leftarrow A.colptr[k]$; $aEnd \leftarrow A.colptr[k+1]$
3: $n \leftarrow aEnd - aStart$
4: **for** $i = 0$ **to** $n - 1$ **do**
5: $local\_A\_rows[i] \leftarrow A.rowidx[aStart + i]$
6: $local\_A\_vals[i] \leftarrow$ **MARK_STREAM**($\&A.values[aStart + i]$, "STREAM_A")
7: **end for**
8: **MILES_wait_load**()
9: $bStart \leftarrow B.rowptr[k]$; $bEnd \leftarrow B.rowptr[k+1]$
10: **for** $bp = bStart$ **to** $bEnd - 1$ **do**
11: $j \leftarrow B.colidx[bp]$
12: **for** $at = 0$ **to** $n - 1$ **do**
13: $val_B \leftarrow$ **MARK_STREAM**($\&B.values[bp]$, "STREAM_B")
14: $values[local\_A\_rows[at]][j] \leftarrow values[local\_A\_rows[at]][j] + val_B \times val_A$
15: **end for**
16: **if** $j = N - 1$ **then**
17: **MILES_wait_load_delay**()
18: **else**
19: **MILES_wait_load**()
20: **end if**
21: **end for**
22: **end for**
23: **for** $pos = 0$ **to** $M \times N - 1$ **do**
24: **MARK_STREAM**($\&C.values[pos]$, "STREAM_C") $\leftarrow values[pos]$
25: **MILES_wait_store**()
26: **end for**
27: **MILES_ci_finish**()

B. Representation for Sigma Accelerator

Algorithm 2 shows the Sigma accelerator [8] represented using MCAR. Lines 3–12 correspond to the STA stage, where a vector `aVal` of length nnz is read from memory. This vector contains all nonzero elements in the i-th row of matrix A. `startK` and `endK` denote the starting and ending positions of the nonzero elements in the i-th row, respectively. In line 13, synchronization is performed whenever `aVal` in a row has been read, ensuring that all data in the vector is ready. Lines 14–30 correspond to the STR stage. For each column of matrix B, a vector `bVec` is read, which consists of all nonzero elements whose indices correspond to `aVal`, ranging from row index `startK` to `endK`. Line 20 indicates that synchronization is performed after each `bVec` is read. Lines 31–35 are used to write back a vector of length N. This vector is obtained by computing the inner products between the i-th row of matrix A and the N columns of matrix B.

Compared with Algorithm 2, the implementation in MILES further considers two cases: (1) when the number of nonzero elements in the i-th row of matrix A is larger than the number of multipliers X, the i-th row is processed in blocks; and (2) when the number of nonzero elements in the i-th row is smaller than X, adjacent rows are concatenated.

Algorithm 2 MCAR for Sigma

Require: Matrix A (Bitmap) Matrix B^T (Bitmap)
Ensure: Matrix C

for $i = 0$ **to** $M - 1$ **do**
  $startK \leftarrow \infty$; $endK \leftarrow 0$; $nnz \leftarrow 0$
  for $k = 0$ **to** $K - 1$ **do**
    if $A.bitmap[i][k] \neq 0$ **then**
      if $startK == \infty$ **then**
        $startK \leftarrow k$
      end if
      $endK \leftarrow k$; $vecK[nnz] \leftarrow k$; $idx \leftarrow A.rowptr[i]$
      $aVal[nnz] \leftarrow$ **MARK_STREAM**($\&A.values[idx]$, "STREAM_A")
      $nnz \leftarrow nnz + 1$
    end if
  end for
  MILES_wait_load()
  for $j = 0$ **to** $N - 1$ **do**
    $t \leftarrow 0$
    for $kk = startK$ **to** $endK$ **do**
      if $B.bitmap[j][kk] \neq 0$ **then**
        $idx \leftarrow B.rowptr[j]$
        $bVal \leftarrow$ **MARK_STREAM**($\&B.values[idx]$, "STREAM_B")
        if $t < nnz$ **and** $vecK[t] == kk$ **then**
          $bVec[j][t] \leftarrow bVal$; $t \leftarrow t + 1$
        end if
      end if
    end for
    if $j = N - 1$ **then**
      MILES_wait_load_delay()
    else
      MILES_wait_load()
    end if
  end for
  for $j = 0$ **to** $N - 1$ **do**
    $acc \leftarrow \sum_{t=0}^{nnz-1} aVal[t] \times bVec[j][t]$
    $idx \leftarrow i \times N + j]$
    MARK_STREAM($\&C.values[idx$, "STREAM_C") $\leftarrow$ $C.values[i \times N + j] + acc$
  end for
  MILES_wait_store()
end for
MILES_ci_finish()