A Simulation Framework for Memristor-Based Heterogeneous Computing Architectures

Haikun Liu, *Member, IEEE*, Jiahong Xu, Xiaofei Liao, *Member, IEEE*, Hai Jin, *Fellow, IEEE*, Yu Zhang, *Member, IEEE*, and Fubing Mao

***Abstract*—Memristor-based accelerator (MBA) has demonstrated its capability in accelerating matrix-vector multiplication (MVM) with high performance and energy efficiency. However, it is hard to determine whether and how well an application can benefit from MBAs in a heterogeneous computing architecture. In this article, we propose a simulation framework called MHSim to evaluate the energy efficiency and performance of applications running with both MBAs and CPUs. MHSim provides flexible system-level interfaces and circuit-level simulation models for designers to configure heterogeneous computing architectures. We design a general-purpose MBA which enables floating-point computation models for general matrix–matrix multiplication (GEMM). Our simulation framework can quantify the performance and energy efficiency of different MBA architectures for various applications. We validate our simulation framework with SPICE and evaluate the accuracy and performance of MBAs via several case studies. Experimental results demonstrate that the deviations of energy consumption and latency are only 0.47% and 0.49% on average compared with SPICE-based simulation.**

***Index Terms*—Heterogeneous computing architecture, memristor, memristor-based accelerator (MBA), simulation framework.**

I. Introduction

MATRIX multiplications are fundamental operations in a wide range of applications, such as convolutional neural networks (CNNs) [1] and graph processing [2], [3]. They often lead to massive data movement between CPU and main memory, which has become a major performance bottleneck in traditional von Neumann architectures. To minimize data movement, a promising approach is to exploit processing-in-memory (PIM) architectures using nonvolatile memory (NVM) devices, such as memristor [4]–[7], phase change memory (PCM) [8], and flash [9], [10].

Emerging memristor-based PIM accelerators have demonstrated their efficiency in accelerating matrix-vector multiplication (MVM) operations. An element of a matrix can be encoded as an analog charge state or a conductance state of a memristor. Thus, a whole matrix can be mapped in memristor-based crossbar (XB) arrays, and the *in-situ* MVM operation can be conducted in a constant time ($O(1)$ complexity) by exploiting Kirchhoff's circuit laws. Compared with traditional CMOS circuits, memristor-based accelerators (MBAs) eliminate data movement and enhance the parallelism of computation, and thus significantly improve energy efficiency and application performance.

Recently, many studies [1], [2], [4], [5], [11]–[15] have explored MBAs for various applications, such as neural networks [1], [4], [5], [13], graph processing [2], [14], [15], and scientific computing [11]. However, there is not a public and general-purpose simulation framework that can support all these applications. Researchers usually have to evaluate the performance and energy efficiency of their MBA designs based on in-house simulators, which are integrated into application programming frameworks, such as Caffe or TenserFlow [1], [13], [16]–[19]. Thus, it is difficult to evaluate different state-of-the-art works fairly. Some existing open-source simulation frameworks, such as MNSIM [20], DL-RSIM [16], NeuroSim [21], AIHWKit [18], and PUMAsim [13] are designed for special goals and applications. Without a high-level abstraction for MVM operations, existing memristor simulators are not sufficient for diverse purposes and applications. Moreover, for a given MBA design, simulation results (energy consumption and speedup) generated by different simulators may be significantly different. On the other hand, existing MBA simulators can only simulate a small portion of computation (such as iterative MVMs) that can be offloaded to MBAs. It is hard to use them to estimate the overall performance of complex applications in a heterogeneous computing architecture comprising both CPUs and MBAs. To facilitate the exploration of heterogeneous computer architectures with MBAs, it is essential to develop an easy to use and flexible simulation framework to support diverse applications and heterogeneous computing architectures.

However, there remain several challenges to develop such a simulation framework. First, existing architectural simulators, such as GEM5 [22] or ZSim [23] do not provide interfaces for MBAs. Although there have been some MBA simulators, such as MNSIM [20] and NeuroSim [21], they mainly focus on simulating the electrical properties of memristor XBs at the behavior level or the circuit level. It is not easy to integrate MBA simulators into architectural simulators. Second, a general-purpose and flexible simulation framework for various

Manuscript received 19 September 2021; revised 11 January 2022; accepted 7 February 2022. Date of publication 17 February 2022; date of current version 22 November 2022. This work was supported by the National Natural Science Foundation of China (NSFC) under Grant 62072198, Grant 61832006, Grant 61825202, and Grant 61929103. This article was recommended by Associate Editor K. Olcoz. *(Corresponding author: Haikun Liu.)*

The authors are with the National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China (e-mail: hkliu@hust.edu.cn; jhxu@hust.edu.cn; xfliao@hust.edu.cn; hjin@hust.edu.cn; zhyu@hust.edu.cn; fbmao@hust.edu.cn).

Digital Object Identifier 10.1109/TCAD.2022.3152385

applications requires high-level abstractions of computation operators that can be accelerated by MBAs. Third, existing MBA simulators have not yet supported floating-point operations in MVMs. Since floating-point numbers are commonly used in many real-world applications, it is necessary to support floating-point operands in matrix multiplications. Furthermore, it is challenging to map different matrices to memristor arrays with high resource utilization and significant performance improvement. Thus, a general and flexible MBA simulator and the corresponding resource management scheme are desired.

In this article, we take the first step to simulate the "CPU + MBA" heterogeneous computing architecture by extending ZSim [23] and NeuroSim [21] simulators. We propose MHSim [24], a general-purpose and open-source simulation framework for benchmarking all real-world applications in the heterogeneous computing architecture. We use dynamic binary instrumentation (DBI) to hook MVM functions in applications and use a simple performance model to offload MVM operations to the MBA. MHSim provides easy-to-use interfaces to design heterogeneous computing systems and evaluate their performance and energy efficiency with high accuracy, without modifying applications' source codes. The major contributions of this work can be summarized as follows.

1) We design a reconfigurable and easy-to-extend heterogeneous computing architecture that can facilitate the design of different MBAs for various applications with massive MVM operations. The proposed MBA module is integrated with a CPU simulator–ZSim [23] to benchmark different kinds of applications.
2) To extend the adoption of MBAs for most real-world applications, we redesign circuit models in NeuroSim [21] to enable floating-point MVM operations and provide simple APIs for applications to use the MBA easily.
3) To best utilize the memristor resource, we propose a best fit algorithm combining with a binary tree bin-packing (BTBP) algorithm to map weight matrices to memristor XB arrays.
4) We validate the accuracy of MHSim with SPICE. Experimental results show that MHSim leads to only 0.47% and 0.49% deviations in energy consumption and latency compared with the SPICE model, respectively. Our case studies also show that MHSim can quantify the performance speedup of various neural network models and the inference accuracy in MBA architectures.

The remainder of this article is organized as follows. Section II introduces the background of MBA and the related work. Section III describes our design of MHSim. Section IV presents our simulation framework in detail. Section V presents our evaluation methodology and experimental results. We conclude in Section VI.

II. Background and Related Work

A. Memristor and Crossbar Structure

Memristors are the fourth fundamental circuit elements that provide the functional relation between charge and magnetic flux. A memristor cell is actually a variable resistor because the magnetic flux changes along with the charge. With multiple resistance states, a memristor cell is able to encode a range of values with multiple bits in the analog domain. A number of previous proposals [1], [2], [4], [5] have explored memristors as PIM accelerators since memristors in XB arrays can act as resistive random-access memory (ReRAM) and perform MVM operations *in-situ* in the analog domain efficiently.

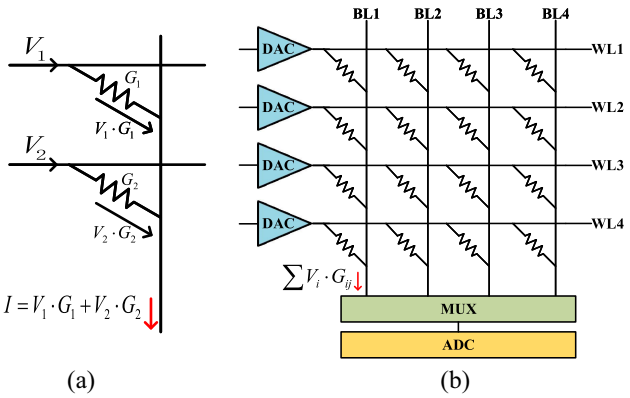


Fig. 1. (a) Analog dot-product operation in a single BL. (b) Analog MVM using an XB array.

A memristor array is mainly comprised of multiple memristor cells, wordlines (WLs), bitlines (BLs), digital-to-analog converter (DAC), and analog-to-digital converter (ADC) circuits. Fig. 1(b) illustrates a 4×4 memristor array. Each memristor cell in the XB is located at the cross points of WLs and BLs. Fig. 1(a) depicts how memristors perform multiplication and accumulation in the memristor XB array. When the XB array performs analog computation, the voltages V_i are applied to each WLs by DACs, and memristor cells act as resistors with conductance G_{ij}. According to Kirchhoff's circuit laws, the output currents I_j of each BLs are equal to $\sum_i V_i G_{ij}$ and read by ADCs. Since the size of an ADC is much larger than the memristor cell, a multiplexer (MUX) is employed to share ADCs with multiple columns for area efficiency. To perform an MVM operation *in-situ*, a matrix element is encoded as a conductance in each memristor cell, and the input vector is encoded as the corresponding input voltages. Finally, the voltage-conductance product can be read at the end of BLs. The analog computation can be formulated as (1). However, due to the analog nature of values encoded in these memristors, only limited precision can be achieved in MVMs and this limits the computation accuracy of MBAs. To address this problem, a few proposals [4], [5] use multiple memristors together to represent a high-precision value through *adder* and *shift* registers

$$\begin{bmatrix} V_1 \\ \vdots \\ V_m \end{bmatrix}^{\mathsf{T}} \times \begin{bmatrix} G_{11} & \dots & G_{1n} \\ \vdots & \ddots & \vdots \\ G_{m1} & \dots & G_{mn} \end{bmatrix} = \begin{bmatrix} I_1 \\ \vdots \\ I_n \end{bmatrix}^{\mathsf{T}}. \quad (1)$$

Compared with the CMOS-based approach, memristor-based *in-situ* MVM operations eliminate data movement of weight matrices and offer much higher computation

parallelism. Thus, memristor XB arrays can significantly improve the performance and energy efficiency of MVM operations.

B. Memristor-Based Heterogeneous Computing Architectures

Memristor XB arrays have been increasingly explored to accelerate applications that contain massive MVMs. Both PRIME [4] and ISAAC [5] propose memristor-based heterogeneous computing architectures to accelerate CNN inference. PRIME is a morphable PIM architecture that exploits shared peripheral circuits to support both memory and computation modes of memristors. ISAAC proposes an MBA heterogeneous architecture that supports interlayer and intratile pipeline to enhance tile parallelism. There have been also a number of studies [1], [25]–[28] on neural network training with memristor XB arrays. TIME [25] proposes an in-memory training architecture to avoid frequent read/write operations during model training. PipeLayer [1] proposes a heterogeneous architecture that uses intra- and inter-layer pipelines to accelerate CNN training. ReGAN [29], LerGAN [12], and ZARA [30] employ MBAs to accelerate generative adversarial networks (GANs) in both testing and training phases. PSB-RNN [31] proposes an MBA architecture using Fourier transform and systolic dataflow to support block circulant compression for recurrent neural networks (RNNs).

Since graphs can be represented as sparse matrices in the vertex-centric model, GraphR [2], RAGra [15], and GaaS-X [14] use memristor XB arrays to achieve in-suit graph computing efficiently. GraphR proposes a streaming-apply execution model to process large graphs. RAGra employs 3-D ReRAM to improve the computing parallelism of graph processing. GaaS-X leverages content addressable memory (CAM) and memristor XB arrays to accelerate sparse graph data processing. Feinberg *et al.* [11] proposed an MBA heterogeneous architecture for scientific computing. They use memristor XB arrays to accelerate sparse MVMs.

Most MBA heterogeneous computing systems use memristor XB arrays as dot-product engines since MVMs are fundamental operations in diverse compute-intensive applications. However, since those MBA architectures are implemented with their in-house simulators, it is difficult to evaluate the performance speedup and energy efficiency of different proposals fairly. To this end, we focus on developing a general simulation framework to evaluate the effectiveness and efficiency of MBAs when they are used to accelerate diverse applications with MVM operations.

C. MBA Simulators

Recently, a few MBA simulators have been designed to simulate nonideal properties of memristors, such as DL-RSIM [16], RxNN [17], GraphRSim [32], MemTorch [19], and AIHWKit [18]. They all focus on the accuracy of MBAs since the nonideal electrical properties of memristor XB arrays may cause significant accuracy degradation. DL-RSIM [16] creates an error table based on an error analytical module and then injects errors during computation to simulate the nonideal properties of memristor cells and sense amplifiers. RxNN [17] provides a fast XB model to simulate errors of MVMs caused by nonideal properties of XB arrays. GraphRSim [32] simulates the accuracy degradation of diverse graph algorithms that use nonideal memristor XB arrays as dot-product engines. MemTorch [19] is a high-level MBA simulator that is integrated into the PyTorch machine learning framework. It focuses on simulating the performance degradation of memristive DNNs introduced by the nonideal device model. Similarly, AIHWKit [18] also simulates the nonideal properties of analog devices and evaluates the impact of these properties on the model accuracy of AI algorithms using the PyTorch framework. Unlike those MBA simulators that focus on simulating the nonideal property of memristors for machine learning programming frameworks, MHSim provides a general-purpose simulation framework that can quantify the performance and energy efficiency of different MBA architectures for various applications besides neural networks.

A few simulators are designed to evaluate the performance speedup and energy efficiency of MBAs for domain-special applications, such as MNSIM [20], NeuroSim [21], and PUMAsim [13]. MNSIM [20] is an MBA simulation platform for neuromorphic computing systems. MNSIM estimates the area, latency, power, and computing accuracy of MBA heterogeneous computing systems based on behavior-level models. However, since MNSIM does not run real applications in a cycle-accurate model, the accuracy becomes a major concern of using MNSIM. NeuroSim [21] provides detailed models to quantify performance, energy consumption, and inference accuracy of MBAs. It is integrated into a multilayer perceptron (MLP) neural network. Since NeuroSim fixes the peripheral circuits according to the MLP algorithm, its scalability is very limited. Moreover, NeuroSim does not provide user-friendly programming interfaces to design heterogeneous computing architectures with memristor devices. Thus, it is hard to construct a new MBA heterogeneous system and run other applications using NeuroSim. Similar to NeuroSim, circuit-level SPICE simulation schemes [33] are also hard to simulate real applications and are not cost efficient in terms of simulation time. PUMAsim [13] is an MBA simulator that simulates instructions compiled from general machine learning applications. PUMAsim assumes the weight in neural network inference accelerators had been mapped into XB arrays before the *in-situ* computation, without considering the cost of data-mapping. Thus, PUMAsim can be only used to simulate MBAs for inference algorithms, limiting its application scope.

Overall, the existing MBA simulators are designed for special purposes or special applications. None of them provides interfaces to integrate them with traditional architectural simulators. Thus, it is hard to use them to simulate complex applications in a heterogeneous computing architecture composed of both CPUs and MBAs. In contrast, MHSim is a general-purpose and reconfigurable simulation framework for designing MBA heterogeneous computing architectures with easy-to-use interfaces. MHSim can evaluate the performance and energy consumption of various applications on heterogeneous computing architectures, without modifying applications' source codes.

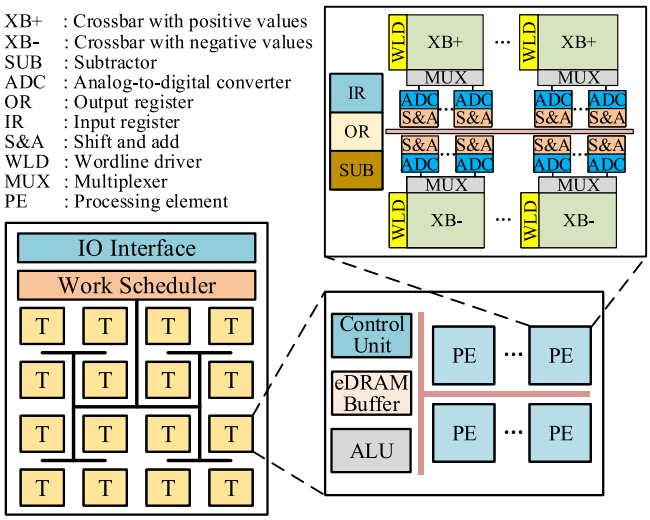


Fig. 2. Architecture of MBA.

III. Memristor-Based Accelerator

In this section, we first present the design of the MBA architecture and its key modules. Second, we present the resource allocation scheme that combines a best fit algorithm with a BTBP algorithm. Third, we illustrate how floating-point MVM operations are enabled in memristor XB arrays. Finally, we introduce the key instructions offered by MBAs and how they are executed in a pipeline.

A. MBA Architecture

To simulate various applications accelerated in an MBA heterogeneous computing architecture, we design a reconfigurable MBA, as shown in Fig. 2. The MBA chip consists of a global I/O interface, a work scheduler, and multiple tiles (T) connected by an H-tree. We choose the tree structure because it shows higher energy efficiency than mesh or ring structures [34], [35]. The MBA chip accesses the main memory via a special channel to avoid bus contention with CPUs. Each tile in an MBA chip is composed of an eDRAM buffer, a simple arithmetic logic unit (ALU), and multiple processing elements (PEs). PEs are responsible for *in-situ* fixed-point MVM operations. Each PE consists of several memristor XB arrays, multiple input registers (IRs) and output registers (ORs), and a subtractor (SUB). The IR and OR are used to store input operands and output results, respectively. Each XB array is connected with a number of MUXs, shift and adder (S&A) units, and ADCs. XB+ and XB− arrays are used to store positive and negative values, respectively. S&A units are utilized to support bit slicing. The SUB combining with XB+ and XB− arrays can support operands with different signs in an MVM operation. In the following, we describe the detailed design of other key modules in the MBA.

Work Scheduler: To perform an analog MVM in an XB array, the weight matrix usually should be loaded in the XB array in advance because the write latency of memristors is rather high for mapping a large matrix to XB arrays. To efficiently utilize the memristor resource for executing multiple MVM operations concurrently, we design a work scheduler

Address (64 bits)	Flag (1 bit)	Tile ID (7 bits)
Matrix 1	1	1
Matrix 2	0	2
...	...	...

(a)

Tile ID (7 bit)	Available Size (25 bits)
1	0.1 MB
2	1.8 MB
...	...
128	2 MB

(b)

Fig. 3. (a) Mapping table records the mappings between weight matrices and tiles. (b) Available resource table records the remaining memristor resource in each tile.

to monitor the memristor resource and dispatch computing tasks to available tiles. The work scheduler remains a table to record the mappings between weight matrices and tiles. As shown in Fig. 3(a), the mapping table contains the address of weight matrices, the index of tiles mapping to matrices, and the state flag that indicates whether the time-consuming data mapping has been completed. When the work scheduler receives a request for matrix multiplication, it first consults the mapping table about the address of the weight matrix. If the mapping table does not contain the corresponding item, the work scheduler would allocate XB arrays to this MVM task based on a best fit algorithm. If the weight matrix is still being mapped to XB arrays (i.e., the state flag is zero), the work scheduler denies this MVM operation at this time and returns the task to the host's CPUs. When a matrix is mapped into XB arrays, we calculate the remaining capacity of each XB array and get the available capacity of each PE. Then, we update the available capacity of each tile in Fig. 3(b).

eDRAM Buffer: The eDRAM buffer in each tile is composed of a data buffer and an instruction buffer. The data buffer consists of an input buffer and an output buffer for caching the input and output data, respectively. The instruction buffer is used to store instructions in each tile.

ALU: To support floating-point MVM operations in MBAs, we use an ALU to align the input operands in advance and generate a series of fixed-point mantissas with the same exponent for the input driver. The ALU can also perform simple arithmetic operations to support general-purpose applications.

Control Unit: The control unit in each tile is responsible for decoding instructions in the buffer and providing control signals to PEs and the ALU.

B. Data Mapping to XB Arrays

To minimize resource fragmentation during mapping matrices to tiles/PEs, we use a best fit algorithm to allocate storage space to matrices in two layers, i.e., tiles and PEs. Given the available sizes of tiles $<S_1, S_2, \ldots, S_n>$ in an ascending order, the scheduler finds the best fit tile with the smallest available size to map this matrix. Since a tile contains multiple PEs with different free capacities $<C_1, C_2, \ldots, C_n>$, we find a PE with the best fit capacity to the matrix. If the matrix is larger than the max capacity of PEs, we partition the matrix and map it into multiple PEs. When a PE is allocated, we calculate the remaining capacity of the PE and the corresponding tile and then resort the available capacity of PEs and tiles.

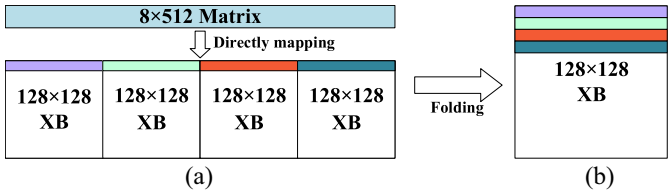


Fig. 4. (a) Directly mapping an 8 × 512 matrix into four 128 × 128 XB arrays. (b) Folding and mapping the 8 × 512 matrix into a single 128 × 128 XB array.

As the shapes of matrices in real-world applications are diverse while XB arrays are square and fix sized, irregular matrices often degrade the resource utilization of XB arrays. We usually have to partition a matrix to fit the XB array. A naive approach is to partition a huge matrix according to the size of XB arrays. For example, if we map an irregular 8 × 512 matrix to four 128 × 128 XB arrays directly, only 6.25% resource is utilized in the four XB arrays, as shown in Fig. 4(a). However, if we fold the 8 × 512 matrix to a 32 × 128 matrix, only one single 128 × 128 XB array is required to store the original matrix, as shown in Fig. 4(b). In this way, the resource utilization is significantly improved by four times.

Although the folding approach achieves maximum resource utilization, it degrades the computation parallelism of MVM operations. As shown in Fig. 4(b), only one 8 × 512 matrix can be activated to perform the MVM operation at a time. Totally, four times analog MVM operations are performed sequentially. This approach cannot make full use of the parallelism of XB arrays and thus lowers the performance speedup. To guarantee high computing parallelism and resource utilization as much as possible, we consider both shapes and sizes of matrices when they are mapped into XB arrays. Actually, data mapping is essentially a 2-D bin-packing problem. We exploit the BTBP algorithm [36] to solve this problem efficiently. Algorithm 1 shows the procedure of the weight mapping. We use a stack structure to store intermediate matrices generated in each matrix mapping.

When a small matrix is mapped, we should find an XB array that can best fit the matrix in the list of the remaining capacities XB$[0, \ldots, N-1]$. If an XB array can accommodate the matrix directly (lines 5–12 of Algorithm 1), we map the matrix into the top-left corner of this XB array. The remaining capacity of this XB array is usually composed of two unused rectangles. To simplify the solving of bin packing and reduce the storage cost of metadata for the remaining capacity, we only record the width and height of the largest unused rectangle. Later, we can still map other small matrices to those unused rectangles based on a best fit algorithm.

When a large matrix cannot be mapped in a single XB array, it should be partitioned into multiple small matrices, and then we map them into multiple XB arrays. To achieve high computing parallelism and resource utilization, we use the BTBP algorithm (lines 13–19 of Algorithm 1) to solve this problem. We use an example to illustrate the mapping strategy in Fig. 5. Assume there is a list of XB arrays with different remaining capacities, as shown in Fig. 5(a), we first pack the top-left elements of the original matrix into XB0

Algorithm 1: BTBP-Based Mapping Algorithm

Input: The original matrix M_0; The remaining capacities $XB[0, \ldots, N-1]$ in ascending order
Output: The available capacities $XB[0, \ldots, N-1]$

```
1  Initiate stack S;
2  Push M_0 into S;
3  while S is not empty and XB[N − 1] != 0 do
4      Popup a matrix M from S;
5      for i = 0 to N-1 do
6          if XB[i] is larger than M then
7              Map M to the top-left corner of XB_i;
8              Update XB[i];
9              Sort XB;
10             Break;
11         end
12     end
13     if M is not mapped then
14         Map the top-left elements of M to XB_(N−1);
15         Update XB[N − 1];
16         Sort XB;
17         Partition the remaining M into M_1 and M_2;
18         Push M_1 and M_2 into S;
19     end
20 end
```

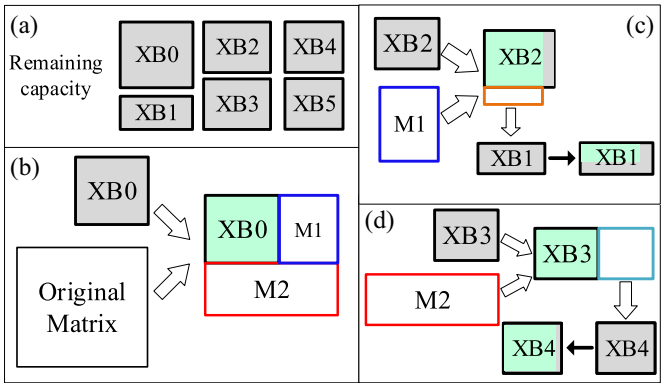


Fig. 5. Mapping a large matrix into multiple XB arrays based on the BTBP algorithm. (a) The remaining capacity of each XB array. (b) Map the original matrix into XB0, and the unmapped portion are partitioned into two matrices (M1 and M2). (c) Map M1 into XB2 and XB1. (d) Map M2 into XB3 and XB4.

with the maximum remaining capacity. Then, the remaining portion of the original matrix can be partitioned into one or two smaller matrices (e.g., M1 and M2), as shown in Fig. 5(b). If the partitioned matrix is smaller than the maximum size of the remaining XB arrays, we use the best fit algorithm (lines 5–12 of Algorithm 1) to map it. Otherwise, we repeat the BTBP procedure (lines 13–19 of Algorithm 1) for all partitioned matrices till they are all mapped into XB arrays, as shown in Fig. 5(c) and (d).

Since a large matrix may be partitioned into multiple smaller matrices that are mapped into different XB arrays, we have to combine these partial outputs to get the final result when performing an analog MVM. As a binary tree is used to organize the partitioned matrices, we use a postorder traversal algorithm

to combine the partial results. Taking Fig. 5(b) as an example, we regard the result calculated by XB0 as the current node of the tree, and the partial results of M1 and M2 as the left subtree and the right subtree, respectively. First, we coalesce the partial results obtained from the left subtree (M1) and XB0 and get a larger vector. Second, we add this vector to the partial result obtained from the right subtree (M2) and get the final result.

C. Supporting Floating-Point MVM Operations

General matrix–matrix multiplication (GEMM) uses floating-point numbers instead of fixed-point numbers, however, existing MBA simulators have not yet supported floating-point MVM operations because they are only designed for domain-specific applications using fixed-point numbers. To evaluate the performance benefit of using MBAs for more general applications, we propose an effective design in MBAs to support floating-point arithmetic according to the IEEE-754 standard. First, to map the matrix of floating-point numbers into XB arrays, MBA should align the exponents of operands according to the maximum exponent in the matrix. Second, since a single memristor can only represent a number with limited precision (usually 2 bits), MBA splits the aligned mantissas in the matrix and maps them into multiple XB arrays to process high-precision mantissas. Third, XB arrays perform the MVM operation in the analog domain. Finally, MBA coalesces the intermediate results with the assistant of S&A units and ORs. The detailed scheme is described in the following.

1) Matrix Preprocessing: An IEEE-754 single-precision floating-point number contains a sign bit, an 8-bit biased exponent, and a 23-bit mantissa (with implied leading 1). As XB arrays perform multiplications and accumulations through WLs and BLs, it is necessary to align the mantissas in the same BLs before the data mapping. For example, the mantissas of 2 and 4 are both 800000H, while their exponents are 80H and 81H, respectively. Before the accumulation, the exponent of 2 should be aligned to 81H by shifting the mantissa right by 1 bit. Then, the two mantissas can add directly.

Exponent alignment has to find the maximum exponent of elements in the target matrix. A simple approach is to scan all the exponents of elements in the matrix. However, scanning all the elements is usually costly. Feinberg *et al.* [11] exploited the locality of exponent range to reduce the alignment overhead of double-precision floating-point values. In our design, we follow Feinberg's approach and predefine a rather large exponent for the mantissa alignment. As shown in Fig. 6(a), we predefine the large exponent to be "11," and thus the input vector is aligned according to the exponent 11 rather than the maximum exponent "10" in the vector. On the other hand, we can update the predefine exponent in the matrix during aligning operands in the matrix incidentally. If an exponent in the matrix is larger than the predefined exponent, we update the predefined exponent to keep it as the largest.

We use an ALU in each tile to preprocess source operands as follows. Given the maximum exponent $E_{\max}$, for each operand in the vector, the ALU calculates the difference of exponent

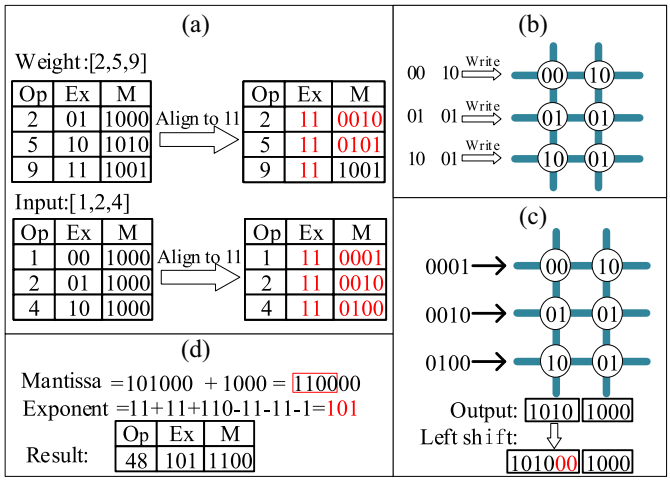


Fig. 6. Example of performing floating-point analog MVM. (a) Alignment. (b) Mapping. (c) Analog MVM. (d) Composing.

(ΔE_i) and shifts the mantissa right by ΔE_i bits. The aligned mantissas are sent to IRs of PEs for mapping. As XB arrays can only store unsigned mantissas, we employ positive XB arrays (XB+) and negative XB arrays (XB−) [4] to handle operands with different signs. The positive operands and negative operands are mapped into XB+ and XB− arrays, respectively.

We note that our design does not support not a numbers (NaNs) and infinity defined by the IEEE-754 standard. Once NaNs or infinity are detected by the ALU during the data mapping, the MVM operation is interrupted and is sent to the CPU for further processing.

2) Mantissas Mapping: We use multiple memristor cells to represent multibit operands based on bit slicing [5]. The mantissas in the IRs are split according to the precision of memristor cells. Then, multiple slices of a mantissa are written to adjacent cells in the same row to share the ADC and S&A. As shown in Fig. 6(b), the weight element "0010" is split to "00" and "10," and they are placed in two adjacent memristor cells.

NeuroSim [21] provides a rowwise write scheme that can update the conductance states of memristor cells in an XB array row by row. This approach may result in a high miss rate of the data buffer. When a matrix-vector product ($M \times V$) is performed, the columns of the matrix should be fetched from the main memory and map into the XB array row by row. Since these operands in each column of the matrix are usually not in a contiguous address space, resulting in a high miss rate of the data buffer. In contrast, performing a vector-matrix product ($V \times M$) does not lead to the above problem since the operands in each row of the matrix are contiguous in the main memory and are written to the same row of the XB array. On the other hand, when we multiply two matrices ($M_1 \times M_2$) in an MBA, one matrix is split into multiple vectors which are used as the input vectors. If the split matrix is the latter one (M_2), each element in a vector is not contiguous in the memory space since the latter matrix is processed column by column. Considering these two observations, for the matrix multiplication ($M_1 \times M_2$), we can reduce the miss rate of the

TABLE I
INSTRUCTIONS OF MBA

Type		Syntax	Description
Data Transfer	**map**	**map** dst, src, width	Map the weight from the register (src) to the XB array (dst) with (width) operands
	set	**set** dst, imm	Set the register (dst) with an immediate value (imm)
	load	**load** dst, src, ld_width	Load the data from memory (src) with (ld_width) data to buffer (dst)
	store	**store** dst, src, st_width	Store the output vector (src) with (st_width) data to memory (dst)
Computational	**mvm**	**mvm** dst, src, v_width, w_width	Perform an analog MVM with the input vector (src, v_width) and the weight matrix (dst, w_width)
	alu	**aluop** dst, src1, src2	Scalar arithmetic (add/sub/multiply/divide)
	alui	**aluop** dst, src1, imm	Scalar arithmetic with an immediate value (imm)

buffer both in the mapping phase and the computing phase if we map the latter matrix (M_2) to XB arrays. Therefore, if the matrix M_1 must be mapped as the weight matrix, we transform $M_1 \times M_2$ to $(M_2^T \times M_1^T)^T$ and map M_1^T into XB arrays. Finally, we transpose the output matrix.

3) Matrix-Vector Multiplication: Since input vectors are also floating-point numbers, these operands also should be preprocessed using the same approach for matrix preprocessing. We store the aligned mantissas and record the sign bit in registers. After the exponent alignment, the mantissas of the input vector are applied to the XB array as voltages by WLs drivers. The ADC converts the analog result to a digital result and sends it to the S&A, which shifts the digital result left by a number of bits according to the order of input mantissas and the position of cells, as shown in Fig. 6(c). In the following, the S&A adds up all the intermediate results, as shown in Fig. 6(d). To correctly handle signed operands, we perform MVM operations in the XB+ and the XB−, respectively. First, we apply voltages to the XB+ and XB− using the positive and negative mantissas, respectively. The output intermediate results of the two MVM operations are all positive. Second, we swap the positive mantissas with the negative mantissas and use them as input voltages for the XB+ and the XB− again to obtain the negative intermediate results.

4) Intermediate Result Composing: We subtract the negative intermediate result from the positive intermediate result by a SUB in each PE to get the sign bit and the absolute value of mantissa for each dot-product result. Then, we find the position of the most significant bit E_{MSB} in the mantissas via a bit scan reverse (BSR) instruction. We use E_{dec}, E_{bias}, E_{mm}, and E_{mv} to represent the length of decimal, the biased exponent, the maximum biased exponent in the matrix, and the input vector, respectively. In the IEEE-754 standard for single-precision floating-point numbers, the E_{dec} and E_{bias} are 23 and 127, respectively. The biased exponent of the final result E can be calculated as (2). The decimal of the final result can be calculated by shifting the mantissas right by $(E_{\text{MSB}} - E_{\text{dec}} - 1)$ bits and hiding the leading *1*. As shown in Fig. 6(d), the E_{dec}, E_{bias}, E_{mm}, E_{mv}, and E_{MSB} are 11, 0, 11, 11, and 110, respectively. After the decimal normalization, the exponent and the mantissa of the dot-product result are 0x101 and 1.100 (in the IEEE-754 format, the leading 1 is hided)

$$E = E_{mm} + E_{mv} + E_{\text{MSB}} - 2E_{\text{dec}} - E_{\text{bias}} - 1. \tag{2}$$

We note that the above approach may generate NaNs or infinity defined by the IEEE-754 standard for floating-point numbers. In our work, the NaNs and infinities are retained since they may be caused by the nonideal properties of memristors. These exceptions should be considered by MBA designers for further processing.

D. Instructions

We provide primitive instructions to program MBAs based on a previous work [37]. As shown in Table I, we define a *map* instruction to enable the data mapping operation. The parameters *dst*, *src*, and *width* in the *map* instruction indicate the destination address of the first memristor device in the XB array, the index of the first register that stores the operands' mantissas, and the number of mantissas mapping to the row of XB arrays, respectively. The *mvm* instruction perform a series of operations, including ADC, shifting, addition, and subtraction mentioned in Section III-C. Similar to *map*, the parameters of *mvm* contain the location of the weight matrix in the XB array, the first register that stores the input vector, the length of the input vector (rows), and the number of the columns in the mapped matrix.

Recently, a few proposals exploit static/dynamic compilation tools to improve instruction-level parallelism for programs in PIM architectures [38], [39]. These techniques cannot be employed directly because our MBA is mainly designed to accelerate analog MVM operations at the function level. However, inspired by these automatic parallelization and hardware/software co-design techniques, our simulation framework has a potential to further improve the computation parallelism for multiple MVM operations in the MBA. For example, we can also analyze the data dependency between MVMs through static/dynamic compiling analysis and then perform independent MVMs in different XB arrays in parallel. For MVMs with data dependency, we can map them in adjacent XB arrays/PEs/Tiles to reduce the communication cost.

E. Instruction Pipeline

ISAAC [5] and PipeLayer [1] design pipelines for running specific NN applications in MBAs. However, these pipelines are coarse grained because they are scheduled at the granularity of NN layers. Inspired by these proposals, we further design a fine-grained pipelining mechanism at the instruction level. The proposed pipeline can hide the latency of loading vectors by overlapping *load* instructions with other instructions, as shown in Fig. 7. Our instruction-level pipelines can utilize the resource of XB arrays efficiently. We note that analog

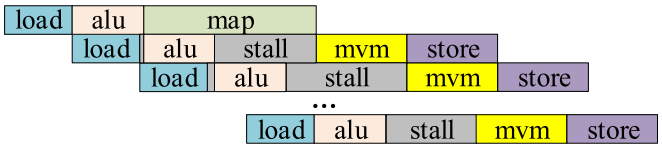


Fig. 7. Instruction pipeline in MHSim.

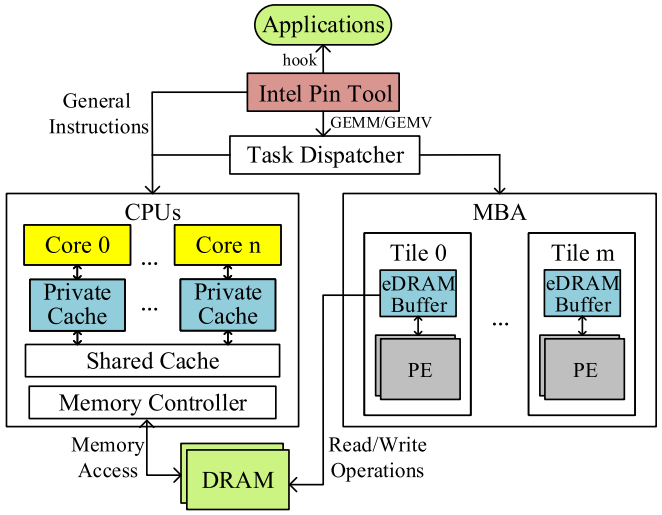


Fig. 8. Simulation framework of MHSim.

MVMs should be performed after the matrix mapping. Since XB arrays perform *mvms* sequentially, the following *mvm* may stall if the previous *mvm* has not completed.

IV. Simulation Framework

In this section, we introduce the simulation framework that supports running diverse applications in a memristor-based heterogeneous architecture, without modifying applications' source codes.

A. Overview

To evaluate the performance speedup and the energy efficiency for applications when they are going to deploy in heterogeneous computing architectures with MBAs, a heterogeneous simulation framework that supports different applications is required. To achieve this goal, we design a general-purpose MBA simulator based on NeuroSim [21] and then integrate it with ZSim [23] to simulate a heterogeneous computing system. ZSim is a cycle-accurate architectural simulator that provides processor, cache, and memory modules. NeuroSim is a circuit-level MBA simulator which provides circuit models of XB arrays and electrical properties of memristors. Fig. 8 shows the framework of MHSim. MHSim designs circuit-level modules of MBAs using NeuroSim and integrates the MBA modules with the architectural simulator ZSim. In addition, we also provide high-level APIs to simulate the analog MVM.

B. Key Modules

Task Dispatcher: In CPU–GPU heterogeneous computing systems, computation tasks are dispatched by programmers and compiled to binary executable files with CUDA. Due to the lack of a mature compiler for MBA heterogeneous architectures, it is hard to determine whether a matrix multiplication should be processed by MBAs. Furthermore, small matrices have low parallelism, and the performance running in XB arrays may be even worse than that of running in CMOS-based processors. In MHSim, we use a task dispatcher to determine whether the task should be executed in an MBA or a CMOS-based processor. Since the latency of analog computing for different MVM operations in XB arrays is the same regardless of the matrix sizes, the total execution time of an MVM operation (T_{XB}) is mainly determined by the latency of ADC operations. To simplify the estimation of T_{XB}, we assume that only one ADC is multiplexed by all columns of an XB array to read each element of the output vector sequentially. Thus, the total execution time of an MVM is mainly determined by the number of columns N_c that are used to map the weight matrix in the XB array. Then, T_{XB} can be estimated by N_c/F_{ADC}, where F_{ADC} denotes the frequency of the ADC. In our simulator, we set N_c as the total number of columns in an XB array to estimate the maximum execution time of an analog MVM. To estimate the latency of MVMs in CMOS-based processors, the task dispatcher uses a performance model to estimate the execution time based on the matrix size and the instruction cycles [40]. Assume K, M, T_m, and T_a represent the rows of the matrix, the columns of the matrix, the execution time of a floating-point multiplication operation, and the execution time of an addition operation, respectively. The total execution time T_{CMOS} of an MVM operation in CMOS-based processors is estimated by (3). If $T_{XB} < T_{\text{CMOS}}$, the MVM task is assigned to the MBA for higher performance

$$T_{\text{CMOS}} = (K-1) * M * T_a + K * M * T_m. \tag{3}$$

Extensions for ZSim: ZSim is a DBI-based simulator which inserts simulation codes between instructions. However, it is difficult to recognize MVM operations from code snippets. An effective approach to recognize MVM operations is to find out MVMs based on the functions' names or the programming interfaces. However, ZSim does not support function-level instrumentation. Thus, we complement the function-level instrumentation for ZSim to trace MVM functions and orchestrate it with instruction-level instrumentation to simulate instructions cautiously.

Extensions for NeuroSim: As discussed in Section II-C, NeuroSim is designed for a specific neural network algorithm, and the peripheral circuits are fixed. We redesign the structure of the peripheral circuit in NeuroSim to support different sizes of matrices and floating-point MVM operations. We also provide APIs to simulate analog MVMs for arbitrary sizes with our reconfigurable peripheral circuit models and device models provided by NeuroSim. These APIs can be easily integrated into other applications or simulators.

C. Integration of CPU and MBA Simulations

As shown in Fig. 8, we employ DBI to track the specific functions such as MVMs through Intel Pin tool [41], which provides function-level instrumentation. Originally, the

TABLE II
PARAMETERS OF MEMRISTOR DEVICES

Devices	Nonlinearity (SET/RESET)	# of States	R_{on}	CtoC Variation
Ag:a-Si [43]	2.40/-4.88	97	26 MΩ	3.50%
TaOx/HfOx [42]	0.04/-0.63	128	100 KΩ	3.70%

Pin tool tracks the instructions of MVM operations and simulates them in the CPU simulator. In MHSim, we replace the simulation of MVM operations in CPU with equivalent operations simulated in MBAs.

As MBA mainly accelerates MVM operations, we track each MVM function via a uniform interface. It is well known that basic linear algebra subprograms (BLAS) provide a standard interface to perform common linear algebra operations, such as dot products and matrix multiplication. The specification contains GEMM and general MVM (GEMV) functions. As shown in Fig. 8, we hook applications and track the GEMM and GEMV functions and then deliver them to the task dispatcher.

We use ZSim to simulate CPUs and the memory hierarchy and exploit NeuroSim to simulate the MBA. Once GEMM/GEMV functions are assigned to the MBA, MHSim adds instrumentation for them and skips the simulation of these functions in ZSim. Thus, applications with GEMM/GEMV functions will call the MBA module simulated by NeuroSim without any modification of source codes.

V. EVALUATION

We first validate our simulation framework with SPICE and then conduct case studies to evaluate the inference accuracy and performance of MBAs.

A. Experimental Setup

System Configurations: Table II shows the parameters of two memristor devices. The number of states is the available conductance states, i.e., the number of values that one memristor cell can represent. The CtoC variation refers to the cycle-to-cycle variation during write operations. More details on these memristor devices could be found in [42]. Table III shows the system setup about the configuration of CPU, main memory, the MBA, and the GPU. We note that a memristor cell only represents a 2-bit number, and a number of cells should be used to represent a single high-precision number via bit slicing [5]. In our case studies, we choose the memristor device TaOx/HfOx [42], which shows excellent linearity for both SET and RESET operations.

Benchmarks: In our case studies, we use typical neural network models, such as MLP [47], LeNet [44], AlexNet [48], VGG16 [49], ResNet50 [50], and long short-term memory (LSTM) [51] for different datasets, and run them in Caffe [52]. Table IV depicts the detail of neural network models and datasets, including the number of convolutional (Conv) layers and fully connected (FC) layers, and LSTM layers.

TABLE III
SYSTEM SETUP

CPU and Main Memory	
Processor	4 cores; 3GHz; Out-of-order
L1I&L1D cache	4 KB; 4-way; 2 cycles access
L2 cache	2 MB; 8-way; 10 cycles access
LLC	12 MB; 16-way; 27 cycles access
DRAM	100 cycles access
Memristor-based Accelerator	
eDRAM buffer	1 MB
ADC	8 bit resolution; 1.2 GHz; 16 per PE
Memristor device	TaOx/HfOx [42]
XB array	128×128; 2 bits per cell; 16 per PE
PE	16 per tile
Tile	128 per chip
Chip	8
GPU Platform	
On-chip memory	16 GB HBM2
# CUDA cores	3584
Graphic card	NVIDIA Tesla P100 (Pascal)

TABLE IV
NEURAL NETWORK MODELS

Dataset	Network	# Conv Layers	# FC Layers	# LSTM Layers
MNIST [44]	MLP	-	3	-
	LeNet	2	2	-
ILSVRC 2012 [45]	AlexNet	5	3	-
	VGG-16	13	3	-
	ResNet-50	53	1	-
COCO 2014 [46]	LSTM	-	1	2

B. Validation of Floating-Point MVMs

Because circuit/device models have been validated in NeuroSim, we only validate the accuracy of the floating-point MVM in our MBA simulator. We follow the circuit/device parameter settings in SPICE for validation. We use ADCs with 10-MHz frequency and 0.31-mW power consumption. Each ADC is shared with 16 columns of an XB array by a MUX. To mitigate the impact of DACs on computation accuracy, we do not use DACs to convert digital inputs into analog voltages. Instead, we use 24-cycle pulses of voltages to represent 24-bit mantissas of the input vector [21]. Since different values of memristor cells have an impact on the computing latency and energy consumption, we set the initial value of all memristor cells to high conductance (i.e., one) for all tests. We conduct our experiments in different XB arrays whose sizes increase from 16 × 16 to 256 × 256. We measure the latency and energy consumption of one floating-point MVM operation and compare experimental results in MHSim with the state-of-the-art MNSIM and PUMAsim. We also use SPICE as the baseline.

Fig. 9 shows the energy consumption and latency of one floating-point MVM operation performed by an XB array with different sizes. For each kind of XB arrays, the difference of energy and latency between MHSim and SPICE is trivial. Overall, the mean absolute percentage errors (MAPEs) of energy and latency for different sizes of XB arrays are only 0.47% and 0.49%, respectively. The errors of MHSim are mainly introduced by clock signals and digital supplies which are not simulated in SPICE. In contrast, the MAPEs

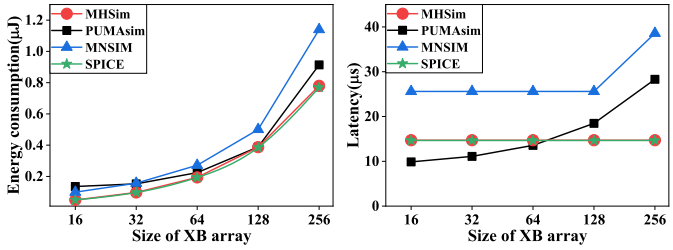


Fig. 9. Energy consumption and latency of one floating-point MVM measured in different simulators.

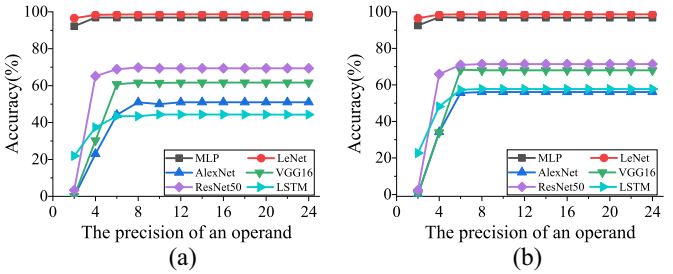


Fig. 10. Inference accuracy of neural network models using (a) Ag:a-Si and (b) TaOx/HfOx devices.

of energy and latency in PUMAsim are 53.49% and 36.73%, respectively. MNSIM shows even higher MAPEs for energy and latency estimations. For PUMAsim and MNSIM, the significant latency and energy deviations relative to SPICE mainly stem from different circuit models, such as the frequency and resolution of ADC. Despite the deviation of latency, MNSIM and PUMAsim show a similar tendency of energy consumption with MHSim and SPICE. This implies that MNSIM and PUMAsim can achieve similar results using the same circuit/device models/parameters. These experimental results demonstrate that MHSim is able to accurately simulate floating-point MVM operations in XB arrays.

C. Inference Accuracy

Fig. 10 shows how the precision of memristor cells affects the inference accuracy of different neural networks. The inference accuracy reflects the accuracy of trained neural networks for the image-recognition and natural language processing applications. When multiple memristor cells are used to support full precision (24-bit mantissas), the inference accuracy of MLP, LeNet, AlexNet, VGG16, ResNet50, and LSTM using the Ag:a-Si device are 96.9%, 98.6%, 46.1%, 61.7%, 69.4%, and 44.3%, respectively. Using TaOx/HfOx device, the inference accuracy of these neural networks are 96.8%, 98.6%, 56.1%, 68%, 71.4%, and 57.8%, respectively. Ag:a-Si leads to lower inference accuracy compared with TaOx/HfOx since the higher nonlinearity of Ag:a-Si causes much more computation errors in MVM operations.

Fig. 11 shows the accuracy degradation of these neural networks when the MBA uses Ag:a-Si and TaOx/HfOx as the memristor device. The accuracy degradation refers to the difference of the inference accuracy when the same neural network model is executed in the MBA and the CPU. When we use Ag:a-Si as the memristor device with full

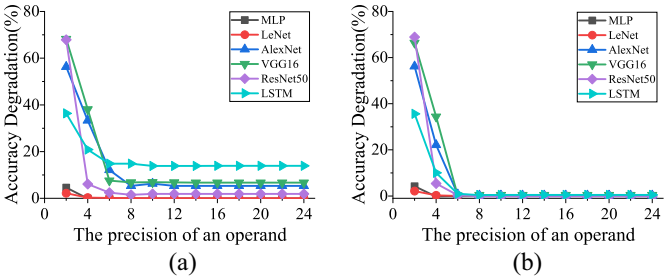


Fig. 11. Accuracy degradation of neural network models using (a) Ag:a-Si and (b) TaOx/HfOx devices.

precision, the accuracy degradation is 0.1%, 5.3%, 6.7%, 1.9%, and 13.9% for LeNet, AlexNet, VGG16, ResNet50, and LSTM, respectively. For MLP, the MBA even achieves 0.1% accuracy improvement. This implies that the inference accuracy of AlexNet, VGG16, and LSTM is more sensitive to the computation accuracy. However, the accuracy degradation approximates zero for these neural networks when we employ TaOx/HfOx as the memristor device because TaOx/HfOx shows excellent linearity for both SET and RESET operations. These experimental results demonstrate that MHSim can effectively simulate the computation accuracy of analog MVMs with different device models.

We also find that almost all neural networks achieve the maximum accuracy when the precision of operands increases to 6 bits. This implies that it is unnecessary to perform analog MVM operations with full-precision floating-point operands for these applications. This is because the least significant bits have a little impact on the final result since the most significant bits may lose computing precision due to the nonideal properties of memristor devices. In addition, the bit slicing for a small exponent may also lose computing precision during the exponent alignment. These experimental results demonstrate that MHSim can be used to determine how many memristor cells are needed to represent operands for a given application while still guaranteeing adequate computation accuracy.

D. Performance of MBA

In this section, we evaluate the performance speedup of the MBA for typical neural networks. We use eight 2-bit memristor devices to represent a weight operand, and 16-cycle voltage pulses to represent an input operand. To fully utilize XB arrays, we map weight matrices into XB arrays in advance based on the weight replication technique [5]. We map duplicate weights into XB arrays according to their available capacities based on the BTBP algorithm. ALUs are also used to process rectified linear unit (ReLU) and max pooling for tiles. We note that the interlayer pipeline proposed in ISAAC [5] is not applicable since our MBA architecture does not contain dedicated hardware for normalization or activation operations.

Fig. 12 shows the normalized performance of MVM operations using CPU, GPU, and MBA. MBA achieves 83.9×, 20.8×, 398.9×, 145.2×, 70.7×, and 588.4× performance speedup for MLP, LeNet, AlexNet, VGG16, ResNet50, and LSTM, respectively. The significant speedup mainly stems

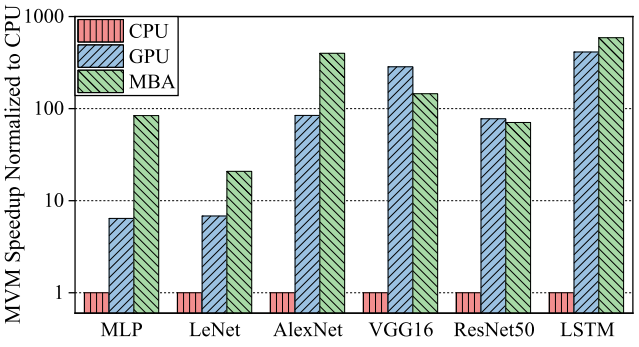


Fig. 12. Performance speedup of MVM operations.

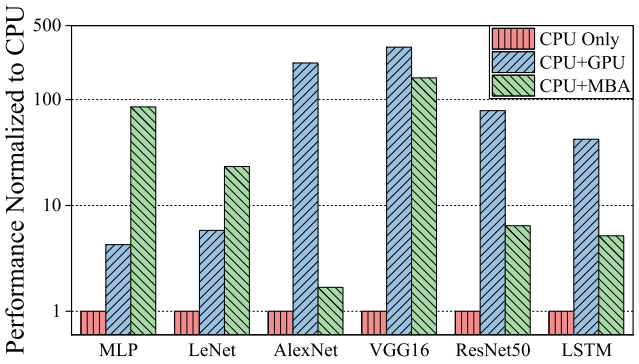


Fig. 13. Performance speedup of different neural networks in different architectures.

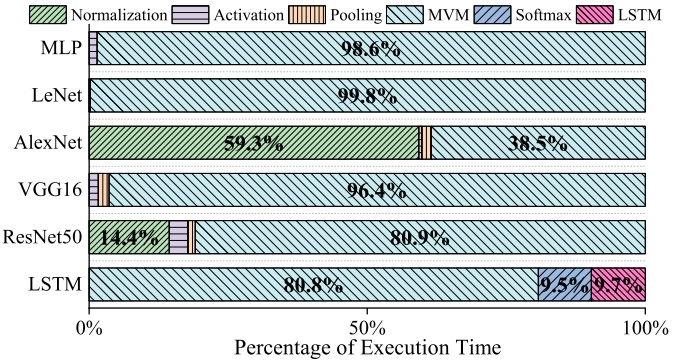


Fig. 14. Breakdown of the execution time for different neural networks in the CPU-only architecture.

from the high parallelism of XB arrays. For VGG16 and ResNet50, we find that the speedups of the MBA are less than that of GPUs because the finite memristor resource limits the number of weight replications due to massive weight parameters. For the first layer of VGG16, we can only replicate 512 weights for total 50 176 input vectors. We note that the number of weight replications cannot fit the maximum input vectors for most benchmarks except MLP and LeNet. Despite the limited number of weight replications, the MBA achieves better performances of MVM operations than GPU for MLP, LeNet, AlexNet, and LSTM.

Fig. 13 shows the performance speedup of "CPU + GPU" and CPU + MBA architectures, all relative to the "CPU-only." To figure out the root cause of the speedup, Fig. 14 also elaborates the breakdown of the execution time for different neural networks when they run in the CPU-only architecture.

As shown in Fig. 13, for MLP and LeNet, CPU + MBA achieves 84.2× and 22.4× speedups, while CPU+GPU achieves less than 7× speedups compared with CPU-only. The MBA achieves significant speedups because MVMs spend about 99% of the total execution time, as shown in Fig. 14. On the other hand, since the weight matrix is small, the MBA can accommodate sufficient weight replications to improve the parallelism.

For AlexNet and ResNet50, CPU + MBA only achieves 1.7× and 6.4× speedups compared with CPU-only. The reason is that local response normalization (LRN) layers and batch normalization (BN) layers cannot be accelerated by memristor XB arrays [5]. As shown in Fig. 14, LRN layers and BN layers consume 59% and 14% of the total execution time for AlexNet and ResNet50, respectively. Although Fig. 12 shows the execution time of MVM operations is 398× and 70× folded for AlexNet and ResNet, the total speedups of these neural networks are limited because the normalization layers become the performance bottleneck of these neural networks after MVMs are accelerated by the MBA. CPU + GPU achieves 220× and 70× speedups for AlexNet and ResNet50 because all layers can be processed by GPUs effectively in Caffe [52].

For VGG16, both CPU + MBA and CPU + GPU achieve the maximum speedups among those neural networks. Although MVMs are the dominant operation in VGG16, the huge weight matrix limits the number of replicate weights due to finite XB arrays. For LSTM, CPU + MBA and CPU + GPU achieve 3.7× and 37.1× speedups, respectively. Although MBA achieves higher performance speedup than GPU for MVM operations solely, as shown in Fig. 12, the performance speedup for LSTM is limited to softmax functions and LSTM layers (approximate 20% of total execution time). Because memristor XB arrays are not able to accelerate the sigmoid, *hyperbolic tangent* (tanh), and power functions in the LSTM layers and softmax layers, they become the performance bottleneck in the CPU + MBA architecture. In contrast, because these layers are optimized using CUDA and GPU, CPU + GPU achieves higher performance compared with CPU + MBA.

Overall, these case studies demonstrate that MHSim is able to simulate a heterogeneous computing system composed of CPUs and MBAs. It can evaluate the inference accuracy of neural networks when they use the MBA to accelerate MVM operations. MHSim can be also used to profile neural network applications in a heterogeneous computing architecture and estimate the performance speedup of MBAs.

VI. Conclusion

In this article, we develop MHSim, a simulation framework for memristor-based heterogeneous computing architectures. MHSim supports floating-point MVM operations to extend the adoption of MBAs for various real-world applications. MHSim can simulate diverse applications implemented with GEMM/GEMV functions via DBI. Experimental results demonstrate that MHSim can evaluate the performance

speedup of various NN applications and the inference accuracy in MBA computing architectures.

REFERENCES

[1] L. Song, X. Qian, H. Li, and Y. Chen, "PipeLayer: A pipelined ReRAM-based accelerator for deep learning," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, 2017, pp. 541–552.

[2] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "GraphR: Accelerating graph processing using ReRAM," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, 2018, pp. 531–543.

[3] C. Yang, L. Zheng, C. Gui, and H. Jin, "Efficient FPGA-based graph processing with hybrid pull-push computational model," *Front. Comput. Sci.*, vol. 14, no. 4, pp. 1–16, Jan. 2020.

[4] P. Chi *et al.*, "PRIME: A novel processing-in-memory architecture for neural network computation in ReRAM-based main memory," in *Proc. 43rd Int. Symp. Comput. Archit. (ISCA)*, 2016, pp. 27–39.

[5] A. Shafiee *et al.*, "ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *Proc. 43rd Int. Symp. Comput. Archit. (ISCA)*, 2016, pp. 14–26.

[6] J. Zhao, "Exponential stabilization of memristor-based neural networks with unbounded time-varying delays," *Sci. China Inf. Sci.*, vol. 64, no. 8, pp. 1–3, Jun. 2021.

[7] E. Zhou, L. Fang, R. Liu, and Z. Tang, "Area-efficient memristor spiking neural networks and supervised learning method," *Sci. China Inf. Sci.*, vol. 62, no. 9, pp. 1–3, Jul. 2019.

[8] V. Joshi *et al.*, "Accurate deep neural network inference using computational phase-change memory," *Nat. Commun.*, vol. 11, no. 1, p. 2473, May 2020.

[9] Y. Lin *et al.*, "A novel voltage-accumulation vector-matrix multiplication architecture using resistor-shunted floating gate flash memory device for low-power and high-density neural network applications," in *Proc. IEEE Int. Electron Devices Meeting (IEDM)*, 2018, pp. 2.4.1–2.4.4.

[10] M. Cai and H. Huang, "A survey of operating system support for persistent memory," *Front. Comput. Sci.*, vol. 15, no. 4, pp. 1–20, Feb. 2021.

[11] B. Feinberg, U. Vengalam, N. Whitehair, S. Wang, and E. Ipek, "Enabling scientific computing on memristive accelerators," in *Proc. 45th Annu. Int. Symp. Comput. Archit. (ISCA)*, 2018, pp. 367–382.

[12] H. Mao, M. Song, T. Li, Y. Dai, and J. Shu, "LerGAN: A zero-free, low data movement and PIM-based GAN architecture," in *Proc. 51st Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, 2018, pp. 669–681.

[13] A. Ankit *et al.*, "PUMA: A programmable ultra-efficient memristor-based accelerator for machine learning inference," in *Proc. 24th Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, 2019, pp. 715–731.

[14] N. Challapalle *et al.*, "GaaS-X: Graph analytics accelerator supporting sparse data representation using crossbar architectures," in *Proc. ACM/IEEE 47th Annu. Int. Symp. Comput. Archit. (ISCA)*, 2020, pp. 433–445.

[15] Y. Huang, L. Zheng, X. Liao, H. Jin, P. Yao, and C. Gui, "RAGra: Leveraging monolithic 3D ReRAM for massively-parallel graph processing," in *Proc. 22nd Conf. Design Autom. Test Europe (DATE)*, 2019, pp. 1273–1276.

[16] M. Lin *et al.*, "DL-RSIM: A simulation framework to enable reliable ReRAM-based accelerators for deep learning," in *Proc. Int. Conf. Comput. Aided Design (ICCAD)*, 2018, pp. 1–8.

[17] S. Jain, A. Sengupta, K. Roy, and A. Raghunathan, "RxNN: A framework for evaluating deep neural networks on resistive crossbars," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 40, no. 2, pp. 326–338, Feb. 2021.

[18] M. Rasch *et al.*, "A flexible and fast PyTorch toolkit for simulating training and inference on analog crossbar arrays," in *Proc. 3rd Int. Conf. Artif. Intell. Circuits Syst. (AICAS)*, 2021, pp. 1–4.

[19] C. Lammie and M. Azghadi, "MemTorch: A simulation framework for deep memristive cross-bar architectures," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, 2020, pp. 1–5.

[20] L. Xia *et al.*, "MNSIM: Simulation platform for memristor-based neuromorphic computing system," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 5, pp. 1009–1022, May 2018.

[21] P. Chen, X. Peng, and S. Yu, "NeuroSim: A circuit-level macro model for benchmarking neuro-inspired architectures in online learning," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 12, pp. 3067–3080, Dec. 2018.

[22] N. Binkert *et al.*, "The Gem5 simulator," *ACM SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, May 2011.

[23] D. Sanchez and C. Kozyrakis, "ZSim: Fast and accurate microarchitectural simulation of thousand-core systems," in *Proc. 40th Annu. Int. Symp. Comput. Archit. (ISCA)*, 2013, pp. 475–486.

[24] "MHSim." [Online]. Available: https://github.com/CGCL-codes/MHSim (accessed Jan. 18, 2022).

[25] M. Cheng *et al.*, "TIME: A training-in-memory architecture for memristor-based deep neural networks," in *Proc. 54th Annu. Design Autom. Conf. (DAC)*, 2017, pp. 1–6.

[26] M. Prezioso, F. Merrikh-Bayat, B. Hoskins, G. Adam, K. Likharev, and D. Strukov, "Training and operation of an integrated neuromorphic network based on metal-oxide memristors," *Nature*, vol. 521, no. 7550, pp. 61–64, May 2015.

[27] F. Merrikh-Bayat, M. Prezioso, B. Chakrabarti, H. Nili, I. Kataeva, and D. Strukov, "Implementation of multilayer perceptron network with highly uniform passive memristive crossbar circuits," *Nat. Commun.*, vol. 9, no. 1, p. 2331, Jun. 2018.

[28] P. Yao *et al.*, "Face classification using electronic synapses," *Nat. Commun.*, vol. 8, no. 1, May 2017, Art. no. 15199.

[29] F. Chen, L. Song, and Y. Chen, "ReGAN: A pipelined ReRAM-based accelerator for generative adversarial networks," in *Proc. 23rd Asia South Pacific Design Autom. Conf. (ASP-DAC)*, 2018, pp. 178–183.

[30] F. Chen, L. Song, H. Li, and Y. Chen, "ZARA: A novel zero-free dataflow accelerator for generative adversarial networks in 3D ReRAM," in *Proc. 56th Annu. Design Autom. Conf. (DAC)*, 2019, pp. 1–6.

[31] N. Challapalle *et al.*, "PSB-RNN: A processing-in-memory systolic array architecture using block circulant matrices for recurrent neural networks," in *Proc. 23rd Conf. Design Autom. Test Europe (DATE)*, 2020, pp. 180–185.

[32] C. Nien, Y. Hsiao, H. Cheng, C. Wen, Y. Ko, and C. Lin, "GraphRSim: A joint device-algorithm reliability analysis for ReRAM-based graph processing," in *Proc. 23rd Conf. Design Autom. Test Europe (DATE)*, 2020, pp. 1478–1483.

[33] W. Fei, H. Yu, W. Zhang, and K. Yeo, "Design exploration of hybrid CMOS and memristor circuit by new modified nodal analysis," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 20, no. 6, pp. 1012–1025, Jun. 2012.

[34] G. Krishnan, S. Mandal, C. Chakrabarti, J. Seo, U. Ogras, and Y. Cao, "Interconnect-aware area and energy optimization for in-memory acceleration of DNNs," *IEEE Design Test*, vol. 37, no. 6, pp. 79–87, Dec. 2020.

[35] G. Krishnan, S. Mandal, C. Chakrabarti, J. Seo, U. Ogras, and Y. Cao, "Impact of On-chip interconnect on in-memory acceleration of deep neural networks," *ACM J. Emerg. Technol. Comput. Syst.*, vol. 18, no. 2, pp. 1–22, Apr. 2021.

[36] "Binary Tree Bin Packing Algorithm." 2011. [Online]. Available: https://codeincomplete.com/articles/bin-packing/ (accessed Sep. 10, 2021).

[37] J. Ambrosi *et al.*, "Hardware-software co-design for an analog-digital accelerator for machine learning," in *Proc. IEEE Int. Conf. Rebooting Comput. (ICRC)*, 2018, pp. 1–13.

[38] Y. Xiao, S. Nazarian, and P. Bogdan, "Prometheus: Processing-in-memory heterogeneous architecture design from a multi-layer network theoretic strategy," in *Proc. 21st Conf. Design Autom. Test Europe (DATE)*, 2018, pp. 1387–1392.

[39] Y. Xiao, S. Nazarian, and P. Bogdan, "Plasticity-on-chip design: Exploiting self-similarity for data communications," *IEEE Trans. Comput.*, vol. 70, no. 6, pp. 950–962, Jun. 2021.

[40] (Intel, Mountain View, CA, USA). *Intel 64 and IA-32 Architectures Optimization Reference Manual*. [Online]. Available: https://software.intel.com/content/dam/develop/public/us/en/documents/64-ia-32-architectures-optimization-manual.pdf (accessed Oct. 29, 2020).

[41] C.-K. Luk *et al.*, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, 2005, pp. 190–200.

[42] W. Wu *et al.*, "A methodology to improve linearity of analog RRAM for neuromorphic computing," in *Proc. IEEE Symp. VLSI Technol.*, 2018, pp. 103–104.

[43] S. Jo, T. Chang, I. Ebong, B. Bhadviya, P. Mazumder, and W. Lu, "Nanoscale memristor device as synapse in neuromorphic systems," *Nano Lett.*, vol. 10, no. 4, pp. 1297–1301, Apr. 2010.

[44] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.

[45] O. Russakovsky *et al.*, "ImageNet large scale visual recognition challenge," *Int. J. Comput. Vis.*, vol. 115, no. 3, pp. 211–252, Apr. 2015.

[46] T. Lin *et al.*, "Microsoft COCO: Common objects in context," in *Proc. Eur. Conf. Comput. Vis. (ECCV)*, 2014, pp. 740–755.

[47] F. Murtagh, "Multilayer perceptrons for classification and regression," *Neurocomputing*, vol. 2, no. 5, pp. 183–197, Jul. 1991.
[48] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. 26th Annu. Conf. Neural Inf. Process. Syst. (NIPS)*, 2012, pp. 1106–1114.
[49] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014, *arXiv:1409.1556*.
[50] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2016, pp. 770–778.
[51] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997.
[52] Y. Jia *et al.*, "Caffe: Convolutional architecture for fast feature embedding," in *Proc. 22nd ACM Int. Conf. Multimedia (MM)*, 2014, pp. 675–678.

Haikun Liu (Member, IEEE) received the Ph.D. degree in computer science and technology from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 2012.

He is currently a Professor with the School of Computer Science and Technology, HUST. He has coauthored more than 60 papers in most prestigious conferences and journals, such as SC/ASPLOS/HPCA/ICS/HPDC, and IEEE TRANSACTIONS ON COMPUTERS, IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, and *ACM Transactions on Architecture and Code Optimization*. His current research interests include in-memory computing, virtualization technologies, cloud computing, and distributed systems.

Prof. Liu is a Senior Member of CCF.

Jiahong Xu received the B.S. degree from the School of Control and Computer Engineering, North China Electric Power University, Beijing, China, in 2018. He is currently pursuing the Ph.D. degree with the School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China.

His research interests mainly include nonvolatile memory and memristor-based accelerator.

Xiaofei Liao (Member, IEEE) received the Ph.D. degree in computer science and engineering from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 2005.

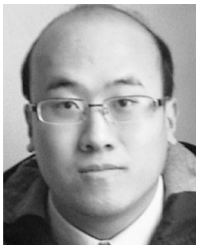
He is a Professor with the School of Computer Science and Technology, HUST. His research interests are in the areas of computer architecture, system software, and big data processing.

Prof. Liao was awarded the Excellent Youth Award from the National Science Foundation of China in 2018, and the CCF-IEEE CS Young Computer Scientist Award in 2017. He is a member of IEEE Computer Society.

Hai Jin (Fellow, IEEE) received the Ph.D. degree in computer engineering from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 1994.

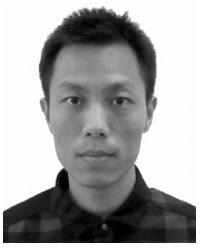
He is a Chair Professor of Computer Science and Engineering with HUST. In 1996, he was awarded a German Academic Exchange Service fellowship to visit the Technical University of Chemnitz, Chemnitz, Germany. He worked at The University of Hong Kong, Hong Kong, from 1998 to 2000, and as a Visiting Scholar with the University of Southern California, Los Angeles, CA, USA, from 1999 to 2000. He has coauthored more than 20 books and published over 900 research papers. His research interests include computer architecture, parallel and distributed computing, big data processing, data storage, and system security.

Dr. Jin was awarded the Excellent Youth Award from the National Science Foundation of China in 2001. He is a Fellow of CCF and a Life Member of ACM.



Yu Zhang (Member, IEEE) received the Ph.D. degree in computer science from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 2016.

He is currently an Associate Professor with the School of Computer Science and Technology, HUST. His current topic mainly focuses on application-driven big data processing and optimizations. His research interests include big data processing, graph computing, and distributed systems.



Fubing Mao received the Ph.D. degree from the School of Computer Science and Engineering, Nanyang Technological University, Singapore, in 2018.

He is currently an Assistant Professor with the School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China. His research interests include computer architecture, system software, and optimization algorithms.