2021 IEEE 3rd International Conference on Artificial Intelligence Circuits and Systems (AICAS) | 978-1-6654-1913-0/20/$31.00 ©2021 IEEE | DOI: 10.1109/AICAS51828.2021.9458441

MNSIM-TIME: Performance Modeling Framework for Training-In-Memory Architectures

Kaizhong Qiu, Zhenhua Zhu, Yi Cai, Hanbo Sun, Yu Wang, Huanzhong Yang
Department of Electronic Engineering, *Tsinghua University*, Beijing, China
e-mail: yu-wang@tsinghua.edu.cn

Abstract—**Emerging memristors and Processing-In-Memory (PIM) architectures have shown powerful capabilities in improving the computing energy efficiency of neural network (NN) algorithms. Existing work has proposed the memristor-based NN training architecture [2], which can improve more than 10x energy efficiency improvement compared with CMOS-based solutions. In this paper, we propose a behavior-level modeling framework for memristor-based training-in-memory architectures, called MNSIM-TIME. Compared with existing modeling tools, MNSIM-TIME supports configurable architecture design and fast hardware performance modeling, which helps researchers to realize efficient design space exploration in the early architecture design stage.**

I. Introduction

In the past few years, Convolutional Neural Networks (CNNs) have shown great success in several fields. However, as CNN models become more and more complex, the number of parameters and computations in CNN models increase dramatically, which causes CNN training to consume high energy and long computational time.

Existing work has demonstrated that emerging memristors (*e.g.*., RRAM, Resistive Random Access Memory) and memristor-based Training-In-Memory (TIM) architectures can improve the energy efficiency of CNN training by 19.6x compared with CMOS-based accelerators [2]. This remarkable improvement mainly comes from the *in-situ* Matrix-Vector-Multiplication (MVM) implementation in memristor crossbars without matrix data movements between memory and computational units, which are the main bottleneck of improving CNN training efficiency in von Neumann architectures.

To perform design space exploration efficiently and optimize the architecture design of TIM architectures, accurate simulation of hardware performance in early-stage design is essential. However, due to the huge architecture design space and the large-scale parameter set (*e.g.*, different device technologies, various architecture designs, etc.), the circuit-level simulation time of the entire TIM architectures takes an unacceptably long time. For example, the SPICE simulation of one 128×128 Crossbar takes over 2 hours, but the last layer of the VGG-19 model includes 288 crossbars. Therefore, a fast and accurate simulation tool is necessary for TIM architecture design, especially in the early design stage. TIME [2] and Long Live TIME [1] illustrate the feasibility of TIM architectures and provide some performance results of TIM. However, they mainly focus on specific architecture design and do not support performance simulation of general and configurable TIM architectures used for design space exploration. MNSIM 2.0 [15] and XPESim [14] are two simulation tools for memristor-based CNN inference accelerators. Compared with the memristor-based CNN inference accelerators, the data flow and basic operations of TIM are significantly different, which make existing simulators designed for CNN inference can not be applied to TIM architectures directly. For example, memristor-based CNN inference accelerators perform MVMs by using crossbar read operations without writing memristors (*i.e.,* weight values will not be changed during inference). While for the purpose of updating the weight matrices during CNN training, massive memristor write operations are required, which are excluded from the CNN inference simulators. NeuroSim V2.0 [10] is a circuit-level simulator for TIM architectures. It introduces device non-ideal factors into the training framework and provides accurate performance simulation results considering memristor characteristics. Nevertheless, at the architecture level, NeuroSim lacks flexibility in supporting various user-defined architecture designs. TxSim [11] is another modeling framework designed for TIM, but it only simulates training accuracy, neglecting the hardware performance modeling, *i.e.,* area, power, and latency.

In this paper, we expand MNSIM 2.0 to support the hardware performance modeling for TIM architectures, called MNSIM-TIME. Compared with circuit-/device-level simulators, MNSIM-TIME is designed for architecture and algorithm researchers to quickly evaluate the performance of user-defined architectures for efficient design space exploration. The main contributions of this paper are as follows.

(1) We propose a hierarchical modeling structure for TIM architectures, which provides multiple computing units configuration and connection options to describe diversified TIM architectures. According to these options, users are allowed to customize their own TIM architecture designs.

(2) Based on the above modeling structure and the algorithm mapping method used in MNSIM 2.0, we propose the CNN training dataflow analysis for TIM architectures, which considers the data dependency of the training procedure and maximizes the throughput under hardware resource constraints.

(3) To reduce the circuit simulation time of TIM architectures, we propose the hardware performance modeling flow. It provides a performance model for basic computing units according to circuit simulation and synthesis results. Thus, MNSIM-TIME can evaluate architecture performance under different hardware configurations and CNN models within a short time for further design space optimization.

II. Preliminaries

A. Back-propagation In CNNs Training

Compared with inference, CNNs training consists of the forward computation and the error back-propagation to generate weights gradients and update weights. The back-propagation of convolutional (CONV) layers contains (1) and (2),

$$\delta^{l-1} = \delta^{l} * rot180(W^{l}) \odot \sigma^{'}(z^{l-1}) \tag{1}$$

$$dW^{l} = a^{l-1} * \delta^{l} \tag{2}$$

where δ^{l}, δ^{l-1}, W^{l} represent the input of error back-propagation, output of error back-propagation, and the weight matrix of the l^{th} layer, respectively. dW^{l} is the update deviation of the original weight matrix W^{l} and a^{l-1} is the input feature map of the l^{th} CNN layer. $rot180(\cdot)$ means rotate the weight matrix by 180° and $\sigma(\cdot)$ represents the activation function. The forward computation is $a^{l-1} = \sigma(z^{l-1})$, where z^{l-1} is the input of activation function. $\odot$ is the Hadamard Product of two matrices, and $*$ represents the convolution between two matrices. Similarly, the computations of error back-propagation in fully-connected (FC) layers are listed as (3) and (4),

$$\delta^{l-1} = (W^{l})^{T}\delta^{l} \odot \sigma^{'}(z^{l-1}) \tag{3}$$

$$dW^{l} = \delta^{l} \times (a^{l-1})^{T} \tag{4}$$

B. *Memristor and Training-In-Memory Architectures*

Memristor is a kind of non-volatile memory that stores information with different resistance values. Multiple memristor cells can construct the memristor crossbar, which is an area-efficient structure combining storage and computing together. When mapping CNN models onto memristor crossbars, the weight matrix is stored in the crossbar, while the input vector is represented by a voltage vector $\boldsymbol{V}$ applied onto the word-line (WL) of crossbars. Then the MVM results are represented by the output current vector $\boldsymbol{I}$ from each bit line (BL):

$$i_{out,k} = \sum_{j=1}^{N} g_{k,j} v_{in,j} \tag{5}$$

Since MVMs are the major operations of CNN training, researchers propose memristor-based TIM architectures to improve training efficiency, *e.g.,* TIME [2] and PipeLayer [12].

III. MNSIM-TIME Overview

A. *Architecture Design*

MNSIM-TIME provides a hierarchical modeling structure and multiple choices for customizing different TIM architectures, as shown in Figure 1. The hierarchical modeling structure is composed of three levels, *i.e.,* bank level, tile level, and Processing Element (PE) level. As mentioned in Section II-A, the significant difference between CNN training and inference is the back-propagation, which contains error δ generation, weight gradients calculation, and weights update. To support these training-in-memory operations, MNSIM-TIME makes several modifications based on the CNN inference architecture abstraction used in MNSIM 2.0, *i.e.,* transpose crossbars, vector rearrangement units, hierarchical feature map storage structure, weight gradient computation units, etc.

There are two kinds of back-propagation error generation listing in (1) and (3). Firstly, to fulfill δ^{l-1} generation in FC layers, we use transpose crossbars similar with NeuroSim [10], which have digital-to-analog converters (DAC) and analog-to-digital converters (ADC) both on row-wise and column-wise.

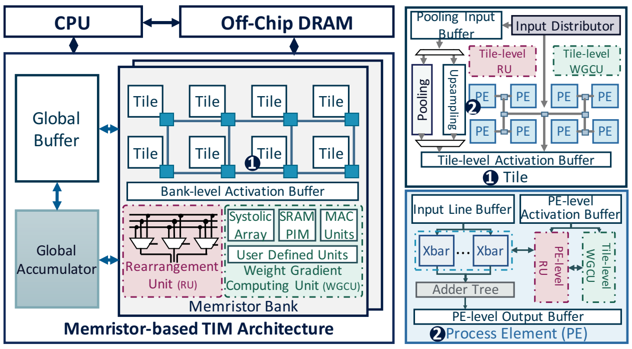

Fig. 1. The hierarchical modeling structure used in MNSIM-TIME.

To be specific, the transpose crossbars use DACs to activate rows and ADCs to read the output of columns for forward-computing, as the traditional memristor-based CNN inference accelerator. While for the back-propagation operations in training, the transpose crossbars use additional DACs to activate columns and additional ADCs to read the output of rows for error generation of FC layers. Secondly, in order to calculate the back-propagation error in CONV layers, we design vector rearrangement units to support rotation function in (1). Because the mapping of weight matrix onto crossbars is fixed, we rewrite (1) as (6) to replace the rotation of matrix with the rotation of input/output error vector for hardware implementation.

$$\delta^{l-1} = rot180(rot180(\delta^{l}) * W^{l}) \odot \sigma^{'}(z^{l-1}) \tag{6}$$

The vector rearrangement units design is shown in Figure 1, it contains multiple multiplexers to rearrange the positions of vector elements to perform $rot(\cdot)$. The number of multiplexers in each vector rearrangement unit is depended on the position of rearrangement units (*e.g.,* PE/tile level) and other hardware parameters (*e.g.,* crossbar size). When calculating back-propagation errors, several multiplexers are configured and activated due to the length of vector to be rotated.

Considering the generation of weight gradient, it should be noted that both (1) and (3) take the intermediate feature map as inputs, thus all the intermediate feature map should be stored during CNN training. In the CNN inference, we can discard the intermediate data that has been used in the forward process, making it possible to use line buffer with small capacity as the intermediate buffer in CNN inference accelerators [15]. However, the training process needs to save the intermediate feature map of each layer until the gradient calculation is completed, which brings a heavy storage burden for TIM architecture. For example, when training VGG-8 in TIM architectures with the batch size of $\boldsymbol{B}$ (usually $\boldsymbol{B} = 32, 64, 128...$), we need to store 320$\boldsymbol{B}$ KB. To solve this problem, we design the hierarchical feature map storage structure, which consists of activation buffers in different architecture levels and off-chip DRAM with large storage capacity. The hierarchical storage structure leverages on-chip activation buffers and off-chip DRAM to realize fast access for gradient calculations and adequate storage capacity for a huge intermediate feature map. Benefit from the layer-by-layer computation characteristics, the calculation, on-chip buffer read, and off-chip DRAM read can be paralleled to improve the performance. More details

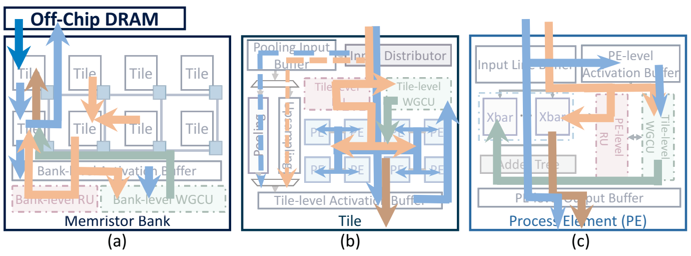

Fig. 2. Dataflow analysis for TIM architectures: (a) memristor bank level, (b) tile level, (c) PE level. Blue arrow: forward computation; orange/brown arrow: back-propagation; green arrow: weight matrix update.

will be discussed in Section III-C. Besides, to offer more flexibility for architecture customization to users, MNSIM-TIME allows users to define buffer sizes or DRAM capacity in different architecture levels, and we also provide some capacity calculation formulas according to CNN size as default.

After solving the storage problem of the intermediate feature map, we need to calculate the weight gradients according to (2) and (4). The computation of dW requires δ and a as inputs, and these inputs will be changed in every iteration. Thus, if TIM architectures also compute dW in memristors, we need to rewrite the memristor resistance value in every iteration. Due to the high write energy and latency cost of memristors and a large amount of data that needs to be rewritten, the overhead of calculating dW in memristors will be too high. Consequently, instead of implementing MNSIM-TIME architecture totally based on memristors, we adopt various CMOS-based computing units to realize matrix operations for dW generation (*i.e.,* weight gradient computation units), *e.g.,* systolic arrays in Eyeriss [13], SRAM-based PIM design used in [10], and simple multiply-accumulate operation units. Besides, MNSIM-TIME also offers users great flexibility to calibrate their own computing units design by modifying the configurable parameters of the weight gradient computation units, *i.e.*, area, power, and latency.

Besides, MNSIM-TIME also provides other CMOS-based computing units to support different activation functions in the algorithm level. Currently, MNSIM-TIME supports four activation functions, *i.e.*, ReLU, leaky-ReLU, tanh, and Sigmoid. These units are constructed based on the core function unit $exp(x)$. At the same time, we construct the upsampling modules to support error back-propagation in the pooling layer.

B. Data Flow Analysis

According to the proposed hierarchical modeling structure, MNSIM-TIME allows users to customize their own TIM architecture design by specifying the design parameters, placements, and connections of the computing units. On the basis of the hierarchical modeling structure, we propose the CNN training dataflow analysis for TIM architectures to describe the hardware behavior and model the performance considering various algorithm models and architectures. Figure 2 shows the dataflow details, which contain three parts:

1. Forward computation (the blue arrow): The difference between inference and training is that the latter one should store all the intermediate feature map in the hierarchical storage structure for further back-propagation. When one layer gets the calculation results, these intermediate data will be written into the line buffer (for inference) and the hierarchical storage structure (for back-propagation). Because the data transfer time and off-chip DRAM write time are much longer than the line buffer write time, writing intermediate data into the hierarchical storage structure and forward computation could be paralleled during forward computation.

2. Error back-propagation (the orange arrow represents δ^l, and the brown arrow represents δ^{l-1}): Back-propagation in MNSIM-TIME is computed layer-by-layer in a pipeline manner. When the back-propagation error δ^l reaches the l^{th} layer, δ^l should be rotated in the vector rearrangement units (to compute (1)) or fed to the transpose crossbars (to compute (3)). Because multiple tiles are needed for large-scale layers, the intermediate δ results of different tiles need to be merged together to get δ^l. After that, δ^l is transferred to the tiles of the $(l-1)^{th}$ layer. As for the error back-propagation in pooling layers, the data flow passes through the upsampling units to get δ^{l-1} from δ^l. The generation of dW relies on the back-propagation error δ^l and the input feature map a^{l-1}, and it has no data dependency with δ^{l-1} computation. Therefore, computations of (1) and (2) (or (3) and (4)) can be paralleled in MNSIM-TIME to accelerate CNN training. During the training process, a^{l-1} is read from the off-chip DRAM and hierarchical activation buffers.

3. Weight matrix updating (the green arrow): At the bank level, the data flow of dW will go to the tile which merges the output feature map of the same layer in CNN inference and then distributed it according to the CNN mapping results. dW is finally sent to different crossbars, and the weight matrix in crossbars are updated row-by-row. Besides, since the weight update can only be performed after calculating δ^l and dW^l, we parallel the weight matrix updating of the l^{th} layer and the error back-propagation of the $(l-1)^{th}$ layer.

C. Simulation Process

To model TIM architectures, existing work (*e.g.,* NeuroSim V2.0 [10] or TxSim [11]) simulates the computing behavior of memristor crossbars in the CNN training framework. Though this simulation method can simulate training accuracy, it may take too much time for one complete simulation due to the long training time of large CNN models. When concentrating on the hardware performance modeling, this long-time-simulation can not meet the rapid iteration requirements of design space exploration. To solve this problem, MNSIM-TIME focuses on modeling hardware performance from the architecture level without simulating the computing behavior in the algorithm code, which can provide efficient and fair performance comparison for design space exploration.

For the input algorithm model, MNSIM-TIME provides CNN training description interfaces to describe the training process of the specific CNN model, *e.g.*, CNN layers and size, number of training epochs, and batch size. According to these parameters and TIM architecture design, MNSIM-TIME maps the CNN model onto TIM architecture with the mapping strategy in [15] and computes the iteration number for training.

In the traditional simulation flow, designers need to write hardware description language (HDL) or SPICE code according to the architecture design, and then it will take a long time for

SPICE simulation and circuit synthesis. In MNSIM-TIME, we provide pre-simulation data of the basic computing units in the proposed hierarchical modeling structure. This helps us to extract the most time-consuming part (*i.e.,* circuit-level simulation) before the architecture design. After determining the architecture design and getting the mapping results, MNSIM-TIME analyzes dataflow, calculates the resource usage, and models the entire performance according to the unit performance. Since the architecture modeling is executed by looking up the pre-simulation data and approximate fitting instead of solving the circuit-matrices problems, the simulation results can be obtained in several seconds, which is suitable for architecture design space exploration.

IV. Case Studies

A. Experiments Setup

We use three typical CNNs on Cifar-10 dataset as benchmarks [4], *i.e.,* LeNet [7], VGG [5], and AlexNet [6]. The buffer (SRAM) data, DRAM data, and RRAM data come from [9], [3], and [8], respectively. The digital circuits modules (*e.g.,* vector rearrangement unit) are synthesized at TSMC 65nm technology node by Synopsys Design Compiler, and the frequency is set to 500MHz.

B. Latency and Energy Estimation

We evaluate the simulation results of latency and energy in one iteration (***B*** times forward computation and one-time back-propagation) on different CNNs and different batch sizes ***B***. The results are shown in TABLE I. As ***B*** increases from 16 to 64, the training latency of one iteration increases significantly. The reason is that the number of forward computations and back-propagation increases with the batch size, which are the most time consuming parts in CNN training. Besides, increasing batch size does not increase the times of updating the weight matrix in crossbars, which accounts for 65% of total energy in VGG-8 while ***B*** equals to 16 (one RRAM write operation consumes 2.84nJ [8]). Therefore, the energy consumption will not increase too much as the batch size increases.

C. Comparisons of CNN Inference and Training

As demonstrated in Section III, multiple hardware modifications are required for supporting back-propagation in TIM architectures. In this section, we will discuss these modifications from the following two aspects:

(1) To store all the intermediate feature map for back-propagation, we design the hierarchical storage structure in TIM architectures. Benefit from the hierarchical storage structure, the large capacity off-chip DRAM offers the opportunity to support training with large batch size. For example, for VGG-8, the required DRAM capacity is 4.99 MB when the batch size is 16. When the batch size increases to 64, the required capacity will increase to 19.97 MB. Besides the off-chip DRAM, the demand for on-chip buffer also increases. For the inference of VGG-8, buffers take up 7% area of the whole architecture. While in CNN training, the ratio of activation buffers to the total area is 19.84% as the total area increases by 22.97%.

(2) For the purpose of computing errors and updated weights, we modify the peripheral circuits of crossbars and add other digital modules. The transpose crossbars need additional DACs and ADCs for column-wise activation and row-wise readout, respectively. Other digital modules (*i.e.,* the weight gradient computation units, the upsampling units, and the vector rearrange units), together with the additional ADCs/DACs, totally take up 8.19% area of total architecture.

TABLE I
(Latency; Energy) Simulation Results under Different Batch Sizes ***B*** $(unit : ms, mJ)$

B	AlexNet	VGG-8	LeNet
16	263.66; 1,527.09	498.81; 826.01	9.62; 3.87
32	373.46; 1,587.24	823.83; 1,136.48	17.50; 4.35
64	641.47; 1,710.40	1,495.54; 1,797.64	36.18; 5.96

V. Conclusion and Future Work

In this paper, we propose MNSIM-TIME, a performance modeling framework for memristor-based training-in-memory architectures. Compared with circuit-level simulators, MNSIM-TIME helps architecture designers to customize the specific TIM architecture and can evaluate the hardware performance within several seconds, which is useful for architecture design space exploration. In the future, we will integrate more fine-grained hardware simulation in the training process to provide training accuracy simulation for TIM architectures as another configuration mode of MNSIM-TIME.

Acknowledgment

This work was supported by National Natural Science Foundation of China (No. U19B2019, 61832007, 61621091), National Key R&D Program of China (No. 2017YFA02077600); Beijing National Research Center for Information Science and Technology (BNRist); Beijing Innovation Center for Future Chips; Beijing Academy of Artificial Intelligence.

References

[1] Y. Cai et al. Long live time: Improving lifetime and security for nvm-based training-in-memory systems. *IEEE TCAD*, 39(12), 2020.
[2] M. Cheng et al. Time: A training-in-memory architecture for memristor-based deep neural networks. In *DAC 2017*. IEEE, 2017.
[3] K. Guo et al. Rram based buffer design for energy efficient cnn accelerator. In *ISVLSI 2018*, 2018.
[4] Kaggle et al. Cifar-10 - object recognition in images. website, 2014. https://www.kaggle.com/c/cifar-10.
[5] S. Karen et al. Very deep convolutional networks for large-scale image recognition. *Computer Science*, 2014.
[6] A. Krizhevsky et al. Imagenet classification with deep convolutional neural networks. In *NIPS, 2012*. 2012.
[7] Y. LeCun et al. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE, 1998*, Nov 1998.
[8] M. Mao et al. Optimizing latency, energy, and reliability of 1t1r reram through appropriate voltage settings. In *ICCD, 2015*. IEEE, 2015.
[9] N. Muralimanohar et al. Cacti 6.0: A tool to model large caches. *HP laboratories*, 27, 2009.
[10] X. Peng et al. Dnn+ neurosim v2. 0: An end-to-end benchmarking framework for compute-in-memory accelerators for on-chip training. *arXiv preprint arXiv:2003.06471*, 2020.
[11] S. Roy et al. Txsim: Modeling training of deep neural networks on resistive crossbar systems. *arXiv preprint arXiv:2002.11151*, 2020.
[12] L. Song et al. Pipelayer: A pipelined reram-based accelerator for deep learning. In *HPCA, 2017*, Feb 2017.
[13] Y. Chen and others. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. In *ISSCC 2016*, 2016.
[14] W. Zhang et al. Design guidelines of rram based neural-processing-unit: A joint device-circuit-algorithm analysis. In *DAC, 2019*, 2019.
[15] Z. Zhu et al. Mnsim 2.0: A behavior-level modeling tool for memristor-based neuromorphic computing systems. In *GLSVLSI 2020*. ACM, 2020.