This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2020.3043731, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems



# DNN+NeuroSim V2.0: An End-to-End Benchmarking Framework for Compute-in-Memory Accelerators for On-chip Training

Xiaochen Peng, *Student Member*, Shanshi Huang, *Student Member*, Hongwu Jiang, *Student Member*, Anni Lu, and Shimeng Yu, *Senior Member, IEEE*

***Abstract*— DNN+NeuroSim is an integrated framework to benchmark compute-in-memory (CIM) accelerators for deep neural networks, with hierarchical design options from device-level, to circuit-level and up to algorithm-level. A python wrapper is developed to interface NeuroSim with a popular machine learning platform: Pytorch, to support flexible network structures. The framework provides automatic algorithm-to-hardware mapping, and evaluates chip-level area, energy efficiency and throughput for training or inference, as well as training/inference accuracy with hardware constraints. Our prior inference version of DNN+NeuroSim framework available at *https://github.com/neurosim/DNN_NeuroSim_V1.2* was developed to estimate the impact of reliability in synaptic devices, and analog-to-digital converter (ADC) quantization loss on the accuracy and hardware performance of an inference engine. In this work, we further investigated the impact of the "analog" emerging non-volatile memory (eNVM)'s non-ideal device properties for on-chip training. By introducing the nonlinearity, asymmetry, device-to-device and cycle-to-cycle variation of weight update into the python wrapper, and peripheral circuits for error/weight gradient computation in NeuroSim core, we benchmarked CIM accelerators based on state-of-the-art SRAM and eNVM devices for VGG-8 on CIFAR-10 dataset, revealing the crucial specs of synaptic devices for on-chip training. The latest training version of the DNN+NeuroSim framework is available at *https://github.com/neurosim/DNN_NeuroSim_V2.1*.**

***Index Terms*— Emerging non-volatile memory, deep learning, on-chip training, in-memory computing, hardware accelerator**

## I. Introduction

THE state-of-the-art deep convolutional neural networks (CNNs) have shown remarkable breakthroughs in various applications, including speech recognition and image classification. As the popular CNNs tend to introduce huge amount of high-dimensional convolutional layers and hundreds of megabytes of parameters, to solve the bottleneck of extensive data transfer in the conventional von Neumann architectures, compute-in-memory (CIM) has emerged as a promising paradigm for designing the machine learning hardware accelerator [1].

Emerging non-volatile memory (eNVM) devices such as RRAM [2], PCM [3], EpiRAM [4], ECRAM [5] and FeFET [6] have been proposed by the device community as candidates of "analog" synaptic devices, to represent the weights of deep CNNs in CIM accelerators. To evaluate these device properties from system-level perspective, we published a prior work in IEDM 2019 [7], whose latest version is named as DNN+NeuroSim V1.2, and served as an end-to-end benchmarking framework for the inference engine design. It supports flexible deep CNN topologies and versatile device technologies from CMOS to beyond-CMOS, with automatic CIM floor-planning to evaluate inference engine hierarchically. We focused on the impact of variability/reliability in synaptic devices, such as conductance variation and retention, and analog-to-digital converter (ADC) quantization loss, to investigate the trade-offs among inference accuracy, energy efficiency, throughput, chip area and memory utilization. Therefore, the DNN+NeuroSim V1.2 can be used as a supporting tool to find optimal design options of CIM inference engine for various synaptic device candidates and neural networks. By benchmarking the popular synaptic devices (including RRAM [2], and FeFET [6]) on CIM inference engine for VGG-8 with CIFAR-10 dataset, we learnt that, for "analog" synaptic device based CIM designs, the parallel read-out scheme and large on-state resistance (>100kΩ) are two of the most important parameters to achieve superior energy efficiency (in TOPS/W) and throughput (in TOPS) in the CIM accelerators for inference.

To further study the potential applications of various synaptic devices for on-chip training, here we introduce more non-ideal properties of synaptic devices that are critical for *in-situ* training accuracy, such as nonlinearity and asymmetry, device-to-device variation and cycle-to-cycle variation during weight update. In CIM accelerators, to support on-chip training, we also implement extra peripheral circuits to calculate error and weight gradient in the back-propagation.

In this work, we extended the DNN+NeuroSim framework to support evaluation of on-chip training performance in CIM accelerators, and benchmark across SRAM and versatile eNVM devices for VGG-8 on CIFAR-10 dataset. The rest of the paper is organized as follows: Section II introduces the framework structure of DNN+NeuroSim V2.0. Section III describes the detailed architectures to support feed-forward and back-propagation computation in deep CNNs. Section IV discusses about benchmark results of CIM accelerators in on-chip training with versatile synaptic devices. Section V summarizes the work.

## II. Integrated Framework Principles

Fig. 1 shows the framework structure of DNN+NeuroSim V2.0. As what has been proposed in prior framework [7], the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2020.3043731, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems



NeuroSim core is wrapped by the python library, to support flexible network topologies, and the default model is VGG-8 for CIFAR-10 based on low precision training method WAGE [8]. However, larger model such as ResNet-18 is also supported and arbitrary CNN topology could be defined by the user.

As shown in Fig. 1 (b), during training phase, non-ideal properties of synaptic devices during weight update are introduced into the python wrapper, including nonlinearity and asymmetry, device-to-device and cycle-to-cycle variations. It should be noted that, the number of pulses that will be applied to each synaptic device (to update the weights) are defined in a linear relationship with the calculated delta weights (accumulated by digital circuit modules). However, the actual weight-updated is not simply by adding linear weight gradients on current weights, but by applying distorted or random conductance increments to the synaptic devices owing to device-to-device or cycle-to-cycle variations. Such non-ideal weight-update will lead to accuracy degradations in *in-situ* training.

In Fig. 1 (c), we show the device retention model [9] and ADC quantization loss of partial sums during feed-forward operations, which were previously introduced in [7]. Fig. 1 (d) shows the simulator taking the predefined network topology as input to automatically design the chip floorplan, while weight-duplication [10] is introduced to maximize memory utilization (defined as percentage of the used memory over the total memory capacity) to certain layers in the network. This is a feature needed for convolutional layer where the unrolled kernel size is smaller than the memory sub-array size (typically in shallow layers), in order to speed up DNN processing.

Fig. 1 (e) shows that in the python wrapper, the neural activations and updated synaptic weights are stored for feed-forward evaluation, and the old synaptic weights (before weight update) are also stored for weight update evaluation during back-propagation. Within one epoch, due to the different weight gradients in each iteration, the hardware performance will also be different, however, to limit the overhead of running time in the framework, by default, we only take the real traces (neural activation, new and old synaptic weights) from the last iteration in each epoch, and run the traced-based simulation in NeuroSim core only once for every epoch in V2.0. By doing so, we could guarantee reasonable simulation time, while still take a track of the hardware performance among different epochs during the iterative training. By contrast, in [7] we could

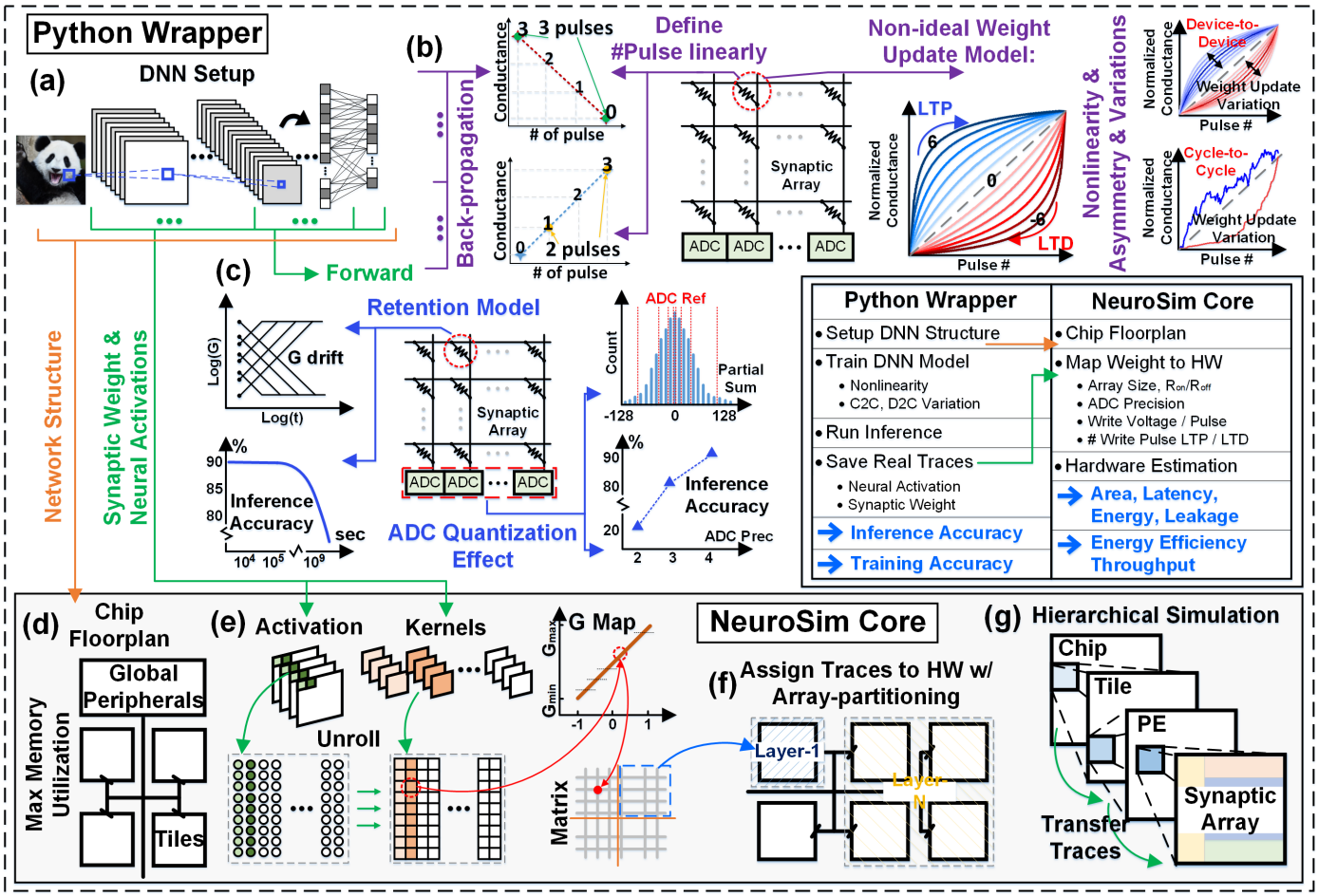


Fig. 1. Framework structure of DNN+NeuroSim V2.0. (a) DNN setup in python wrapper; (b) during training, introducing non-ideal properties of synaptic devices, including nonlinearity, asymmetry, device-to-device, and cycle-to-cycle variation during weight update; (c) during inference, introducing retention model and ADC quantization effects; (d) pre-defined network structure is loaded as input to NeuroSim core, for automatic floor-planning with weight-duplication to maximize memory utilization; (e) loading real trace (synaptic weights and neural activations) into NeuroSim, mapping data to conductance and digital voltage input cycles; (f) traces to be partitioned and assigned to different locations of the CIM system; (g) hierarchical simulation from chip to tile, and from processing element (PE) to synaptic array.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2020.3043731, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems



support real-trace based simulation as inference is not iterative.

Fig. 1 (f) shows the traces are partitioned and assigned to different locations of the chip according to the automatic floor-planning rule as introduced in [7]. The top-down hierarchy of the CIM system is defined as chip, tile, processing element (PE) and synaptic array. The framework outputs include the hardware-constrained training or inference accuracy (from python wrapper), and hardware metrics such as chip area, latency, dynamic energy, leakage power, as well as energy efficiency and throughput for training or inference (from NeuroSim core). The modular circuit component estimation are all calibrated by SPICE simulations across technology nodes from 130nm down to 7nm with PTM models [11], as shown in our prior MLP+NeuroSim work [12] (a small-scale 2-layer fully-connected network benchmarking). The calibration is also presented in the supplimentary material of this paper.

## III. CIM Architecture For Training

In CIM accelerators, to support training, additional peripheral circuits to calculate errors and weight gradients are necessary to be implemented. In this section, we discuss about the detailed architectures for the four key steps in training, namely, 1) feed-forward, 2) computation of errors, 3) computation of weight gradients, and 4) weight update. Steps 2) to 4) represent the backpropagation. We firstly introduce the main hierarchical design of the entire CIM architecture, and then breakdown to the details of different computation steps.

### A. CIM Architecture

As Fig. 2 shows, in chip-level, the key components of the CIM accelerators are tiles, global buffer, neural functional units including pooling, accumulation and activation. There are four main hierarchies in the CIM accelerators, chip level, tile level, processing element (PE) level and synaptic array level. In different levels, peripheries are introduced, including buffers, interconnects (based on H-tree routing), and computational units (such as adder trees).

To support backpropagation on-chip, our prior works [13~15] proposed transposable synaptic arrays, where [13] presents a SRAM-based CIM chip macro with transposable function; [14] and [15] proposed optimized CIM architectures based on RRAM and SRAM, respectively. In this way, the synaptic weights (which are stored in the transposable synaptic arrays) are not only used in feed-forward, but also can be efficiently reused to compute errors in backpropagation. More details of the transposable synaptic arrays and error computation will be discussed in Section III-C.

Besides the error computation, we also implement the weight gradient computation function at the chip-level, which is mainly built up by a group of SRAM-based CIM arrays, local buffers and accumulation units. In the weight gradient computation function, we will map errors into the SRAM-based CIM arrays, and apply the activations as the input to the arrays, to generate weight gradients as the output in a CIM fashion (more details in section III-D). Since we need to frequently write errors into the arrays (along the batch), we choose the SRAM-based CIM arrays in the function unit. Although SRAM is not as area-efficient as eNVMs, and also has the problem of standby leakage, its fast/low energy write performance and infinite endurance still makes it more suitable for the gradient computation compared with the eNVMs. More details of the weight gradient computation function will be discussed in Section III-D.

### B. Training Strategy in CIM Architecture

Normally the mini-batch based training with batch size=$\boldsymbol{B}$ is used, thus it means the number of intermediate data that need to be utilized and stored is huge. For example, during feed-forward, the $\boldsymbol{B}$ activations of all the layers will be stored to be used for the computation of weight gradients later in backpropagation; the $\boldsymbol{B}$ computation of errors of all the layers obtained in backpropagation will be stored; and the $\boldsymbol{B}$ weight gradients in one batch will be stored and accumulated to produce the incremental weight gradients for the final weight-update after a batch. Therefore, if the batch size is $\boldsymbol{B}$, we have to store $\boldsymbol{B}$ copies of activations, errors and weight gradients of

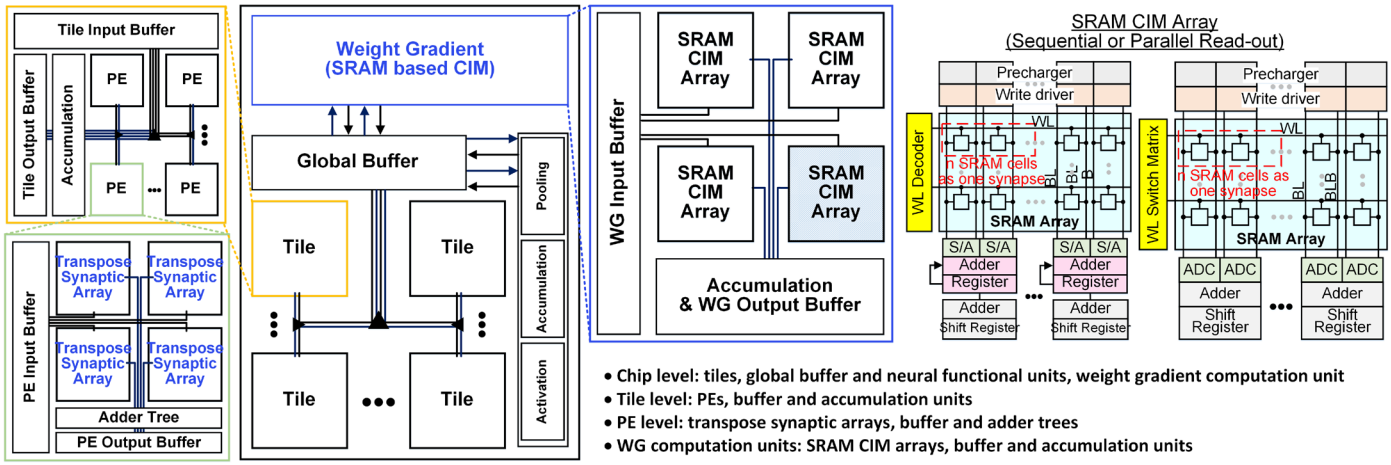


Fig. 2. Architecture structure defined in the simulator, the top level of chip contains tiles, global buffer, neural-functional peripheries (including pooling, accumulation and activations) and weight gradient computation function which is mainly built up with a group of SRAM-based non-transposable CIM arrays. Inside a tile, it is further portioned into multiple processing elements (PEs), while each PE consists of several synaptic arrays, along with adder trees and local buffers. H-tree routing is used for interconnect. To support training, the synaptic arrays are designed transposable.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2020.3043731, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems



the entire network, before we can update the weights for a specific batch. To limit the on-chip buffer overhead, we assume that those intermediate data will be sent back to off-chip memory (i.e. DRAM) for the entire batch, and will be retrieved back to chip for error and weight gradient computation (Section III-C and III-D).

Following this dataflow, we simplify the scheduling strategy in Fig. 3, and assume that: the feed-forward, computation of errors, computation of weight gradients across the batch and weight-update will not be operated simultaneously on the CIM accelerators, which also helps us to limit the hardware overhead for those additional back-propagation computational units (Section III-D):

1) *Feed-forward (#1):*
   During feed-forward, the ***B***× images will be fetched to on-chip global buffers one by one from the off-chip memory, and then delivered to the CIM arrays for computation. Meanwhile, the activations of each layer will be sent back to off-chip memory for the ***B***× images.
2) *Error computation (#2):*
   After feed-forward, the ***B***× errors will be fetched to on-chip global buffers one by one from the off-chip memory (#2), and then delivered to the CIM arrays for computation. Meanwhile, the ***B***× errors of each layer will be sent back to off-chip memory.
3) *Gradient computation (#3):*
   Similarly, during the computation of weight gradients, ***B***× errors and ***B***× activations will be fetched to on-chip buffers from the off-chip memory, and then delivered to the weight gradient computation units. It should be noted that, to limit hardware overhead, we assume the SRAM-based CIM arrays are large enough to support the gradient computation of the largest layer in the network. Hence, during weight-gradient computation, the errors will be fetched to on-chip buffer layer by layer for each image, and then image by image for the entire batch. At the same time, the ***B***× weight gradients will be sent back to off-chip memory layer by layer for each image, and then image by image for the entire batch.
4) *Weight update (#4):*
   Finally, to update the weights, we need to accumulate the weight gradients and calculate the delta weights. The on-chip digital accumulation units (precision and numbers) are assumed to be able to support at least ***B***× weight gradients inside one single CIM synaptic array. As is shown, to calculate the delta weights to be updated, each layer will be partitioned into multiple parts, and for each part, the gradients will be fetched to on-chip buffer and be accumulated image by image, after the weight gradients are accumulated for the entire batch, one specific CIM synaptic array will be updated. Meanwhile, we can start with the computation of the delta weights for the next part of the current layer. Thus, the weight gradients will be fetched to on-chip part by part (across entire batch) and then layer by layer. This means that the CIM synaptic arrays will be updated one by one for each layer, and then layer by layer for the entire network.

In DNN+NeuroSim V2.0, we do not consider inter-step pipeline among the four key steps in training, i.e. #1 feed-forward, #2 computation of errors, #3 computation of weight gradients, and #4 weight update, but the users can potentially optimize the design as done in other works [13~15]. However, the framework provides an option to build up intra-step pipeline system for feed-forward and computation of errors, as we assume all the weights are stored on-chip in CIM synaptic arrays, we can process multiple images simultaneously on-chip, i.e. inter-image pipeline with on-chip global buffer overhead.

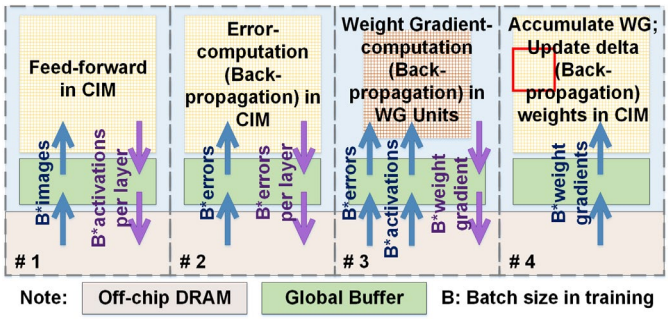


Fig. 3. The schedule of mini-batch based (batch size is B) training in CIM architecture.

This assumption (of four separated training steps) helps us to limit the global buffer size, as the global buffer will not be used to support different operations simultaneously, the specs for global buffer size is to hold all the intermediate data to complete the computation for each layer or each step as mentioned above. In other words, the maximum requirements of on-chip buffer among the four separate operations (feed-forward, computation of errors, computation of weight gradients across the batch and weight-update) will decide the size of global buffer.

For example, in a simple design, where the operations of #1 feed-forward and #2 computation of errors are layer by layer, and image by image, while #3 computation of weight gradients and #4 weight update are assumed for one (CIM synaptic) array by array, and layer by layer, the global buffer size will equal to *MAX* (activation size of largest layer, error size of largest layer, weight gradient size of largest layer). Similarly, the on-chip computational hardware (such as accumulation units) are also limited, thus, the overall on-chip buffer size of the CIM accelerator is still acceptable even for training.

### C. Feed-Forward

In Fig. 4, the computation of convolutional layer during feed-forward is shown as computation among tensors. In layer<n>, the size of input feature maps (IFMs) is $\boldsymbol{W}\times\boldsymbol{W}\times\boldsymbol{D}$ (where $\boldsymbol{D}$ is the depth of input feature channel), which are the outputs from layer<n-1>. The size of each 3D kernel is $\boldsymbol{K}\times\boldsymbol{K}\times\boldsymbol{D}$, with kernel depth of $\boldsymbol{N}$ (i.e. there are $\boldsymbol{N}$ such 3D kernels), thus the total size of the kernels in layer<n> will be $\boldsymbol{K}\times\boldsymbol{K}\times\boldsymbol{D}\times\boldsymbol{N}$. To get the outputs, a group of IFMs (with size $\boldsymbol{K}\times\boldsymbol{K}\times\boldsymbol{D}$) will be selected at each time, and to be multiplied and accumulated with $\boldsymbol{N}$ kernels with size $\boldsymbol{K}\times\boldsymbol{K}\times\boldsymbol{D}$, then each of them will generate a $\boldsymbol{1}\times\boldsymbol{1}\times\boldsymbol{1}$ output, the output from the top kernel (shown as light orange cube) goes to the front, and the output from the bottom kernel (shown as dark orange cube) goes to the back, thus, in total there will be $\boldsymbol{1}\times\boldsymbol{1}\times\boldsymbol{N}$ outputs.

It is obvious to see that part of the input data will be reused

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2020.3043731, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems



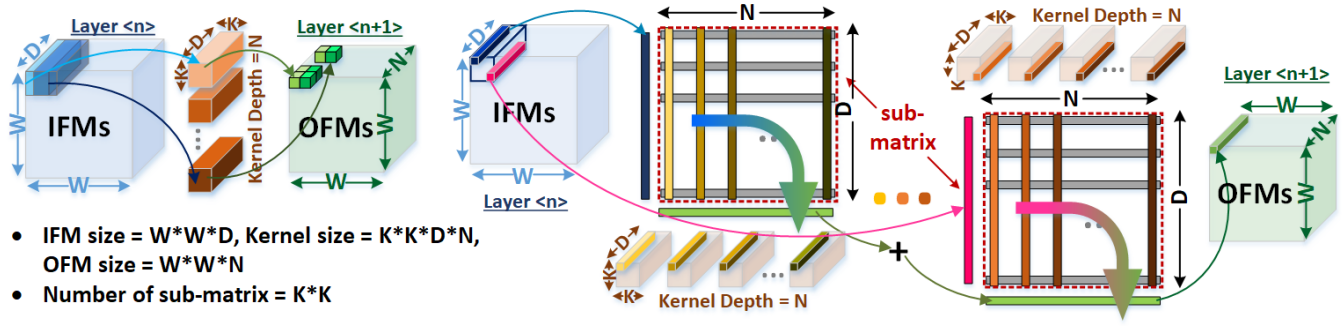


Fig. 4. A novel mapping method of convolutional layer in feed-forward [16] to maximize input data reuse, where the weights are mapped along their spatial location to a group of sub-matrix. K×K kernel is mapped to K×K sub-matrices (or processing elements, PEs). Partial sums are accumulated by adder tree.

for the computation of the next output. To realize the input data reuse practically, we have proposed a novel mapping method and dataflow for CIM inference in prior work [16], where the weights at different spatial locations of each kernel are mapped into different sub-matrices. As Fig. 4 shows, if we cut each ***K×K×D*** kernel along its first and second dimensions, we will get several ***1×1×D*** partitioned kernel data, and for each kernel, there are ***K×K*** of them. Similarly, the input data which should be assigned to various spatial locations in each kernel, will be sent to the corresponding sub-matrices respectively and reused among their neighbors. To practically map and operate large convolutional layers on chip, array partitioning [17] is introduced, which could split a single large matrix into several sub-arrays, and parallelize the computation efficiently.

In this framework, we also utilize this novel mapping method for CIM training, since it groups the kernels according to their spatial locations, which makes it efficient to implement transposable synaptic array to calculate the errors. Beside the convolutional layers, the fully-connected (FC) layers can also be mapped as special convolutional layers (since they are simple 2D vector-matrix multiplications) with kernel size equals to 1×1. For example, in the VGG-8, the size of weight-matrix in the first FC layer equals to 8192×1024, it can be considered that, the IFM size is 1×1×8192, each kernel size is 1×1×8192, and the kernel depth is 1024, finally the output size is 1×1×1024. Thus, this FC layer is directly mapped as the weight-matrix into a group of PEs (with multiple sub-arrays).

### *D. Backpropagation for Error*

As Fig. 5 shows, during backpropagation, the errors (i.e. the gradient of loss function respective to the activation) from deeper layers will be fetched backwards and conduct the element-wise multiplication and accumulation with the prior kernels. In layer<n>, the errors from layer<n+1> will be the input data, and be separated into different channels, then applied to corresponding kernels. For example, the first channel of errors (shown as light green plane) will be applied to the first channel of the first kernel (shown as light yellow plane), to do the element-wise multiplication; the last channel of errors (shown as dark green plane) will be applied to the first channel of the last kernel (shown as dark yellow plane), to do the element-wise multiplication; then the partial-sums from first channels of all the kernels will be accumulated, and become the first element in the first channel of layer<n>'s error.

According to the novel mapping method, the kernels are partitioned based on their spatial locations and collected along their channels (dimension=***D***), and mapped into the columns in CIM synaptic arrays. It would be considered in a transposed way that, the rows in such CIM arrays are the weights in a specific location in each kernel along different channels (dimension=***N***). For example, as shown on the right-side of Fig. 5, the first channel of the weights that are on top-left corners of each kernel (shown as light yellow nodes) are mapped as the first row in the first sub-matrix; the last channel of the weights that are on bottom-right corners of each kernel (shown as dark orange nodes) are mapped as the last row in the last sub-matrix.

In other words, in such synaptic weight matrix, the rows represent channel depth, and the columns represent kernel depth. Thus, accumulating products along columns in synaptic weight matrix means accumulating products along channels in DNN models (feed-forward); and accumulating products along rows in synaptic weight matrix means accumulating products

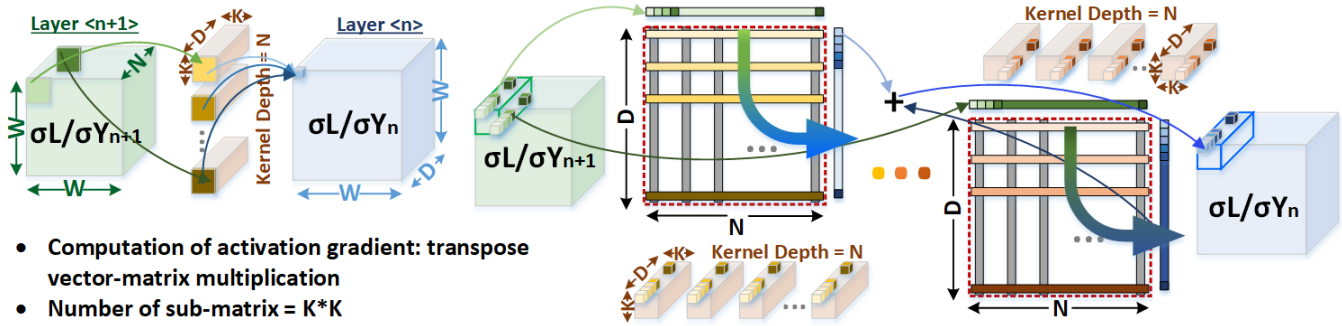


Fig. 5. The computation of error in CIM during backpropagation, based on novel mapping method [16] of convolutional layer.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2020.3043731, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems



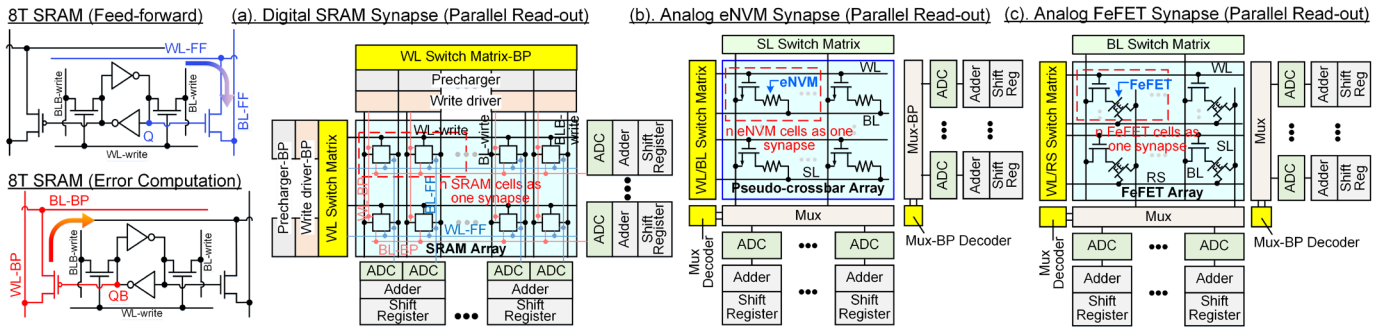


Fig. 6. In DNN+NeuroSim V2.0, transposable synaptic arrays are implemented to support on-chip training. Available synaptic devices are (a) SRAM, (b) two-terminal eNVMs (e.g. RRAM) and (c) three-terminal eNVMs (e.g. FeFET), with both sequential and parallel read-out schemes (only parallel designs are shown in the figure as examples).

along kernels in DNN models (error computation).

In this case, it is straightforward to find that, in a transposed way of feed-forward, we can automatically accumulate the products along each rows in the CIM array, and sum up the partial-sums among sub-matrix to get the errors. Thus, the errors in layer<n+1> will be sent to each sub-matrix according to their spatial locations, and each of them will be considered as inputs to each column in the CIM arrays.

Therefore, we need to introduce transposable synaptic arrays in the CIM architecture, to support both conventional (feed-forward) and transposed (error) computations. As shown in Fig. 6, in this framework, we provide the transposable synaptic array designs with versatile synaptic devices, ranging from SRAM, two-terminal eNVMs (like RRAM) and three-terminal eNVMs (like FeFET), and can be designed with either sequential (compute-near-memory) or parallel (compute-in-memory) read-out schemes.

In SRAM-based synaptic arrays, during feed-forward, we need to activate each row, then read out and accumulate the products along the columns. During backpropagation, to calculate the errors, we need to activate each column and accumulate the products along rows. To do so, as prior work [15] proposed, we need to implement transposed word-lines, bit-lines with 8T-SRAM cells [18] to realize transposed computation, with additional peripheral circuits (such as WL decoder or switch matrix, sense amplifier or ADC, and shift-adder with registers). One example of 8T-SRAM transposable cell is shown in Fig. 6, during feed-forward, the additional NMOS (in blue lines) is activated, WL-FF acts as activation input and BL-FF is used as bit-line to accumulate products along columns; during backpropagation, the additional PMOS (in red lines) is activated, error is fed through WL-BP, and BL-BP is used to accumulate products along rows.

On the other hand, in eNVM-based synaptic arrays, there are SL switch matrix (in two-terminal eNVM designs) or BL switch matrix (in three-terminal eNVM designs) for weight update. Hence, in transposed computation (for error), we could use the SL or BL switch matrix to rotate the input and the output. In this case, for eNVM-based synaptic arrays, we only need to add sense amplifier or ADC (along with adder, shift-adder and registers) to read out the partial-sums horizontally along rows. The reason that we do not reuse ADC between rows and columns is to avoid the complex interconnect routing.

### *E. Weight Gradient Calculation*

When the computation of errors is done and the errors are stored to the off-chip DRAM memory (for each batch), we need to start the computation of weight gradients. As Fig. 7 shows, to calculate the weight gradients of layer<n>, the errors from layer<n+1> will be applied to perform the element-wise multiplication and accumulation with the activations from layer<n> in a channel-to-channel manner.

For example, a part of the first-channel activations in layer<n> (shown as light green plane) will be multiplied with the first-channel errors in layer<n+1> (shown as light blue plane), the element-wise products will be accumulated and become the first weight gradient in the first channel of the first kernel (shown as light yellow node). Similarly, the last-channel activations of layer<n> (shown as dark green plan) will be multiplied with the last-channel errors of layer<n+1> (shown as dark blue plane), and accumulated to be the first weight gradient in the last channel of the last kernel (shown as dark orange node). During this process, we can get the weight gradients at top-left corners through all the channels for each kernel.

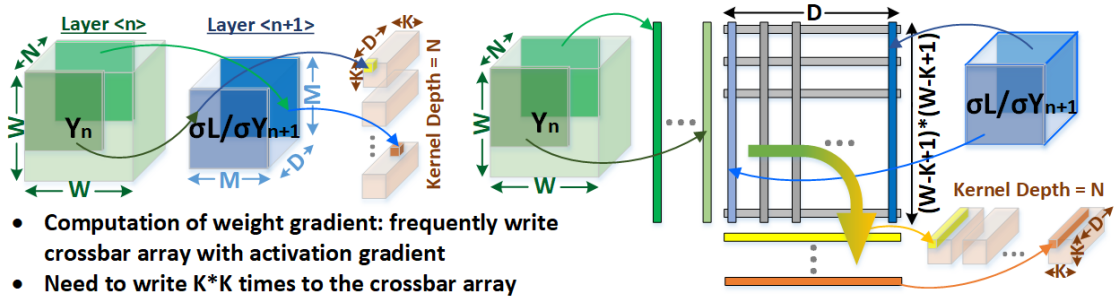


Fig. 7. The computation of weight gradient in CIM.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2020.3043731, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems



To obtain all the weight gradients, we need to sweep the activations and repeat the multiplication-and-accumulation with the errors by $K \times K$ times, representing $K \times K$ spatial locations in the kernels. In other words, the errors will be reused by $K \times K$ times. Therefore, we could easily unroll each channel of the errors into a long column, as the products will be accumulated inside each channel, and with number of channels equals to $D$, there will be $D$ such long columns to form a large matrix. The activations will be treated as the inputs to the matrix, i.e. will be applied to the rows of matrix, and perform vector-matrix multiplication. With channel depth equals to $N$, there will be $N$ unrolled vectors of inputs applied to the matrix, and calculate one group of the weight gradients. These gradients correspond to the weights at a specific spatial location of each kernel, and represent one of the $K \times K$ sub-matrix in the novel mapping method of the weights. Hence, to sweep the activations by $K \times K$ times to get all the weight gradients, in total there will be $K \times K \times N$ unrolled vectors of activations applied to the matrix of errors.

Since the size of error matrix in the popular deep CNNs could be quite large, we will also introduce array-partitioning [17] into this weight gradient computation, to avoid large memory operations. As the weight gradients need to be calculated across the batch for each iteration, we need to frequently re-write the matrix of errors for different images, which leads to a huge overhead of memory write latency and energy. Due to this, we choose the SRAM-based CIM non-transposable arrays for weight gradient calculation (as Fig. 2 shows) over other eNVM-based designs, to avoid the huge memory programming overhead and the limited endurance. It is noticeable that the SRAM-based CIM design could cause a larger area overhead compared with eNVM-based ones. To minimize the on-chip hardware resources for weight gradient computation, we do not need to process all the layers simultaneously, but just perform layer-by-layer weight gradient computation. Therefore, we only put limited number of SRAM-based CIM arrays on chip which are able to support one layer with the largest size of errors. The area overhead of SRAM-based weight gradient computation module is acceptable. For the layers whose unrolled errors' sizes are smaller than the SRAM-based CIM arrays, we will duplicate the errors in SRAM-based CIM arrays to speed up the computation.

### F. *Weight Update*

During the computation of weight gradients (across the batch), the weight gradients of each image will be sent back to the off-chip memory successively. Therefore, before we can update the weights in the CIM accelerator, we need to reload the weight gradients back and accumulate them to get the delta weights of each layer.

Similarly as what we designed for the weight gradient computational units, to minimize the area overhead, we assume that as long as the digital accumulation units in chip-level can support one specific portion of weight gradients accumulation (across the batch), it is ready to process the actual weight update (programming the weights to synaptic arrays). In other words, with limited accumulation units, we can process the weight

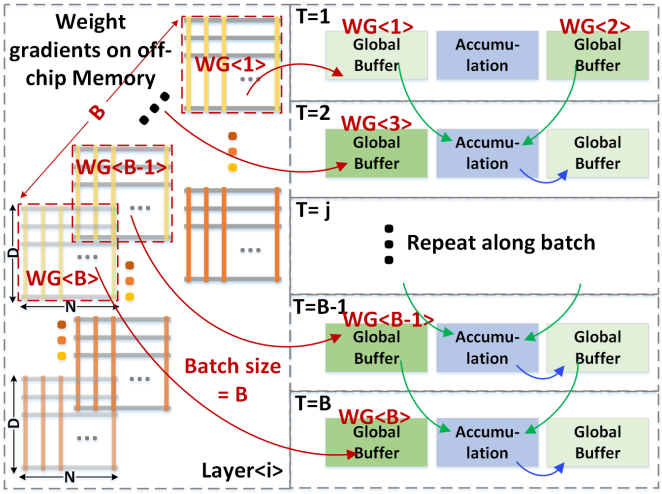


Fig. 8. Example of weight gradient accumulation for one synaptic array with a limited global buffer size.

gradient accumulation (across batch) part-by-part for a layer, and then layer-by-layer for the entire network.

For example, as Fig. 8 shows, we are processing the weight gradient accumulation and weight update for a specific synaptic array in layer<i>. At the very beginning cycle (T=1), we preload the weight gradients of the first two images in the batch (to global buffer). At the next cycle (T=2), these two weight gradients will be loaded to accumulation units for computation (green arrows), meanwhile, the weight gradients of the third image will be loaded into the global buffer (red arrow), and the accumulated gradients will be sent back to the global buffer (blue arrow). Similarly, we can continue these operations for the following images, until we accumulate the gradients for the entire batch. As we assume enough accumulation units to support accumulation of weight gradients (with size equals to one synaptic array), at a specific cycle the new gradients and the accumulated gradients (from prior cycle) can be loaded to accumulation units and be processed simultaneously, and generate the same amount of accumulated gradients (to be saved back to global buffer). Thus, as long as the global buffer is released and the results are ready inside the accumulation units, we can start loading in new weight gradients and store the newly accumulated gradients to the released global buffer. In other words, during such successive accumulation operations, we only need the global buffer storage to be (2×synaptic-array-size×highest-possible accumulated weight gradients precision).

Apparently, this extreme assumption on updating only one synaptic array can help us to significantly limit the area overhead for training, but as a trade-off, it will cause a large latency overhead during such successive weight gradient accumulation, i.e. $B\times$ (buffer-read + accumulation + buffer-write). Thus, in DNN+NeuroSim V2.0, we also provide an option to gradually increase global buffer size, namely, "buffer overhead constraint", with larger constraint ratio, the global buffer size increase, which can support weight gradients of more synaptic arrays, correspondingly, the chip-level accumulation units will also increase, and thus decrease the latency of gradient accumulation and weight update (more

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2020.3043731, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems



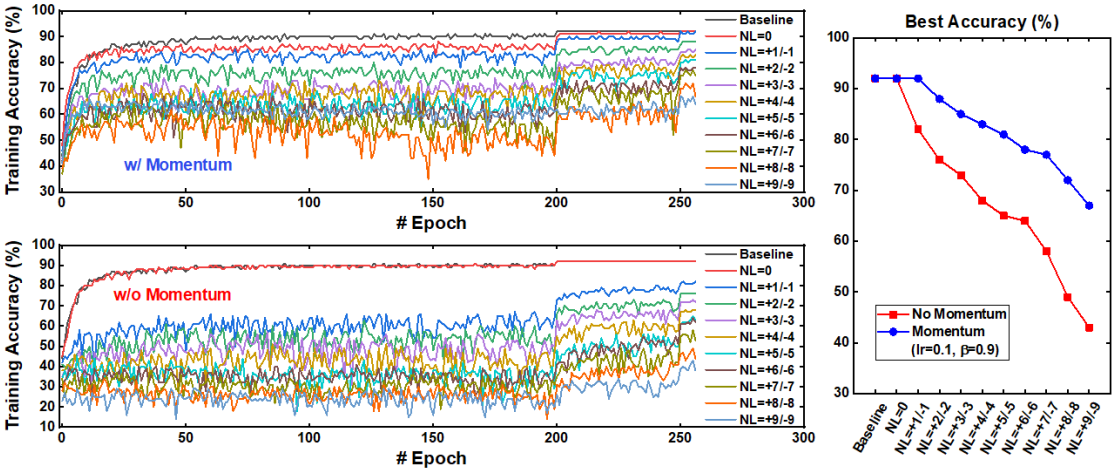


Fig. 9. Analysis of nonlinearity and asymmetry for in-situ training, w/ and w/o momentum optimization.

synaptic arrays can be updated simultaneously). With such option, the user can find an optimal design option with a specific area constraint.

## IV. Benchmark Results

In CIM accelerators for inference-only, the most critical hardware factors of inference accuracy and performance are the on-state resistance $R_{on}$ and ADC precision as shown in our prior work [7]. In this work, we take more emphasis on the non-ideal synaptic device properties (including nonlinearity and asymmetry, device-to-device and cycle-to-cycle variation) for *in-situ* training. We benchmark across device technologies based on VGG-8 for CIFAR-10 dataset. To study the impact of device precision, we assume each eNVM based synaptic weight will be represented by only one device cell, with the exception that n-bit weight is represented by n SRAM cells.

### A. Impacts of Non-ideal Synaptic Device Properties

To analyze the impacts of non-ideal synaptic device properties on the training accuracy, we introduced the models of nonlinearity and asymmetry, device-to-device and cycle-to-cycle variations into the python wrapper, to evaluate the degradations of training accuracy under these non-ideal properties. We run the VGG-8 on CIFAR-10 dataset, and sweep the different values for each non-ideal property, to quantify their effects individually. In this case, we fixed the precision of activation, weight, gradient and error to be 8-bit, which implies that the synaptic device precision is considered as 8-bit (256 levels).

The model of nonlinearity and asymmetry can be expressed by the following equations [12], where the updated conductance value is in a nonlinear relationship with the number of pulses ($P$), the $G_{max}$, $G_{min}$ and $P_{max}$ represent the maximum, minimum conductance values and the maximum number of pulses that synaptic device can achieve (i.e. pulse resolution). $A$ is used to determine the nonlinear behavior of weight update, the value range of $A$ is (0, +∞), the smaller the $A$ is, the more nonlinear the weight-update will be, while $B$ is the function of $A$ that is adjustable along the range of $G_{max}$, $G_{min}$ and $P_{max}$.

$$G_{LTP} = B\left(1 - e^{-P/A}\right) + G_{min} \quad (1)$$

$$G_{LTD} = -B\left(1 - e^{(P-P_{max})/A}\right) + G_{max} \quad (2)$$

$$B = (G_{max} - G_{min})/(1 - e^{-P_{max}/A}) \quad (3)$$

Fig. 9 shows the training accuracy under different nonlinearities with asymmetry, without device-to-device or

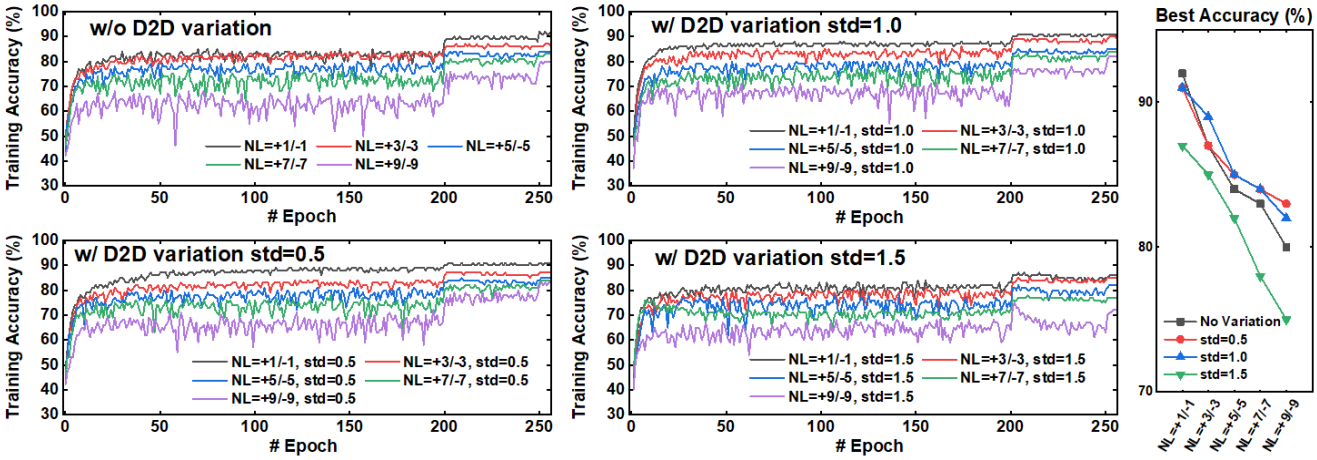


Fig. 10. Analysis of device-to-device variation under different nonlinearities, w/ momentum optimization.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2020.3043731, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems



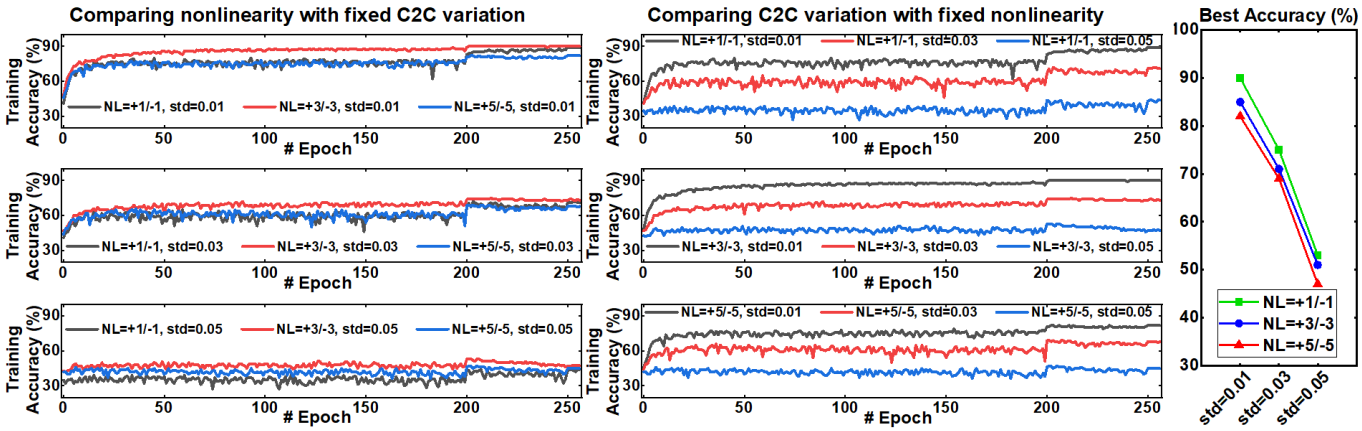


Fig. 11. Analysis of cycle-to-cycle variation under different nonlinearities, w/ momentum optimization.

cycle-to-cycle variations. The nonlinearity (NL) factor ranges from –9 to +9, and NL=+6/-6 are shown in long-term potentiation (LTP) and long-term depression (LTD) curves in Fig. 1. Our recent work [19] argues that the momentum optimization in the weight update rule can significantly help to overcome the drawbacks of large nonlinearity and asymmetry. In this V2.0 framework, we also introduced the momentum optimization method as below:

$$\Delta w_j = \beta \times \Delta w_{j-1} + (1-\beta) \times g_j \quad (4)$$

$$w_j = w_{j-1} - lr \times \Delta w_j \quad (5)$$

where $\beta$, $\Delta w_j$, $g_j$, $lr$ and $w_j$ denote the momentum factor, momentum, gradient, learning rate and weight at j[th] batch of one epoch. Normally the momentum factor $\beta \in [0,1]$, by sweeping it we empirically set momentum $\beta = 0.9$ [19] for the best accuracy under moderate nonlinearities. The results in Fig. 9 show that with $\beta = 0.9$ the *in-situ* training accuracy could achieve ~85% with a moderate asymmetric nonlinearity (NL=+3/-3) and ~77% even with a large asymmetric nonlinearity (NL=+6/-6). Prior works from different groups [20~22] show a universal trend in the training accuracy vs. weight update asymmetric nonlinearity. Our results agree with the trend but shows a much alleviated accuracy degradation with increasing the NL factor thanks to the introduction of momentum.

During weight update, the device-to-device (D2D) variation will introduce different nonlinearities to different synaptic devices. As a behavior model, we randomly generate the NL factors to different synaptic weights with a standard deviation (σ) respect to the mean nonlinearity value (μ). Fig. 10 shows the training with device-to-device variations under different NL means, where the standard deviation is set to 0.5, 1 and 1.5 (σ=0.5, 1, 1.5) respectively. The results show that, with momentum optimization, the device-to-device variation will not affect the accuracy significantly. When the asymmetric nonlinearity becomes larger, the device-to-device variation tends to exacerbate the accuracy degradation.

Furthermore, to study the impacts of cycle-to-cycle (C2C) variation, we develop a behavior model in a similar way as the device-to-device variation. The cycle-to-cycle variation is referred to as the variation in conductance change at every programming pulse. Hence, we can express the cycle-to-cycle variation standard deviation (σ) in terms of the percentage of entire weight range. Fig. 11 shows the results of training accuracy with different cycle-to-cycle variations (σ equals to 0.01, 0.03 and 0.05 respectively, implying 1%, 3% and 5% of the entire conductance range) under different asymmetric nonlinearities (NL equals to +1/-1, +3/-3 and +5/-5). With larger cycle-to-cycle variation, the accuracy drops significantly. This is due to the fact that the cycle-to-cycle variation may cause a weight-update in an opposite direction to the desired one defined by the ideal gradients, and thus induce opposite momentum directions.

### *B. Hardware Performance per Epoch*

To perform the hardware estimation, in DNN+NeuroSim V2.0, we generate detailed reports for each epoch in runtime, which include the area, latency and energy breakdown by main components, as well as total and peak latency and energy breakdown by operations. Here "peak" is defined as computation within synaptic array only, and it does not consider operations related to off-chip memory, buffers or interconnect. Across the entire simulation, the reports of each epoch will be generated successively. Meanwhile, several summary reports will also be generated, which only contain main evaluation results, such as the accuracy, energy efficiency and throughput for each epoch, and the distribution (mean and standard deviation) of weights and delta weights of different layers in each epoch. The list of expected reports is shown below, as the default setting of this framework is on VGG-8 and CIFAR-10 dataset with 256 epochs for training, there will be 256 detailed reports in total:

**Under path of DNN+NeuroSim V2.0 framework:**
- **NeuroSim_Results_Each_Epoch (folder):**
  - *Breakdown_Epoch_0.csv*
  - ……
  - *Breakdown_Epoch_256.csv*
- *NeuroSim_Output.csv* ***(summary of hardware performance)***
- *PythonWrapper_Output.csv* ***(summary of online learning accuracy)***
- *Weight_dist.csv*
- *Delta_dist.csv*
- *Input_activity.csv*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2020.3043731, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems



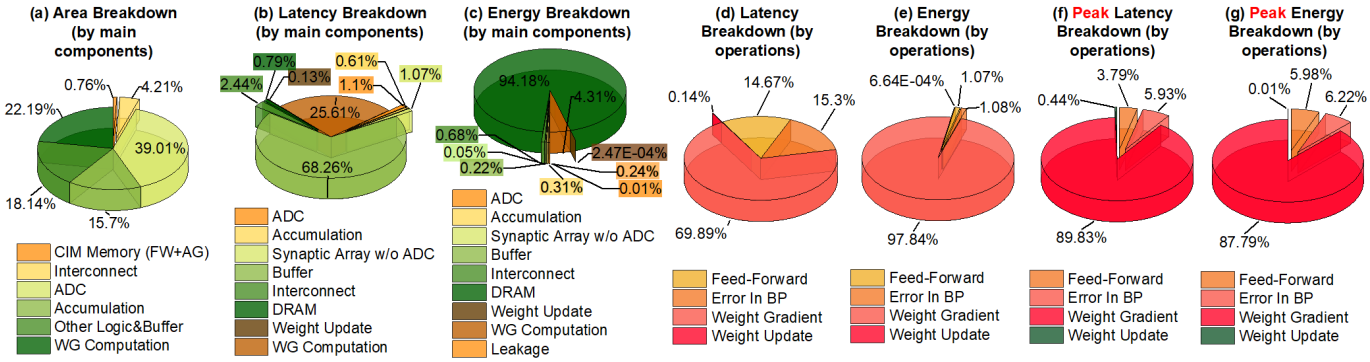


Fig. 12 In CIM accelerator for VGG-8 on CIFAR-10, DNN+NeuroSim V2.0 reports detailed hardware estimation results for each epoch. (a) area breakdown by main components; (b) latency and (c) energy breakdown by main components; (d) latency and (e) energy breakdown by operations; (f) peak latency and (g) peak energy breakdown by operations. The data shown is from the 100th epoch of FeFET-based [6] CIM architecture.

To study the impact of device precision, during technological benchmark, we fix neural network structure, but will change the weight and gradient precision according to different device properties, i.e. we force the weight and gradient precision to be the same as eNVM device precision, because here we just use one single device to represent one synaptic weight. To avoid accuracy drop in ADC quantization, we assume the synaptic array to be 128×128 for all the designs, and use 6-bit ADCs with nonlinear quantization which guarantees sufficiently high-accuracy even for the designs with 7-bit device precision.

Fig. 12 shows the breakdown report in the 100th epoch of the FeFET-based CIM accelerator, where the device parameters follow Ref. [6]. From the area breakdown in Fig. 12 (a), we can find that the 6-bit ADC (Flash-ADC, i.e. multi-level sense amplifiers) is dominant in the total area, while the weight gradient computation units also substantially occupy on-chip area, since they are built up by SRAM-based CIM arrays. Moreover, to support data transfer during each operation, the buffer and control circuits are also quite large. The area of accumulation includes chip-level accumulation units, tile- and PE- level adder trees, and adders or shift-adders in synaptic-arrays.

In the latency and energy breakdown in Fig. 12 (b) and (c), we can find that since there are a lot of data transfer on-chip, the buffer latency and DRAM energy are the bottleneck of the hardware performance. Another way to analyze the contributions is to breakdown the total latency and energy into four main operations. In Fig. 12 (d) and (e), we see that the contributions from feed-forward and computation of error are quite similar, since their operation schemes, volume of input/output data and computations, and utilized hardware resources are quite similar (through the transposable CIM array). While the computation of weight gradients dominate in the total latency and energy, since we need to load in the activations and errors from off-chip memory, and frequently write into the SRAM-based CIM arrays for computation, then send the weight gradients back to off-chip memory. These repeated operations of off-chip memory access and SRAM write make the weight gradient computation to be the bottleneck in the entire training.

On the other hand, the latency and energy of weight update are negligible. This is because we need to operate the prior operations for every input, but we only need to update the weights once per batch. In other words, with the batch size = 200 in this case study, it means the latency and energy of weight update are amortized by 200× in each epoch, i.e. total latency and energy for each batch equals to (200 × feed-forward) + (200 × computation of errors) + (200 × computation of weight gradients) + (1 × weight update). In Fig. 12 (f) and (g), we see that the peak latency and energy breakdown by operations. Similarly, the computation of weight gradients contributes most

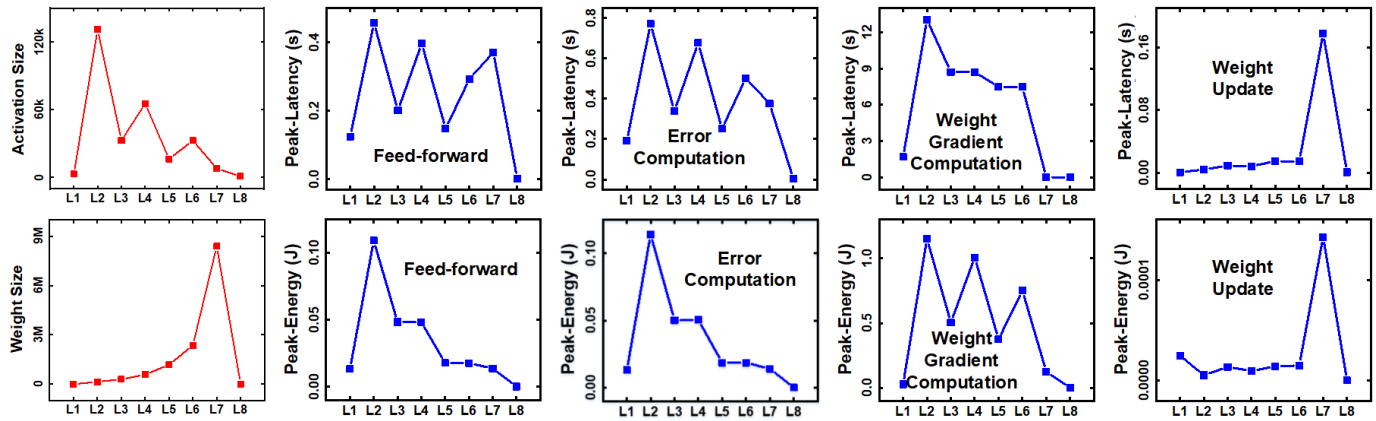


Fig. 13. Peak latency and energy across all the layers in VGG-8 of four main operations (feed-forward, computation of errors, computation of weight gradients and weight update) of one epoch. The data shown is from the 100th epoch of FeFET-based [6] CIM architecture.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2020.3043731, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems



of the latency and energy, as it is based on SRAM and also includes repeated write operations.

In Fig. 13, the peak latency and energy of four main operations are shown across all the layers in the VGG-8 (blue lines and symbols), to analyze the relationship among layers. We also draw the size of activations and weights for each layer (red lines and symbols) as reference. In feed-forward and computation of errors, the latency and energy are in the same trend as the activation size (across the layers), which is reasonable since the activation size determines the number of operations in each layer. In the computation of weight gradients, the energy's trend fits with the activation size, while the latency trend is a little different. This is because in deeper layer, the activation size is smaller, so there are possibilities to compute more gradients simultaneously. As mentioned in section III-A and III-D, if we assume the SRAM-based CIM arrays are large enough to support the largest layer in the entire network and the layers whose unrolled errors' sizes are smaller than the SRAM-based CIM arrays, we will duplicate the errors in SRAM-based CIM arrays, fetch multiple activations simultaneously and speed up the computation. In DNN+NeuroSim V2.0, the NeuroSim core can automatically define the duplication strategy and take the "speed-up" into account. Finally, the latency and energy of weight update are proportional to the weight size.

### C. *Hardware Performance across Epoch*

To explore the hardware performance during the entire training process, we track the data (weight and gradient distributions) and the real-time estimations for every epoch. The feed forward and error computation employed the real-trace of activations and updated weights (every layer) for each epoch from the python wrapper.

It should be noted that, to limit the simulation time in DNN+NeuroSim V2.0, we applied a "pseudo-traced" method to estimate the hardware performance of weight gradient computation. During estimation, we access to the binarized activations (quantized fix-point to digital format) and approximate the percentage of ones for each layer, which tends to represent the input-activity of the SRAM-based CIM arrays. Similarly, we can estimate the percentage of ones stored in the SRAM-based CIM arrays (as binarized errors). While for weight update, we access the old weights and updated weights (every layer) for each epoch, to process a real-traced estimation. In this way, we only need to access the activations, old weights and updated weights (every layer) at end of each epoch, to avoid huge data access in the framework during simulation, and still guarantee reliable estimation.

Fig. 14 shows an example of the traced hardware performance of each epoch during *in-situ* training. The data is extracted from the FeFET-based [6] CIM architecture design.

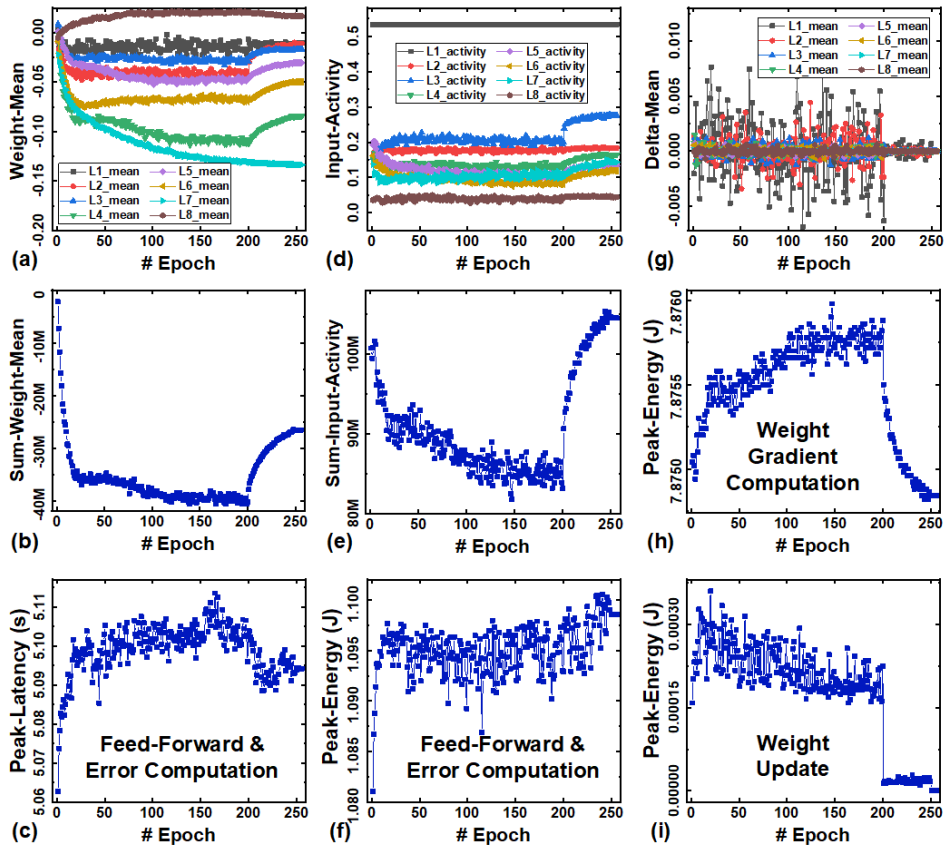


Fig. 14. (a) Weight means of each layer across the epochs. (b) Weight means multiplied with activation and weight size of each layer across the epochs. (c) Peak latency of feed-forward and error computation across the epochs. (d) Input activity of each layer across all the epochs. (e) Input activities multiplied with activation and weight size of each layer across the epochs. (f) Peak energy of feed-forward and error computation across the epochs. (g) Delta weight means of each layer across the epochs. (h) Peak energy of weight gradient computation across the epochs. (i) Peak energy of weight update across epochs. The data shown is from the FeFET-based [6] CIM architecture.

| VGG-8 on CIFAR10, with Novel Weight Mapping and Dataflow, on-chip training with 256 epochs | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Technology node (LSTP) | 7 nm | | 32 nm | | | | | | | |
| Device | SRAM | | SRAM | | Ag:a-Si [23] | PCMO [24] | AlOx/HfO$_2$ [25] | TaOx/HfOx (TsingHua) [2] | EpiRAM [4] | HZO FeFET (NotreDame) [6] |
| # of Conductance States | \ | \ | \ | \ | 97 | 50 | 40 | 128 | 64 | 32 |
| ADC precision | Sequential | 4-bit | Sequential | 4-bit | 6-bit | | | | | |
| Weight / ΔWeight / Cell Precision | 5-bit / 5-bit / 1-bit | | 5-bit / 5-bit / 1-bit | | 6-bit | 5-bit | 5-bit | 7-bit | 6-bit | 5-bit |
| Ron (Ω) | \ | \ | \ | \ | **26 M** | **23 M** | **16.9 K** | **100 K** | **81 K** | **240 K** |
| On/Off Ratio | \ | \ | \ | \ | 12.5 | 6.84 | 4.43 | 10 | 50.2 | 100 |
| Nonlinearity | \ | \ | \ | \ | **2.4/-4.88** | **3.58/-6.76** | 1.94/-0.61 | 0.04/-0.63 | 0.5/-0.5 | 1.75/1.46 |
| C2C Variation | \ | \ | \ | \ | **3.50%** | <1% | **5.00%** | **3.70%** | **2.00%** | <0.5% |
| Write Pulse Voltage | \ | \ | \ | \ | 3.2V/-2.8V | -2V/2V | 0.9V/-1V | 1.6V/1.5V | 5V/-3V | 3.65V/-2.95V |
| Write Pulse Width | \ | \ | \ | \ | **300us/300us** | **1ms/1ms** | **100us/100us** | **50ns/50ns** | 5us/5us | **75ns/75ns** |
| Area (mm$^2$) | **11.12** | **13.05** | **120.89** | **138.95** | **48.29** | **48.29** | **49.88** | **48.50** | **48.59** | **48.29** |
| Memory Utilization (%) | 94.62% | | | | 88.59% | | | | | |
| Training Accuracy (%) | **91.00%** | | | | **49.00%** | **56.00%** | **37.00%** | **81.00%** | **85.00%** | **91.00%** |
| Training Latency (s) / Epoch | 244.50 | 105.51 | 655.32 | 235.75 | 1241.63 | 5795.79 | 611.00 | 177.61 | 193.94 | 176.85 |
| Training Dynamic Energy (J) / Epoch | 105.60 | 80.79 | 149.36 | 95.37 | 92.12 | 92.15 | 93.13 | 92.15 | 92.28 | 92.11 |
| Training Peak Latency (s) / Epoch | 177.92 | 37.18 | 471.43 | 55.33 | 1116.53 | 5670.69 | 485.89 | 52.52 | 68.83 | 51.75 |
| Training Peak Dynamic Energy (J) / Epoch | 14.15 | 4.51 | 113.41 | 15.42 | 9.00 | 9.02 | 10.70 | 9.05 | 9.19 | 8.98 |
| Training Throughput (TOPS) | **0.75** | **1.75** | **0.28** | **0.78** | **0.14** | **0.03** | **0.30** | **1.04** | **0.95** | **1.04** |
| Training Energy Efficiency (TOPS/W) | **1.74** | **2.28** | **1.24** | **1.94** | **2.00** | **2.00** | **1.98** | **2.00** | **2.00** | **2.01** |
| Training Peak Throughput (TOPS) | **1.04** | **4.96** | **0.40** | **3.34** | **0.16** | **0.03** | **0.38** | **3.52** | **2.68** | **3.57** |
| Training Peak Energy Efficiency (TOPS/W) | **13.03** | **40.90** | **1.63** | **11.98** | **20.54** | **20.50** | **17.27** | **20.43** | **20.11** | **20.57** |

Table I. Benchmark results of CIM accelerators training on VGG-8 for CIFAR10, based on SRAM (both sequential and parallel read-out at 7nm and 32nm), and reported "analog" synaptic devices (assumed at 32nm technology). Green bold values shows the good specs and performance.

In Fig. 14 (a), we track the mean value of each layer's weights. Since the contribution of the weights from different layer is also determined by the weight size (defining amount of hardware resources) and activation size (defining number of computations). In Fig. 14 (b), we sum up the weight means of entire network, by multiplying the activation and weight size with the weight means of corresponding layer, and summing them up. Similarly, we track the input activities of each layer across epochs, and sum up according to the activation and weight size in Fig. 14 (d) and (e). In this case study, at 200$^{th}$ epoch, the learning rate is tuned for convergence.

Now we will study the correlation between the summed weights and input activities and the peak latency/energy across epochs. The peak latency and energy of feed-forward and error computation are shown in Fig. 14 (c) and (f). In CIM architectures, the synaptic weights are proportional to the conductance values. Before 200$^{th}$ epoch, the summed weight means gradually decrease, thus the overall conductance decreases. The reduced input activities also induce smaller currents along the columns. Both effects will cause longer propagation latency per epoch as shown in Fig. 14 (c). After 200$^{th}$ epoch, both the summed weight means and input activities increase, the propagation latency decreases. The same reason explain the peak energy trend. Before 200$^{th}$ epoch, the column currents or the power become smaller but the ADC delays tend to increase more rapidly, thus overall energy slightly increase. After 200$^{th}$ epoch, with increased input activities and larger column currents or power, the energy per epoch keeps increasing further despite slightly reduced latency, as shown in Fig. 14 (f).

We also track the delta weights (i.e., the weight changes) in Fig. 14 (g). Due to the tuned learning rate, the delta weights means are close to zero after 200$^{th}$ epoch as the network is converging. This also explains the trends of substantially reduced peak energy of weight gradient computation and weight update in Fig. 14 (h) and (i).

### *D. Benchmark across Technologies*

Finally, we benchmark the CIM accelerators for VGG-8 training on CIFAR-10 dataset, with versatile synaptic devices, including the sequential and parallel read-out SRAM-based accelerators at both 7nm and 32nm, and state-of-the-art parallel read-out eNVM-based (including [2][4][6], and [23~25]) accelerators at 32nm.

For versatile "analog" synaptic devices, the number of conductance varies a lot, i.e. from 32 to 128 levels, thus we change the precision of weights and gradients in the network for different devices according to their device precision. As a result, we have run different network specs from 5-bit (32 levels) to 7-bit (128 levels) for weight and gradient precision. The results in Table I show that with momentum optimization method, and relatively low cycle-to-cycle variation, the FeFET-based design could achieve quite high accuracy (~91%) even with 32 levels. As a reference, we set the digital SRAM-based designs based on 5-bit (5 SRAM cells to represent one weight). The factors that limit the training accuracy on other devices are either too large asymmetric nonlinearities [23][24] or too large cycle-to-cycle variations [2][4][25].

From the hardware performance results shown in Table I, we find that: (1) on-state resistance ($R_{on}$) still plays important role to achieve better "peak" performance. To avoid large voltage drop, the transistors in 1T1R or peripheral mux have to be sized up for small $R_{on}$, yielding significant area overhead. As a result, it takes longer time to activate the synaptic arrays due to the increased capacitance loading, adversely increasing latency and lowering throughput. (2) When write pulse width is short, i.e. below a micro-second (μs), the weight update will not affect the speed, as the operation is amortized by the batch size. (3) The cycle-to-cycle variation is critical factor for *in-situ* training accuracy, as large variation could lead to opposite momentum move and hamper the model from convergence. A preferred cycle-to-cycle variation is lower than 1%. (4) At the same

technology node, the SRAM-based designs suffer from leakage and area overhead, while at advanced 7nm, the parallel-read SRAM design still shows superior energy efficiency and throughput. (5) DRAM access dominates the overall energy efficiency, resulting in ~2 TOPS/W regardless of device technologies.

## V. Conclusion

In this paper, we presented DNN+NeuroSim V2.0, an end-to-end framework to benchmark CIM-based architectures for on-chip training, and support flexible network structures with versatile device options. With the behavior model of nonlinearity and asymmetry, device-to-device and cycle-to-cycle variations during weight update, it is able to investigate the non-ideal properties of "analog" synaptic devices in *in-situ* training. Momentum optimization is introduced to mitigate the impact of asymmetric nonlinearities on accuracy. From the technological benchmark, it reveals that the desired specs for "analog" synaptic devices are: low cycle-to-cycle variation (<1%), large on-state resistance (>100 kΩ), small write pulse width (<1μs), with nonlinearity below +3/-3. Currently, one limitation of the framework is that one specific chip architecture will be customized to implement a pre-defined neural network topology. The potential improvement for the future is to consider the hardware reconfiguration where a fixed chip architecture could support different neural network topologies with a dynamic dataflow (e.g. via network-on-chip).

## Appendix

The prior DNN+NeuroSim V1.2 framework for inference-only will approximately take 5 minutes for the VGG-8 on CIFAR-10 (pre-training model requires ~5 hours before inference simulation), on a computer workstation (Intel 8-core CPU with 3.2 GHz and NVidia Titan V GPU). The proposed DNN+NeuroSim V2.0 framework for on-chip training (with non-ideal weight update in back-propagation, without ADC quantization) will approximately take 12 hours for the VGG-8 on CIFAR-10, on the same computer workstation, with default setting (batch-size=200, 256 epochs in total), while the original algorithm (without hardware constraints) takes ~5 hours for training. This is because the NeuroSim core (operates in CPU) takes 60~90 seconds per run (one run per epoch in default setting), while the non-ideal weight-update functions in python wrapper also takes longer time.

With ADC quantization included in feed-forward operation, the running time will be ~72 hours, and with ADC quantization in both feed-forward and back-propagation, the running time will be ~2 weeks. This long running time is due to the unrolled computation of convolutional layers (binarized input from LSB to MSB, and array partitioning) in python wrapper, which is used to mimic the partial-sum quantization via ADCs in synaptic arrays. This induces a set of deep loops in computation. To provide a generic framework for versatile networks, we use this naïve loop-control method, however, the running time can be improved with some loop optimization methods (e.g. loop tiling) for a specific network. Besides, it is also possible to firstly decide the ADC precision with a fixed set of design options (e.g. synaptic array size and cell precision), which guarantees the computation accuracy in feed-forward/back-propagation, then turn off the ADC quantization feature in python wrapper, to avoid overlong running time for multiple simulations.

| Scalability | VGG-8 | DenseNet -40-12 | DenseNet -100-12 | ResNet -18 | AlexNet |
|---|---|---|---|---|---|
| Dataset | CIFAR-10 | | CIFAR-100 | ImageNet | |
| # of param * | 13M | 1M | 7M | 11M | 60M |
| # of OPs | 0.7M | 0.01G | 0.04G | 1.82G | 0.72G |
| min/epoch | ~3 | ~4 | ~10 | ~35 | ~35 |
| # of epochs | 256 | 300 | 300 | 90 | 90 |
| Total time (hr) | ~13 | ~20 | ~50 | ~53 | ~53 |

* Parameters are quantized to 8-bit

Table II. Run-time performance of DNN+NeuroSim V2.0 on selected popular DNNs.

Table II shows the run-time performance of DNN+NeuroSim V2.0 for various popular networks (on Intel 8-core CPU with 3.2 GHz and NVidia Titan V GPU). Overall, the runtime could vary a lot with different platforms (GPU/CPU settings) and model specs (e.g. network structure, batch size, epoch numbers, synaptic array size, synaptic weight and synaptic cell precision, and so on), while the users are encouraged to optimize and speed up the framework on their owns.

## Acknowledgment

This work is supported by ASCENT, one of the SRC/DARPA JUMP centers, NSF/SRC E2CDA program, and NSF-CCF-1903951.

## References

[1] S. Yu, "Neuro-inspired computing with emerging non-volatile memory," *Proc. IEEE*, vol. 106, no. 2, pp. 260-285, 2018.

[2] W. Wu *et al.*, "A methodology to improve linearity of analog RRAM for neuromorphic computing," *IEEE Symposium on VLSI Technology*, Honolulu, HI, 2018, pp. 103-104.

[3] W. Kim *et al.*, "Confined PCM-based analog synaptic devices offering low resistance-drift and 1000 programmable states for deep learning," *IEEE Symposium on VLSI Technology*, Kyoto, Japan, 2019, pp. T66-T67.

[4] S. Choi, S. Tan, Z. Li, Y. Kim, C. Choi, P. Chen, H. Yeon, S. Yu and J. Kim. "SiGe epitaxial memory for neuromorphic computing with reproducible high performance based on engineered dislocations." *Nature Materials*, vol. 17, no. 4 335-340, 2018.

[5] J. Tang *et al.*, "ECRAM as scalable synaptic cell for high-speed, low-power neuromorphic computing," *2018 IEEE International Electron Devices Meeting (IEDM)*, San Francisco, CA, 2018, pp. 13.1.1-13.1.4.

[6] M. Jerry *et al.*, "Ferroelectric FET analog synapse for acceleration of deep neural network training," *2017 IEEE International Electron Devices Meeting (IEDM)*, San Francisco, CA, 2017, pp. 6.2.1-6.2.4, doi: 10.1109/IEDM.2017.8268338.

[7] X. Peng, S. Huang, Y. Luo, X. Sun and S. Yu, "DNN+NeuroSim: An end-to-end benchmarking framework for compute-in-memory accelerators with versatile device technologies," *IEEE International Electron Devices Meeting (IEDM)*, San Francisco, CA, USA, 2019, pp. 32.5.1-32.5.4.

[8] S. Wu, G. Li, F. Chen and L. Shi, "Training and Inference with Integers in Deep Neural Networks," *International Conference on Learning Representations (ICLR)*, 2018.

[9] P. Chen and S. Yu, "Reliability perspective of resistive synaptic devices on the neuromorphic system performance," *IEEE International Reliability Physics Symposium (IRPS)*, Burlingame, CA, 2018, pp. 5C.4-1-5C.4-4.

[10] X. Peng, R. Liu and S. Yu, "Optimizing weight mapping and data flow for convolutional neural networks on RRAM based processing-in-memory architecture," *IEEE International Symposium on Circuits and Systems (ISCAS)*, Sapporo, Japan, 2019, pp. 1-5.
[11] ASU PTM model, http://ptm.asu.edu/
[12] P.-Y. Chen, X. Peng, S. Yu, "NeuroSim: A circuit-level macro model for benchmarking neuro-inspired architectures in online learning," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 12, pp. 3067-3080, 2018.
[13] H. Jiang, S. Huang, X. Peng, J.-W. Su, Y.-C. Chou, W.-H. Huang, T.-W. Liu, R. Liu, M.-F. Chang, S. Yu, "A two-way SRAM array based accelerator for deep neural network on-chip training," *ACM/IEEE Design Automation Conference (DAC)* 2020.
[14] H. Jiang, X. Peng, S. Huang, S. Yu, "MINT: Mixed-precision RRAM-based in-memory training architecture," *IEEE International Symposium on Circuits and Systems (ISCAS)* 2020.
[15] H. Jiang, X. Peng, S. Huang and S. Yu, "CIMAT: A compute-in-memory architecture for on-chip training based on transpose SRAM arrays," in *IEEE Transactions on Computers*, vol. 69, no. 7, pp. 944-954, July 2020.
[16] X. Peng, R. Liu and S. Yu, "Optimizing weight mapping and data flow for convolutional neural networks on processing-in-memory architectures," in *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 4, pp. 1333-1343, April 2020.
[17] P.-Y. Chen, S. Yu, "Partition SRAM and RRAM based synaptic arrays for neuro-inspired computing," *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2016.
[18] J. Seo *et al.*, "A 45nm CMOS neuromorphic chip with a scalable architecture for learning in networks of spiking neurons," *IEEE Custom Integrated Circuits Conference (CICC)*, San Jose, CA, 2011, pp. 1-4.
[19] S. Huang, X. Sun, X. Peng, H. Jiang, S. Yu, "Overcoming challenges for achieving high in-situ training accuracy with emerging memories," *IEEE/ACM Design, Automation & Test in Europe (DATE)* 2020.
[20] S. Sidler *et al*., "Large-scale neural networks implemented with non-volatile memory as the synaptic weight element: Impact of conductance response," *IEEE European Solid-State Device Research Conference (ESSDERC)*, Lausanne, 2016, pp. 440-443.
[21] A. Fumarola *et al*., "Accelerating machine learning with non-volatile memory: Exploring device and circuit tradeoffs," *IEEE International Conference on Rebooting Computing (ICRC)*, San Diego, CA, 2016, pp. 1-8.
[22] G. W. Burr *et al*., "Experimental demonstration and tolerancing of a large-scale neural network (165 000 synapses) using phase-change memory as the synaptic weight element," *IEEE Transactions on Electron Devices*, vol. 62, no. 11, pp. 3498-3507, Nov. 2015.
[23] S. H. Jo, T. Chang, I. Ebong, B. B. Bhadviya, P. Mazumder, and W. Lu, "Nanoscale memristor device as synapse in neuromorphic systems," *Nano Letters*, vol. 10, no. 4, pp. 1297-1301, 2010.
[24] S. Park, A. Sheri, J. Kim, J. Noh, J. Jang, M. Jeon, B. Lee, B. R. Lee, B. H. Lee, and H. Hwang, "Neuromorphic speech systems using advanced ReRAM-based synapse," *IEEE International Electron Device Meeting (IEDM)*, pp. 625-628, 2013.
[25] J. Woo, K. Moon, J. Song, S. Lee, M. Kwak, J. Park, and H. Hwang, "Improved synaptic behavior under identical pulses using $AlO_x/HfO_2$ bilayer RRAM array for neuromorphic systems," *IEEE Electron Device Letters*, vol. 37, no. 8, pp. 994-997, 2016.

**Xiaochen Peng** (S'17) received the B.S. degree in Automatic System from Hefei University of Technology in 2014 and the M.S. degree in electrical engineering from Arizona State University in 2016. She is currently pursuing the Ph.D. degree in electrical and computer engineering at the Georgia Institute of Technology in Atlanta, Georgia. Her research interests include development of device-to-system benchmarking framework for machine learning accelerators, and design of emerging-device-based hardware implementation for neural networks.

**Shanshi Huang** received her B.S. degree in communication engineering from Beijing Institute of Technology in 2012 and the M.S. degree in electrical engineering from Arizona State University in 2014. She is currently pursuing the Ph.D. degree in electrical and computer engineering at the Georgia Institute of Technology in Atlanta, Georgia. Her current research interests include Deep learning algorithm & hardware co-design and deep learning security.

**Hongwu Jiang** received the B.S. degree from Dalian University of technology in 2012 and the M.S. degree in electrical engineering from Arizona State University in 2014. He is currently working towards the PhD degree in electrical and computer engineering at the Georgia Institute of Technology in Atlanta, Georgia. His research interests include SRAM-/eNVM- based hardware architecture and accelerator design of deep learning.

**Anni Lu** received the B.S. degree in electronic information engineering from Tianjin University in 2019. She is currently pursuing the Ph.D. degree in electrical and computer engineering with Georgia Institute of Technology, Atlanta, GA, USA. Her current research interests include deep learning algorithms and hardware co-design for in-memory computing and hardware accelerator.

**Shimeng Yu** (M14'-SM19') is an associate professor of electrical and computer engineering at the Georgia Institute of Technology. He received the B.S. degree in microelectronics from Peking University in 2009, and the M.S. degree and Ph.D. degree in electrical engineering from Stanford University in 2011 and 2013, respectively. From 2013 to 2018, he was an assistant professor at Arizona State University. Prof. Yu's research interests are nanoelectronic devices and circuits for energy-efficient computing systems. His expertise is on the emerging non-volatile memories for different applications such as deep learning accelerator, neuromorphic computing, monolithic 3D integration, and hardware security.