User Manual of DNN+NeuroSim Framework V2.0

Developers: Xiaochen Peng and Shanshi Huang

PI: Prof. Shimeng Yu, Georgia Institute of Technology

Mar 1, 2020

Index

1. Introduction 2
2. System Requirements (Linux) 2
3. Installation and Usage (Linux) 3
4. Chip Level Architectures 5
   4.1 Interconnect: H-Tree 6
   4.2 Floorplan of Neural Networks 7
   4.3 Weight Mapping Methods 8
   4.4 Pipeline System 10
5. Circuit Level: Synaptic Array Architectures 11
   5.1 Parallel Synaptic Array Architectures 11
   5.2 Transposable Synaptic Arrays 14
   5.3 Weight Gradient Calculation Unit 16
   5.4 Synaptic Array Peripheral Circuits 17
6. Algorithm Level: PyTorch and TensorFlow Wrapper 22
7. Algorithm Level: Inference Accuracy Estimation in V1.1 23
8. Algorithm Level: On-chip Training Accuracy Estimation in V2.0 27
9. How to run *DNN +NeuroSim* 30
10. Reference 34

---

1. Introduction

DNN+NeuroSim is an integrated framework, which is developed in C++ and wrapped by Python to emulate the deep neural networks (DNN) inference performance (in V1.1) or on-chip training (in V2.0) performance on the hardware accelerator based on near-memory computing or in-memory computing architectures. Various device technologies are supported, including SRAM, emerging non-volatile memory (eNVM) based on resistance switching (e.g. RRAM, PCM, STT-MRAM), and ferroelectric FET (FeFET). SRAM is by nature 1-bit per cell, eNVMs and FeFET in this simulator could support either 1-bit or multi-bit per cell. *NeuroSim* [1] is a circuit-level macro model for benchmarking neuro-inspired architectures (including memory array, peripheral logic, and interconnect routing) in terms of circuit-level performance metrics, such as chip area, latency, dynamic energy and leakage power. With Pytorch and TensorFlow wrapper, *DNN +NeuroSim* framework can support hierarchical organization from the device level (transistors from 130 nm down to 7 nm, eNVM and FeFET device properties) to the circuit level (periphery circuit modules such as analog-to-digital converters, ADCs), to chip level (tiles of processing-elements built up by multiple sub-arrays, and global interconnect and buffer) and then to the algorithm level (different convolutional neural network topologies), enabling instruction-accurate evaluation on the inference or training accuracy as well as the circuit-level performance metrics at the run-time of inference or training.

The target users for this simulator are circuit/architecture designers who wish to quickly estimate the system-level performance with different network and hardware configurations (e.g. device technology choices, sequential read-out or parallel read-out, etc.). Different from our earlier released simulators (*MLP+NeuroSim* [2]), where the network was fixed to a 2-layer MLP and executed purely in C++, this *DNN+NeuroSim* framework is an integrated simulator with Pytorch and TensorFlow wrapper (i.e. C++ wrapped by python). With the wrapper, users are able to define various network structures, precisions of synaptic weights and neural activations, which guarantees efficient inference or training with the popular machine learning platforms. Meanwhile, the wrapper will automatically save the real traces (synaptic weights and neural activations) during the inference, and send to *NeuroSim* for real-time and real-traced hardware estimation. In the released simulator, an 8-layer VGG (VGG-8) network for CIFAR-10 dataset is provided as a default model in the wrapper, with 8-bit synaptic weights and neural activations, while users could modify the precisions and neural network topologies (extendable to ResNet on ImageNet). The hardware parameters (such as technology nodes, memory cell properties, operation modes, and so on) will be defined under *NeuroSim* in **Param.cpp**.

---

2. System Requirements (Linux)

The tool is expected to run in Linux with required system dependencies installed. These include GCC, GNU make, GNU C libraries (glibc). We have tested the compatibility of the tool with a few different Linux environments, such as (1) Red Hat 5.11 (Tikanga), gcc v4.7.2, glibc 2.5, (2) Red Hat 7.3 (Maipo), gcc v4.8.5, glibc v2.1.7, (3) Ubuntu 16.04, gcc v5.4.0, glibc v2.23, and they are all workable.

※ The tool may not run correctly (stuck forever) if compiled with gcc 4.5 or below, because some C++11 features are not well supported.

---

3. Installation and Usage (Linux)

Step 1: Get the tool from GitHub

```
git clone https://github.com/neurosim/DNN_NeuroSim_V2.0.git
```

Step 2: Set up hardware parameters in NeuroSim Core and compile the Code

Set up hardware parameters in "Param.cpp" as below:

```
/*** parameters for SRAM ***/
heightInFeatureSizeSRAM = 8;          // SRAM Cell
widthInFeatureSizeSRAM = 20;          // SRAM Cell
widthSRAMCellNMOS = 2.08;
widthSRAMCellPMOS = 1.23;
widthAccessCMOS = 1.31;
minSenseVoltage = 0.1;

/*** parameters for analog synaptic devices ***/
heightInFeatureSize1T1R = 4;          // 1T1R Cell
widthInFeatureSize1T1R = 9;           // 1T1R Cell
heightInFeatureSizeCrossbar = 2;      // Crossbar C
widthInFeatureSizeCrossbar = 2;       // Crossbar C

resistanceOn = 20e3;                 // Ron resista
resistanceOff = 20e3*10;             // Roff resist
maxConductance = (double) 1/resistanceOn;
minConductance = (double) 1/resistanceOff;
maxNumLevelLTP = 128;                 // Maximum nu
maxNumLevelLTD = 128;                 // Maximum nu
writeVoltage = 1.5;
writePulseWidth = 60e-9;
nonlinearIV = false;                  // This optio
nonlinearity = 10;                    // This is th

readVoltage = 0.5;                    // On-chip re
readPulseWidth = 10e-9;               // read pulse
accessVoltage = 1.1;                  // Gate volta
resistanceAccess = 15e3;              // resistance

/****** design options for on-chip training ******/
/****** in training: we initialize bi-direction subArray
/****** the gradient calculation of weight is processed i

trainingEstimation = true;       // false: only run estimat
                                    // true: run estimatic

parallelBP = true;                  // false: conventional
                                    // true: conventionalI

batchSize = 200;                    // batchSize in traini
numIteration = 250;                 // num of iteration fc

bufferOverHeadConstraint = 1;       // N times of overhead
                                    // For example: if N=

numRowSubArrayWG = 128;             // # of rows of single
numColSubArrayWG = 128;             // # of columns of sir

numRowMuxedAG = 8;                  // How many columns sh
levelOutputAG = 64;                 // # of levels of the
numRowMuxedWG = 8;                  // How many columns sh
levelOutputWG = 64;                 // # of levels of the

activityRowReadWG = 0.5;            // activity of row dur
activityRowWriteWG = 0.5;           // activity of row dur
activityColWriteWG = 0.5;           // activity of col dur

dramType = 2;                       // 1: GDDR5
                                    // 2: HBM2
```

Fig. 1. Example of argument in "Param.cpp" for hardware estimation in NeuroSim core.

>> make (compile code)

Step 3: Set up hardware constraints in Python wrapper

Modify arguments in "train.py" and "inference.py" as below:

```
parser.add_argument('--onoffratio', default=10)
parser.add_argument('--cellBit', default=1)
parser.add_argument('--subArray', default=128)
parser.add_argument('--ADCprecision', default=5)
parser.add_argument('--vari', default=0)
parser.add_argument('--t', default=0)
parser.add_argument('--v', default=0)
parser.add_argument('--detect', default=0)
parser.add_argument('--target', default=0)
parser.add_argument('--nonlinearityLTP', default=0.01)
parser.add_argument('--nonlinearityLTD', default=-0.01)
parser.add_argument('--max_level', default=100)
parser.add_argument('--d2dVari', default=0)
parser.add_argument('--c2cVari', default=0)
```

```
args.wl_weight = 5
args.wl_grad = 5
args.wl_error = 5
args.cellBit = 5
args.max_level = 32
args.c2cVari = 0.003
args.d2dVari = 0.0
args.nonlinearityLTP = 1.75
args.nonlinearityLTD = 1.46
# momentum
gamma = 0.9
alpha = 0.1
```

Fig. 2. Example of argument in "train.py" for training accuracy estimation with hardware constraints.

Step 4: Run Python wrapper (integrated with *NeuroSim*)

>> python train.py (expect simulation results as below)

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***

>> python inference.py (for inference engine benchmarking as DNN+NeuroSim V1.1)

Summary of the useful commands is provided below. It is recommended to execute these commands under the tool's directory.

Command	Description
`make`	Compile the *NeuroSim* codes and build the "main" program
`make clean`	Clean up the directory by removing the object files and the "main" executable

※ The simulation uses OpenMP for multithreading, and it will use up all the CPU cores by default.

※ The wrapper is built under the CUDA 9.0 + cuDNN v7.0.5, python2.7 + tensorflow 1.5.0 (GPU) and python 3.5 + pytorch 1.0(GPU).

---

4. Chip Level Architectures

In this framework, we consider the on-chip memory is sufficient to store synaptic weights of the entire neural network, thus the only off-chip memory access is to fetch in the input data. Fig. 3 shows the modeled chip hierarchy for on-chip training, where the top level of chip is consist of multiple tiles, global buffer, neural functional computation units including accumulation units activation units (sigmoid or ReLU) and pooling units, as well as the weight gradient computation unit. In each tile, it contains several processing elements (PEs), tile buffer to load in neural activations, accumulation modules to add up partial sums from PEs and output buffer. Similarly, a PE is built up by a groups of synaptic sub-arrays, PE buffers, accumulation modules and output buffer.

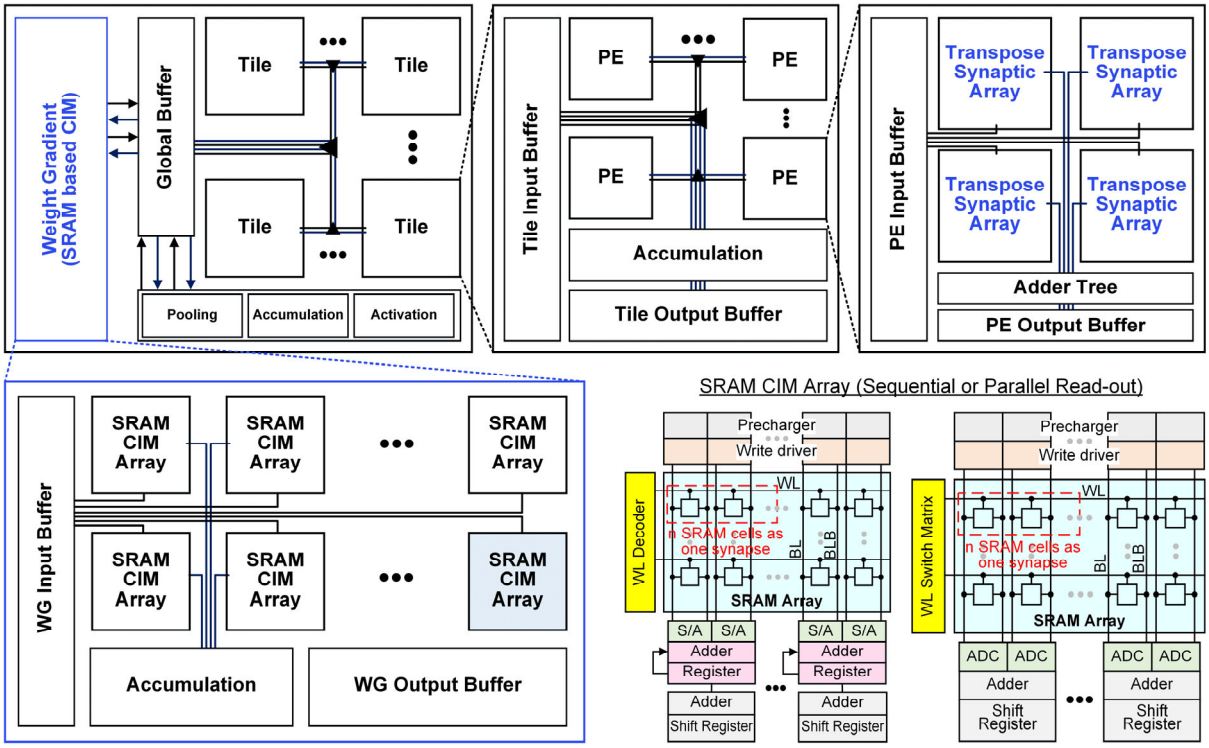


Fig. 3. Architecture structure defined in the simulator, the top level of chip contains tiles, global buffer, neural-functional peripheries (including pooling, accumulation and activations) and weight gradient computation function which is mainly built up with a group of SRAM-based CIM arrays. Inside 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 on-chip training, the synaptic arrays are designed "transposable".

For the weight gradient computation function, we define a group of SRAM-based CIM arrays, local buffers and accumulation units. The reason that we choose the SRAM-based CIM arrays is, we need to frequently write data into the array and do vector-matrix multiplication. Although SRAM is not as area-efficient as eNVMs, and also has the problem of standby leaking, its fast writing performance still makes it more suitable and infinite endurance for the gradient computation compared with the eNVMs. More details of the weight gradient computation function will be discussed in ***Section 5.3***.

Moreover, since normally mini-batch based training with batch size=B is used, it means the number of intermediate data that need to be utilized and stored is huge. For example, during feed-forward, the B

activations of all the layers will be stored to be used for the computation of weight gradients later in backpropagation; the B computation of errors of all the layers obtained in backpropagation will be stored; and the B weight gradients in one batch will be stored and accumulated to produce the delta weight for the final weight-update. Therefore, if the batch size is B, we have to store B copies of activations, errors and weight gradients of the entire network, before we can update the weights for a specific batch. To limit the on-chip buffer overhead, we assume that those data will be sent back to off-chip DRAM [3] memory for the entire batch, and will be retrieved back to chip for activation and weight gradient computation. More details about the on-chip training scheduling can be find in arXiv paper [4].

4.1 Interconnect: H-Tree

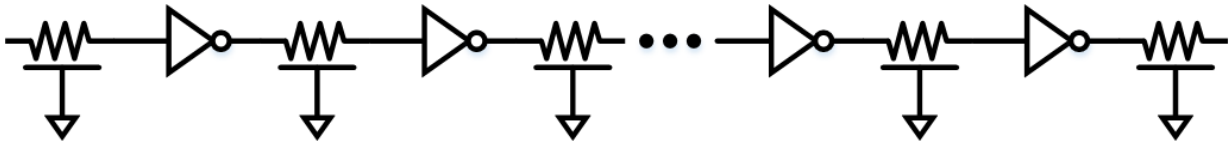


Fig. 4. The diagram of wire with repeaters.

To estimate the area, latency, dynamic energy and leakage of interconnect, we assume the routing among modules in each hierarchy is based on H-tree structure. According to the interconnect engineering, the wire delay could be reduced by introducing repeaters which is used to split the wire into multiple segments. As Fig. 4 shows, a wire could be considered as a group of wire segments and repeaters, to find an optimal length of wire segment between repeaters, which leads to minimum delay, a VLSI design function [5] is introduced as EQ (4.1) shows, where R is the resistance of a minimum-sized repeater, C is the gate capacitance, and diffusion capacitance Cp_{inv}, R_w and C_w are the unit resistance and capacitance of wire, respectively.

$$L_{optimal} = \sqrt{\frac{2RC(1+p_{inv})}{R_w C_w}} \tag{4.1}$$

The repeater size should use an NMOS transistor width of

$$W = \sqrt{\frac{RC_w}{R_w C}} \tag{4.2}$$

However, in practice, to limit the energy consumption of interconnect, we may find a semi-optimal design option of trade-offs between wire latency and energy. In this framework, we introduce two parameter called "globalBusDelayTolerance" (and "localBusDelayTolerance" for global bus and tile/PE local bus respectively) to find the semi-optimal floorplan of bus with such delay sacrifice, which will be defined in **param.cpp**.

Fig. 5 shows an example of H-tree structure for 4 × 4 computation units (either tiles or PEs), where the bus width connected to each units is assumed to be same. We define the H-tree is built up by multiple stages (horizontal and vertical) from the widest (main bus) to the most narrow ones (connected to computation units). The wire length decreases by ×2 at each stage from wide to narrow ones, while the sum of bus width at each stage is fixed, which equals to the width of main bus.

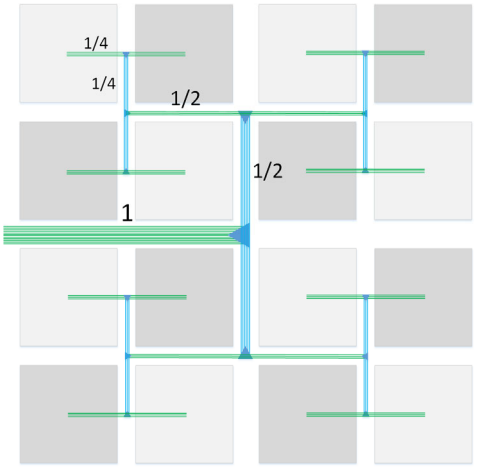


Fig. 5. An example of H-tree for a 4×4 computation-unit array.

4.2 Floorplan of Neural Networks

To map various neural networks according to the defined chip architecture, it is crucial to follow a certain rule which does not violate hardware structure (and data flow) while guarantees high-enough memory utilization. We defined an algorithm to automatically generate the floorplan based on two kinds of weight-mapping methods, which optimize the memory utilization and define the tile size, PE size, number of tiles needed, based on user-defined synaptic array size.

The floorplan starts from tile sizing to PE sizing, while the size of synaptic array is defined by users in **Param.cpp**. With pre-defined network structure and weight mapping method, *NeuroSim* automatically calculate weight-matrix size for each layer (especially for convolutional ones, where 3D kernels will be unrolled to 2D matrices), the tile size firstly is set to a maximum value which could contain the largest weight-matrix among all the layers, then *NeuroSim* calculate the memory utilization (defined as memory mapped by synaptic weights / total memory storage on chip), keep decreasing the tile size till *NeuroSim* find a solution with optimal memory utilization.

To further increase memory utilization and speed up the processing speed of whole network as much as possible, weight duplication is introduced to each layer. Since the layer structure (such as input feature size, channel depth and kernel size) varies significantly in DNNs, which could occupy various amounts of synaptic arrays, it is possible that, the weight of several layers cannot fully fill one PE or even one synaptic array, a naïve way to custom-design the hardware is to mix multiple such small layers into one tile (or even one PE), however, this could make it complicated to define tile/PE size and number of tiles needed, thus, in this framework, we assume one tile is the minimum computation units for each layer, i.e., it is not allowed to map more than one layer into one tile, but there could be multiple tiles to map one single layer.

Hence, similarly, *NeuroSim* will continue to decide the PE size and possibilities of weight duplication among PEs, with pre-defined tile size as discussed above. For example, if the weight-matrix of a specific layer is smaller than the tile size (which means the tile cannot be fully filled by one weight-matrix), it is possible to duplicate the weight-matrix and fetch in multiple neural activation vectors, thus to speed up the process of this layer. In this step, *NeuroSim* start the PE design with a maximum PE size which equals to half of the tile size (to guarantee the exist of defined hierarchy), and decide whether to duplicate the weight-

matrix and how many times of duplication for each layer, then recalculate the memory utilization with weight duplication factors, keep decreasing the PE size till *NeuroSim* find the optimal solution with highest memory utilization.

Finally, weight duplication could be further utilized inside PE, i.e. duplicate weight among synaptic arrays, in the similar way as PE design, the only difference is the synaptic array size if fixed. With these three-stage floorplans, NeuroSim could guarantee high-enough memory utilization, meanwhile optimize the inference processing speed.

Table I Memory Utilization

Network	Conventional Mapping	Novel Mapping
VGG-8 (CIFAR-10)	91.45%	95.23%
AlexNet	98%	97%
VGG-16	98.79%	99.24%
ResNet-34	85.88%	90.13%

Table I shows the overall memory utilization of the floorplan algorithm of AlexNet, VGG-16 and ResNet-34, based on the two supported mapping methods for ImageNet dataset, and the default 8-layer VGG network in the simulator for CIFAR-10 dataset. The results were based on assumption that one memory cell is sufficient to map one synaptic weight (i.e. an 8-bit cell to map an 8-bit synapse), and synaptic array size is 128×128. With various hardware configuration (such as two 4-bit memory cells form one 8-bit synaptic weight), the memory utilization could be slightly different.

4.3 Weight Mapping Methods

We support two mapping methods in this framework, conventional mapping and novel mapping method which was proposed in [6]. Fig. 6 shows the example of conventional mapping for one convolutional layer, where each 3D kernel (weight) is unrolled into a long column, since the partial sums in each 3D will be summed up to get the final output. Thus, the total kernels in each convolutional layer will form a group of such long columns, i.e., a large weight matrix.

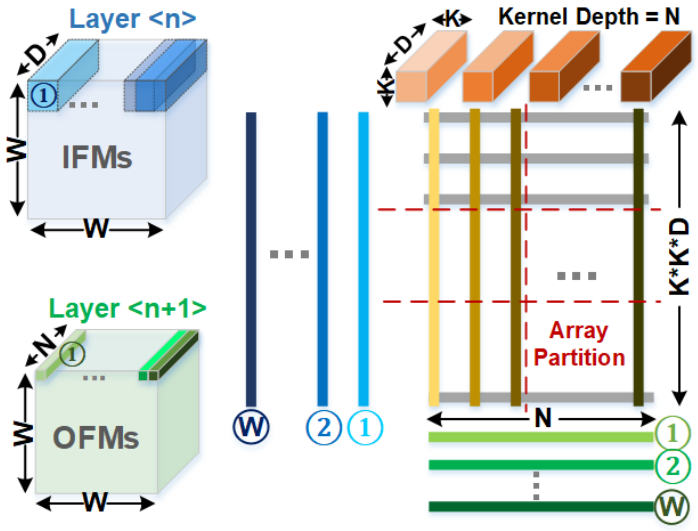


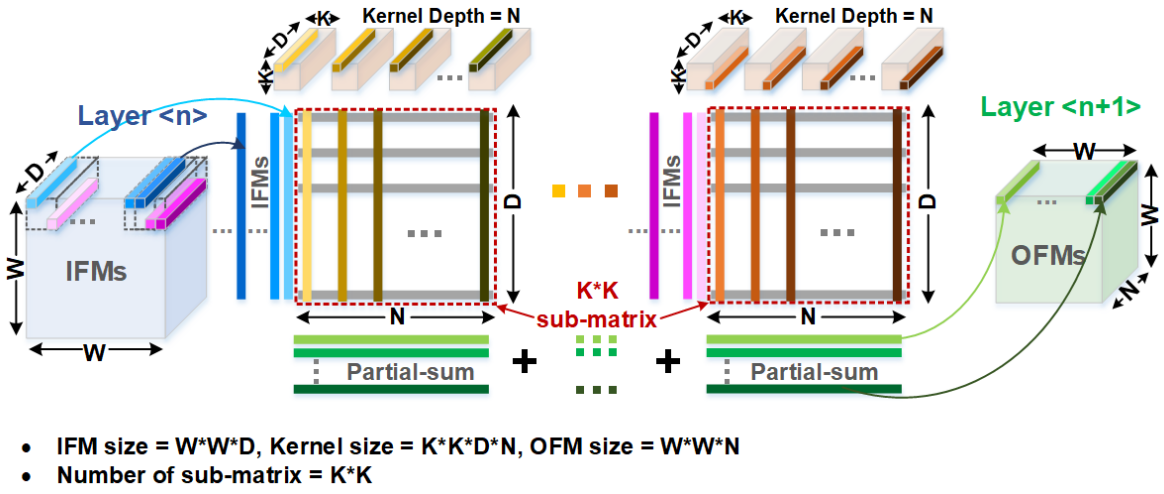
Fig. 6. An example of conventional mapping method of input and weight data.

To get the output feature maps (OFMs), as Fig. 6 shows, at first cycle, a part of input feature maps (IFMs) (shown in dark blue cube) will be multiplied with each 3D kernels. If we assume a single OFM has size of W×W, with channel depth of N, there are N such OFM in total, we call the front OFM as the first OFM, and the back one as the N^{th} OFM. In this way, the sum of dot-products from the first kernel will be the first element in the first OFM, the sum of dot-products from the second kernel will be the first element in the second OFM, and so on, thus, at the first cycle, we could get the first elements in every OFM from front to back (as shown in light green row in size 1×1×N). In the same way, at the second cycle, the kernels will "slide over" the inputs with a stride (equals to one in this example), after the dot-product operation, we will get all the second elements in each OFM. Thus, to generate the total OFMs in layer<n>, we need to "slide over" the IFMs by W×W times, i.e. we need W×W cycles to finish the computation.

It should be noted that, in conventional mapping, during the entire operation, a part of the IMFs used in earlier cycle will always be reused at current cycle. Considering about the huge amount of dot-product operations in convolutional layers, these frequent revisiting of input data from upper-level buffers could cause a significant energy and latency waste. Thus, a novel mapping method is introduced to maximize input data reuse.

Fig. 7 shows an example of novel mapping for the same convolutional layer. Instead of unrolling 3D kernels into a large matrix, the weights at different spatial location of each kernel are mapped into different sub-matrices. According to the spatial location of partitioned kernel data in each kernel, we define which group of these partitioned kernel data should belong to. Hence, K×K sub-matrices are needed for the kernels (whose first and second dimension equal to K and K), since each sub-matrix has size D×N, the size of total weight matrix will be K×K×D×N, which equals to the size of unrolled matrix from conventional mapping method (as Fig. 4 shows). Similarly, the input data which should be assigned to various spatial location in each kernel, will be sent to the corresponding sub-matrix, respectively. Partial sums from sub-matrices could be obtained in parallel. Later, an adder tree will be used to sum up the partial sums.

Hence, such group of sub-arrays with the necessary input and output buffers and accumulation modules can be defined as a processing element (PE). The kernels are split into several PEs according to their spatial locations, and assign the input data into corresponding ones, it is possible to reuse the input data among these PEs, i.e., directly transfer input data among PEs which do not need to revisit upper-level buffers.



- **IFM size = W*W*D, Kernel size = K*K*D*N, OFM size = W*W*N**
- **Number of sub-matrix = K*K**

Fig. 7. An example of novel mapping method of input and weight data.

4.4 Pipeline System

In this framework, we assume all the synaptic weights are mapped on to the chip, which means it is possible to build up a pipeline system with acceptable global buffer overhead (to save activations for different images), to improve throughput and energy efficiency (less leakage for idle cycles).

To avoid overhead of complicated control circuits, we assume each layer as one pipeline stage, and the pipeline system clock cycle is defined as the longest latency among all the layers. According to the mapping method of synaptic weights, the total latency of each layer is related to the size of its input feature maps (IFMs) and stride size.

Since the IFMs tend to become deeper but smaller from shallow layers to deeper layers, the speed of deeper layers will be limited by the shallow layers, since they have to wait for the shallow layers to generate the IFMs. During the waiting period, the deeper layers have to stay idle, and thus cause leakage energy.

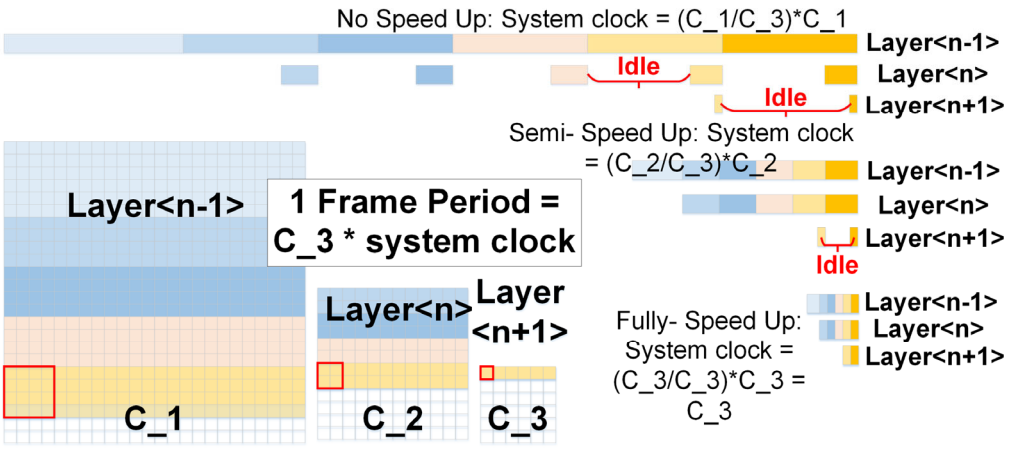


Fig. 8. Speed-up in Pipeline System.

In this case, we defined a parameter “speedUpDegree” in file “Param.cpp” to speed up each layer, by duplicating the weight and processing different IFMs simultaneously. As Fig. 8 shown below, if the size of IFMs is (C_1 * C_1), with 2X speed up, the actual latency will be (C_1 * C_1)/4. It should be noted that, in this framework, to avoid ultimate speed-up (i.e. ultimate weight duplication), we define a boundary of speed-up degree as the maximum speed-up allowed: there is no idle period across all the layers after speed-up.

5. Circuit Level: Synaptic Array Architectures

With various device technologies, the chip could operate in different modes, such as digital sequential (row-by-row) read-out for near-memory computing, or analog parallel read-out for in-memory computing. In the simulator, the parameters of synaptic devices and synaptic array modes will be instantiated in **param.cpp**.

5.1 Parallel Synaptic Array Architectures

Fig. 9 and Fig. 10 show three kinds of supported synaptic arrays, which could be used to process analog in-memory computing. Here are some assumptions that apply to all kinds of array architectures below. The higher precision than 1-bit in the input neuron activation is represented by multiple cycles of input voltage signals to the row, and no analog voltage is used to represented the input, thus no digital-to-analog converter (DAC) is used, as the nonlinearity in I-V curve of eNVMs will introduce distortion in parallel read-out [7]. The higher precision than 1-bit in the weight could be represented by a single analog synaptic cell or multiple synaptic cell. For example, 8-bit weight could be represented a single 8-bit eNVM cell (assuming it is technologically viable), or 2 eNVM cells (4 bits per cell), or 4 eNVM cells (2 bits per cell), or 8 eNVM binary cells. In our design, the inference is performed in parallel mode by activating all the rows, while the weight update in the training is performed in a row-by-row fashion. It should be noted that as the peripheral ADC size is typically much larger than the column pitch of the array, therefore column sharing is used by the column mux (e.g. 8 columns share one ADC).

1) SRAM synaptic array

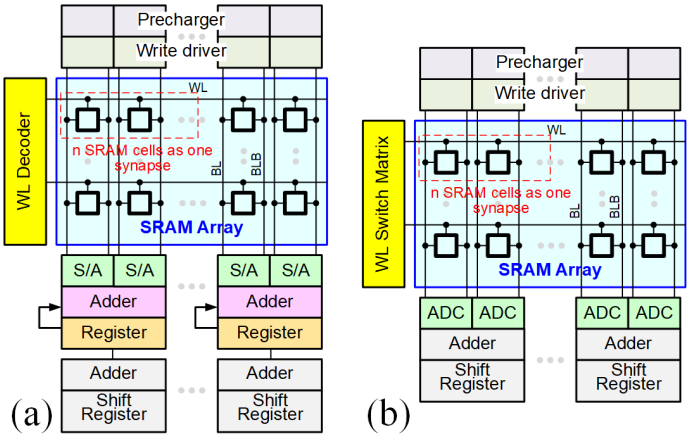


Fig. 9. The diagram of SRAM-based (a) sequential-read-out; (b) parallel-read-out synaptic arrays.

Multiple digital SRAM cells can be grouped along the row to represent one weight with higher precision than 1-bit, as shown in Fig. 9. The weighted sum and weight update operations are similar to the row-by-row read and write operations in conventional SRAM for memory, respectively. In sequential-read-out mode as Fig. 9 (a) shows, to select a row, the WL is activated through the WL decoder. To access all the cells on the selected row, the BLs are pre-charged by the pre-charger and the write driver in weighted sum and weight update, respectively. After the memory data are read by the sense amplifier (S/A), the adder and register are used to accumulate the partial weighted sum in a row-by-row fashion. In parallel-read-out mode as demonstrated in [8], the input vectors will be fetched in via WL switch matrix, the partial-sums will be

collected along columns simultaneously at one time with high-precision flash-ADCs based on multilevel sense amplifier (S/A) by varying references. In both modes, the adders and shift registers are used to shift and accumulate partial sums for multiple cycles of input vectors (which represent MSB to LSB of the analog neural activations). For inference based design, we will use 6T SRAM cell [8], and for training based design, we will use 8T SRAM cell [9] for transposable read-out.

2) Analog eNVM 1T1R synaptic array

Fig. 10 (a) and (b) shows the structure of 1T1R based eNVM array for sequential and parallel read-out, respectively. The WL controls the gate of the transistor, which can be viewed as a switch for the cell. The source line (SL) connects to the source of the transistor. The eNVM cell's top electrode connects to the BL,

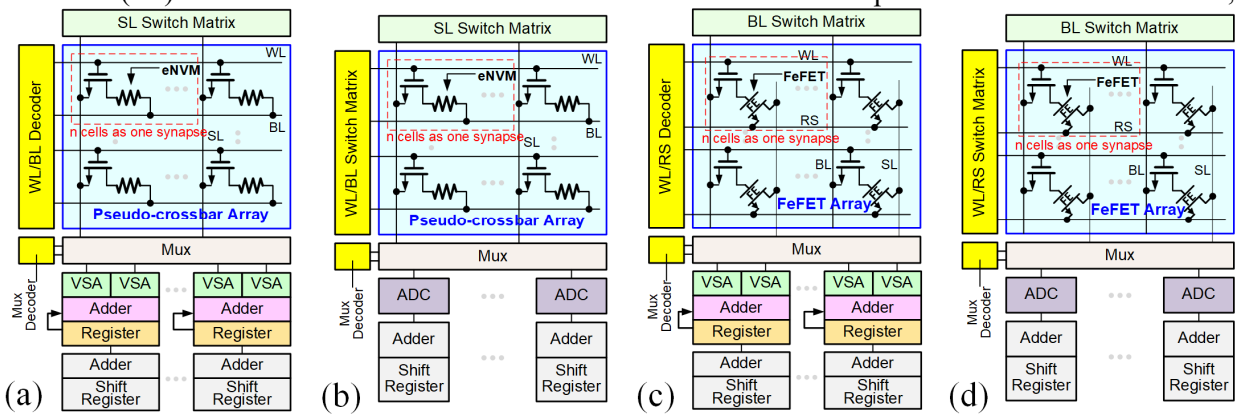


Fig. 10. (a) sequential-read-out and (b) parallel-read-out analog eNVM based pseudo-crossbar synaptic arrays; (c) sequential-read-out and (c) parallel-read-out analog FeFET synaptic arrays.

while its bottom electrode connects to the drain of the transistor through a contact via. In such case, the cell area of 1T1R array is then determined by the transistor size, which is typically $>6F^2$ depending on the maximum current required to be delivered into the eNVM cell. Larger current needs larger transistor gate width/length (W/L). However, conventional 1T1R array is not able to perform the parallel weighted sum operation. To solve this problem, we modify the conventional 1T1R array by rotating the BLs by 90°, which is known as the pseudo-crossbar array architecture, as shown in Fig. 11 (b). In weighted sum operation, all the transistors will be transparent when all WLs are turned on. Thus, the input vector voltages are provided to the BLs, and the weighted sum currents are read out through SLs in parallel. Then the weighted sum currents are digitalized by a current-mode sense amplifier (S/A), and a flash-ADC with multilevel S/A by varying references is used. It should be noted that the same pseudo-crossbar array could be used for transposable read-out during backpropagation in training, simply by switching the input vector and output between BL and SL.

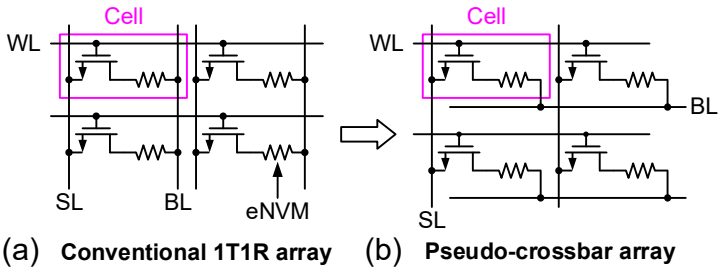


Fig. 11. Transformation from (a) conventional 1T1R array to (b) pseudo-crossbar array by 90° rotation of BL to enable weighted sum operation.

3) Analog eNVM crossbar array

The crossbar array structure has the most compact and simplest array structure for analog eNVM devices to form a weight matrix, where each eNVM device is located at the cross point of a word line (WL) and a bit line (BL). The crossbar array structure can achieve a high integration density of $4F^2$/cell (F is the lithography feature size). Here, the crossbar array assumes there is an ideal two-terminal selector device connected to each eNVM, which is desired for suppressing the sneak path currents during the row-by-row weight update. It should be noted that ideal selector device is still under research and development, therefore, we do not recommend using this mode for practical integration purpose.

4) Analog FeFET array

As shown in Fig. 10 (c) and (d), three-terminal devices (such as ferroelectric transistor, FeFET) could be used in the pseudo-crossbar fashion as proposed in [10]. It has an access transistor for each cell to prevent

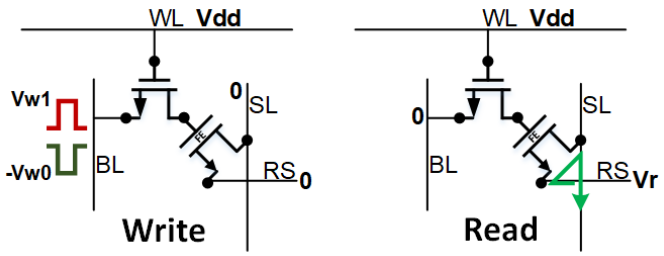


Fig. 12. Operations of (a) write and (b) read in FeFET cell.

programming on other unselected rows during row-by-row weight update. As FeFET is a three-terminal device, it needs two separate input signals to be fetched to activate WLs and introduce read voltages to RS (read select), respectively, where RS is used to fetch in input vectors as Fig. 12 shown below. Here we assume a positive write voltage to the gate to program the FeFET to a higher conductance state, while a negative write voltage to the gate to erase the FeFET to a lower conductance state. If no negative voltage is desired on-chip, then we may rely on the drain-erase scheme as proposed in [11].

5.2 Transposable Synaptic Arrays

In the framework, to support on-chip training, we implement transposable synaptic arrays in NeuroSim core. As Fig. 13 shows, during back-propagation, to calculate the errors, the errors (i.e. the gradient of loss function respective to the activation) from deeper layer need to be fetched backwards and do the element-wise multiplication and accumulation with the prior kernels. In layer<n>, the error from layer<n+1> will be the input data, and be separated into different channels, then applied to corresponding kernels. For example, the error from first channel (shown as light green plane) will be fetched to the first kernel (shown as light yellow plane) to do the element-wise multiplication, and the accumulated output will be 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 location and collected along their channel (dimension=D), and mapped into the columns in CIM arrays. It would be considered that, the rows in such CIM arrays are the weights in a specific location in each kernel from different channels. For example, as shown in Fig. 13, the first-channel weights at left-most and top-most location of each kernel (shown as light yellow nodes) are mapped as the first row in the first sub-matrix; the last-channel weights at right-most and bottom-most location of each kernel (shown as dark orange nodes) are mapped as the last row in the last sub-matrix.

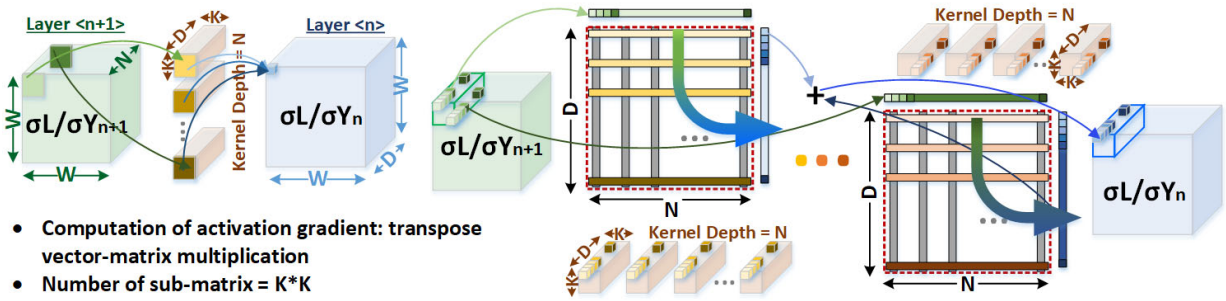


Fig.13. The computation of error in CIM during backpropagation, based on novel mapping method of convolutional layer.

In this case, since the results at the same channel in different kernels will be accumulated to get the outputs, it is straightforward to find that, 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 error. Thus, the error in layer<n+1> will be sent to each sub-matrix according to their spatial locations during kernel-sliding, and each of them will be considered as inputs to each column in the CIM array.

Therefore, we need to modify the design of synaptic arrays in the CIM architecture, to support both conventional (feed-forward) and transposed (activation gradient) computations. As shown in Fig. 14, 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 or parallel read-out schemes.

In SRAM-based synaptic arrays, during feed-forward, the conventional computation scheme is to activate each rows, then read out and accumulate the products along the columns. During back-propagation, to calculate the activation gradients, we need to activate each columns and accumulate the product along rows. To do so, we need to implement transposed word-lines, bit-lines with 8T-SRAM cells (duplicate and rotate 90 degree from the original ones, shown as light blue lines) to realize transposed computation, with additional peripheral circuits (such as WL decoder or switch matrix, sense amplifier or ADC, and shift-adder with registers).

However, in eNVM-based synaptic arrays, since there are SL switch matrix (in two-terminal eNVM designs) or BL switch matrix (in three-terminal eNVM designs) for weight update, in transposed computation (for error), we do not need to add additional peripheral circuits to activate the columns (fetch in horizontal inputs), but just directly use the SL or BL switch matrix to do so. 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.

In DNN+NeuroSim V2.0, we do not consider pipeline among the four key steps in training, i.e. 1) feed-forward, 2) computation of error, 3) computation of weight gradient, and 4) weight update, to avoid complex circuit design (for logical control) above synaptic array level.

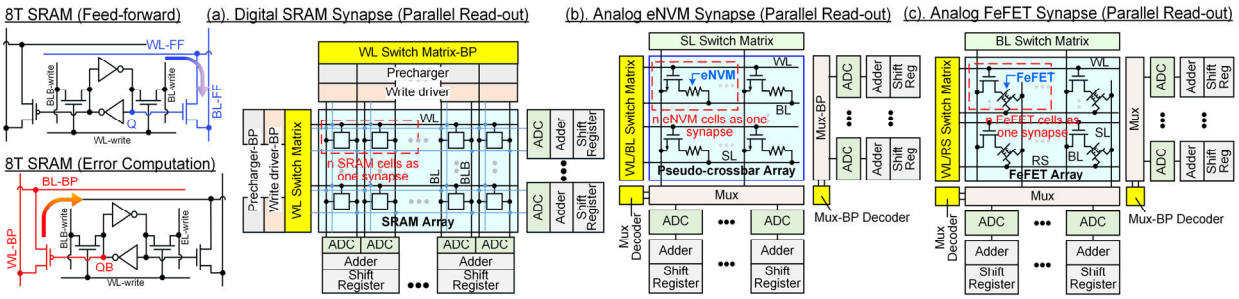


Fig. 14. In DNN+NeuroSim V2.0, transposable synaptic arrays are implemented to support on-chip training. Available synaptic devices are (a) SRAM, (b) tow-terminal eNVMs (e.g. RRAM) and (c) three-terminal eNVMs (e.g. FeFET), with both sequential and parallel read-out schemes.

5.3 Weight Gradient Calculation Unit

When the computation of error is done and stored to the off-chip DRAM memory (for each batch), we need to start the computation of weight gradient. As Fig. 15 shows, to calculate the weight gradients of layer<n>, the error from layer<n+1> will be applied to do element-wise multiplication and accumulation with the activations from layer<n> in a channel-to-channel scheme.

For example, a part of the first-channel activations in layer<n> (shown as light green plane) will be multiplied with the error in layer<n+1> (shown as light blue plane), the element-wise products will be accumulated and be 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 error 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 left-most and top-most spatial locations through all the channels for each kernel. To get all the weight gradients, we need to sweep the activations and repeat the multiplication-and-accumulation with the error by K×K times, representing K×K spatial locations in the kernels.

Therefore, we could easily unroll each channel of the error 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, with channel depth equals to N, there will be N unrolled vectors of inputs applied to the matrix, to get one group of the weight gradients (corresponding to the weights at a specific spatial location of each kernel, and also representing one of the K×K sub-matrix in novel mapping of the weights). Thus, to sweep the activations by K×K times to get all the weight gradients, in total there will be K×K×N unrolled vectors of activations applied to the matrix of activation gradients.

Since the weight gradients need to be calculated for various inputs across the batch, and finally accumulated to get the delta weights for each iteration, we can find that, we need to frequently re-write the matrix of errors for different images, which leads to a huge overhead of memory writing latency and energy. Due to this, we choose the SRAM-based CIM non-transposable array for weight gradient calculation 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 can only put enough SRAM-based CIM arrays to support the layer with largest size of activation gradients through the entire network, and hence, thus we find that the area overhead of SRAM-based weight gradient computation is acceptable.

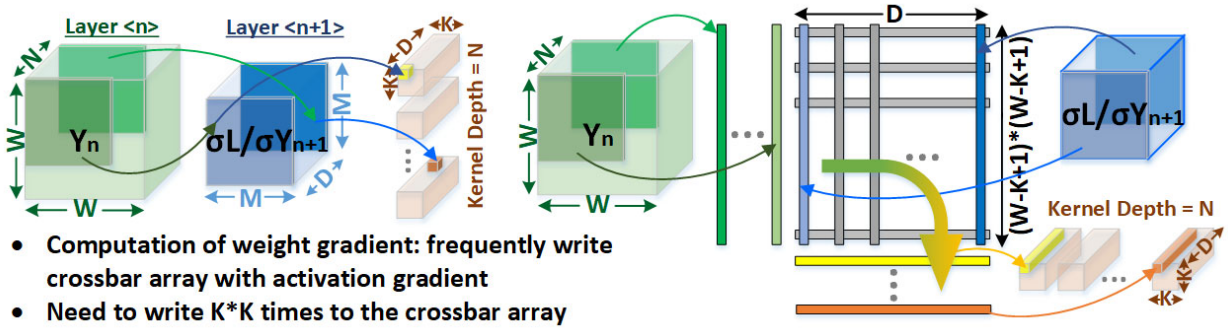


Fig. 15. The computation of weight gradient in CIM.

5.4 Synaptic Array Peripheral Circuits

The periphery circuit modules used in the synaptic arrays in Fig.9 and Fig. 10 are described below:

1) Switch matrix

Switch matrices are used for fully parallel voltage input to the array rows or columns. Fig. 16 (a) shows the BL switch matrix for example. It consists of transmission gates that are connected to all the BLs, with control signals (B_1 to B_n) of the transmission gates stored in the registers (not shown here). In the weighted sum operation, the input vector signal is loaded to B_1 to B_n, which decide the BLs to be connected to either the read voltage or ground. In this way, the read voltage that is applied at the input of transmission gates can pass to the BLs and the weighted sums are read out through SLs in parallel. If the input vector is higher than 1 bit, it should be encoded using multiple clock cycles, as shown in Fig 16 (b). The reason why we do not use analog voltage to represent the input vector precision is the I-V nonlinearity of eNVM cell, which will cause the weighted sum distortion or inaccuracy as discussed above. In the simulator, all the switch matrices (**slSwitchMatrix**, **blSwitchMatrix** and **wlSwitchMatrix**) are instantiated from **SwitchMatrix** class in **SwitchMatrix.cpp**, this module is used in parallel-read-out synaptic arrays

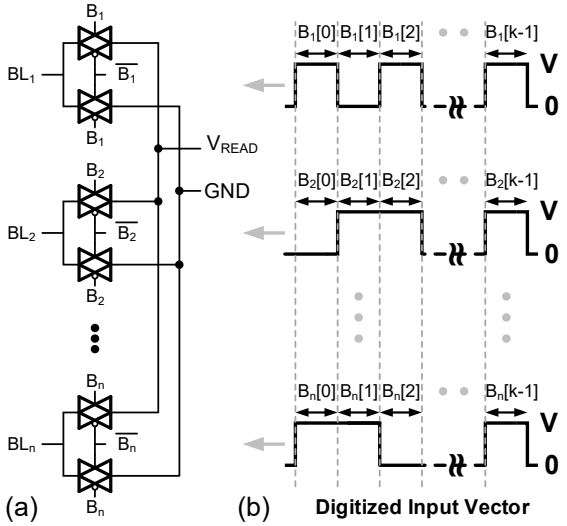


Fig. 16 (a) Transmission gates of the BL switch matrix in the weighted sum operation. A vector of control signals (B1 to Bn) from the registers (not shown here) decide the BLs to be connected to either a voltage source or ground. (b) Control signals in a bit stream to represent the precision of the input vector.

2) Crossbar WL decoder

The crossbar WL decoder is modified from the traditional WL decoder. It has an additional feature to activate all the WLs for making all the transistors transparent for weighted sum. The crossbar WL decoder is constructed by attaching the follower circuits to every output row of the traditional decoder, as shown in Fig. 17. If *ALLOPEN*=1, the crossbar WL decoder will activate all the WLs no matter what input address is given, otherwise it will function as a traditional WL decoder. In the simulator, the crossbar WL decoder contains a traditional WL decoder (**wlDecoder**) instantiated from **RowDecoder** class in **RowDecoder.cpp** and a collection of follower circuits (**wlDecoderOutput**) instantiated from **WLDecoderOutput** class in **WLDecoderOutput.cpp**, this module is used in sequential-read-out synaptic arrays

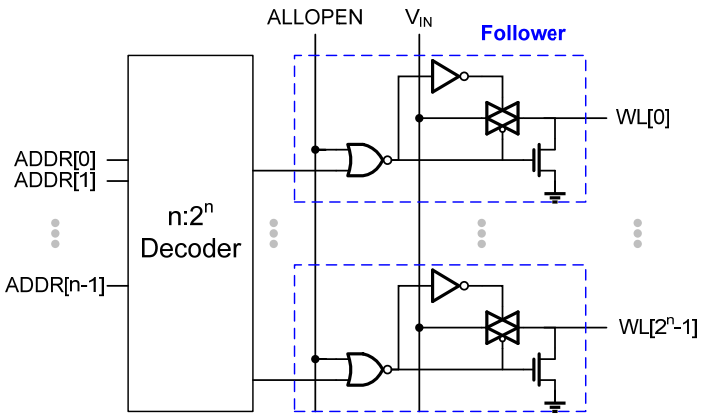


Fig. 17 Circuit diagram of the crossbar WL decoder. Follower circuit is attached to every row of the decoder to enable activation of all WLs when ALLOPEN=1.

3) Decoder driver

The decoder driver helps provide the voltage bias scheme for the write operation when its decoder selects the cells to be programmed. As the digital eNVM crossbar array has the write voltage bias scheme for both WLs and BLs, it needs the WL decoder driver (**wlDecoderDriver**) and column decoder driver (**colDecoderDriver**). These decoder drivers can be instantiated from **DecoderDriver** class in **DecoderDriver.cpp**, this module is used in sequential-read-out synaptic arrays.

4) New Decoder Driver and Switch Matrix

One should be noticed that, for eNVM pseudo-crossbar and FeFET synaptic arrays, the WLs and BLs/RSs could be controlled by same input signals, but with different voltage values, thus, it could significantly save the area for unnecessary BL/RS switch matrix. To achieve this function, there are several extra control gates to be added into the WL decoder driver circuits, and into the WL switch matrix. Fig. 18 shows the circuit diagram of new decoder driver and switch matrix for eNVM pseudo-crossbar array, which could be used to control both WL and BL (or RS) at the same time. In Fig. 18 (a), with the input and decoder output, both of WL and BL will be controlled, where the WLs will be either activated or not, and the BLs to be connected to either the read voltage or ground. Similarly, in Fig. 18 (b), the each single WL switch matrix has two extra transmission gates to be used to send two separate voltages into the corresponding WL and BL. In

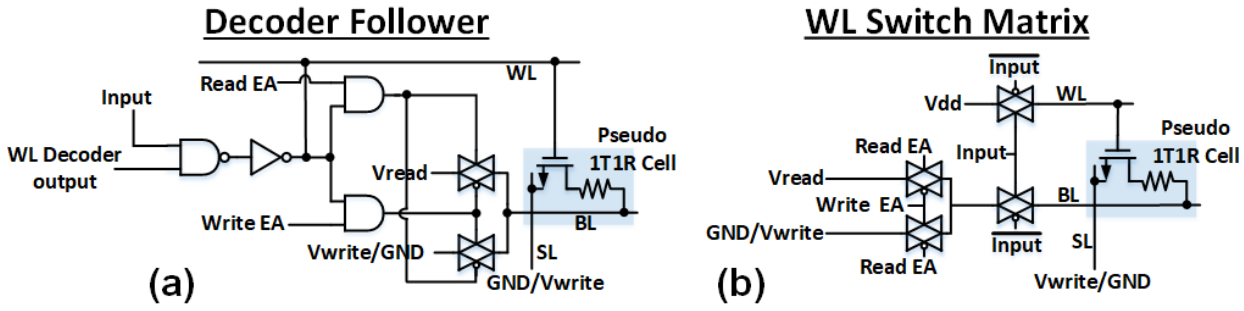


Fig. 18 Circuit diagram of (a) decoder follower and (b) WL switch matrix, which are used to control both WLs and BLs simultaneously, for pseudo-crossbar arrays.

FeFET synaptic arrays, the signals connected to BLs in this example, will be connected to RSs. In the simulator, the WLNewDecoderDriver (decoder driver) is instantiated from **WLNewDecoderDriver** class in **NewDecoderDriver.cpp** and the WLNewSwitchMatrix (WL switch matrix) is instantiated from **WLNewSwitchMatrix** class in **NewSwitchMatrix.cpp**, these new decoder follower and switch matrix are used in eNVM pseudo-crossbar and FeFET synaptic arrays.

5) Multiplexer (Mux) and Mux decoder

The Multiplexer (Mux) is used for sharing the read periphery circuits among synaptic array columns, because the array cell size is much smaller than the size of read periphery circuits and it will not be area-efficient to put all the read periphery circuits to the edge of the array. However, sharing the read periphery circuits among synaptic array columns inevitably increases the latency of weighted sum as time multiplexing is needed, which is controlled by the Mux decoder. In the simulator, the Mux (**mux**) is instantiated from **Mux** class in **Mux.cpp** and the Mux decoder (**muxDecoder**) is instantiated from **RowDecoder** class in **RowDecoder.cpp**.

6) Analog-to-digital converter (ADC)

To read out the partial-sums and further process them in the subsequent logic modules (such as activation and pooling), a group of flash-ADCs with multilevel S/A by varying references are used at the end of SLs to generate digital outputs. In the simulator, we take a conventional current-sense-amplifier (CSA) as shown in Fig. 19, as the unit circuit module, to build up multilevel S/A. To precisely estimate the latency and energy of S/A, we run Cadence simulation across technology from 130nm to 7nm, for each technology node, we chose reasonable BL current range (considering practical device resistance range), and in the range we select multiple specific nodes I_{BL}, detect the latency and power trends of each specific I_{BL} when sweeping I_{ref} (i.e. from $0.001\times I_{BL}$ to $1000\times I_{BL}$). As a detection of multiple experiments based on SPICE simulation, when fix I_{BL} and sweep I_{ref}, both latency and energy varies significantly, with various I_{ref}/I_{BL} values, when I_{ref}/I_{BL} is approaching to 1, the latency and energy will be the maximum (extremely hard for S/A to sense the difference); however, if we fix the I_{ref}/I_{BL} to a minimum value which leads to maximum latency and energy, and sweep the I_{BL}, the changes are quite smooth and not significant.

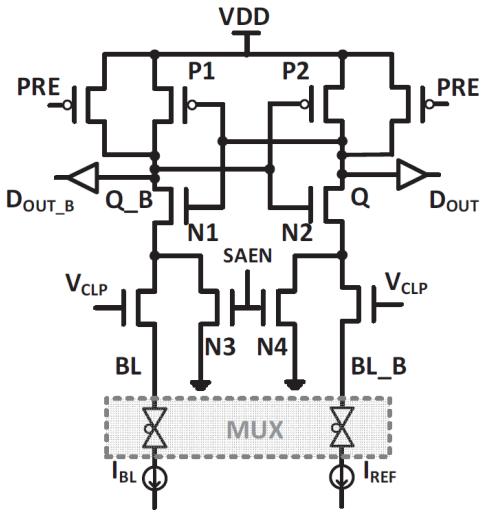


Fig. 19 Schematic of current sense amplifier (CSA).

Then, we sweep the technology nodes, at each technology node, we sweep the I_{BL}, and for each I_{BL}, we sweep the I_{ref}. We collect all the simulated data from SPICE simulation, then fit the data and build up functions of latency and energy in relation with I_{BL} and I_{ref} for each technology node. In this way, in *NeuroSim*, we are able to estimate the latency and energy based on real traces (which gives specific I_{BL}, while I_{ref} are automatically defined by *NeuroSim* according to R_{on}, R_{off}, synaptic array size and precision of ADC). Fig. 20 shows an example of latency estimation based on the fitting functions, where the blue dots are estimated results and red dots are simulated results from SPICE, the fitting function yields reasonable mismatch with much faster simulation compared with SPICE.

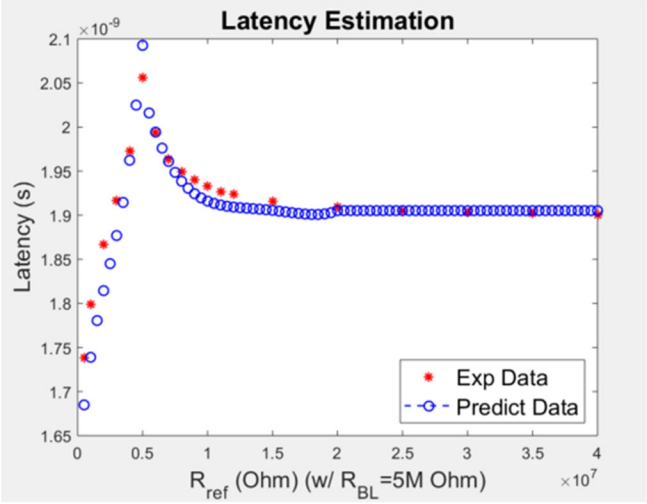


Fig. 20 An example of latency estimation based on fitting functions compared with Cadence results.

To read out the partial-sums in parallel modes, it requires ADC with high enough precision, for example, with synaptic array size 128×128, and each cell represents 1-bit synapse, the partial-sums along one column would be 7-bit which is impractical as ADC precision, thus we have to truncate the precision of ADC (for partial sums) to minimize the area and energy overhead.

As Fig. 21 shows, we perform 8-bit inference of VGG-8 network on CIFAR-10 dataset, to investigate the effects of truncating ADC precision on the inference accuracy. We set the sub-array size to be 128×128, and investigate three schemes with 1-bit per cell, 2-bit per cell and 4-bit per cell. To minimize the ADC truncation effects on the partial-sums, we utilize the nonlinear quantization with various quantization edges (corresponding to different ADC precision), where the edges are determined according to the distribution of partial-sums, as proposed in [12]. Compared to the baseline accuracy (no ADC truncation), the results suggest that at least 4-bit ADC is required to prevent significant accuracy degradation. Compared to a prior work on binary neural network where 3-bit ADC was reportedly sufficient [12], the results in Fig. 21 suggest that higher weight-precision generally requires higher ADC-precision.With larger synaptic array size or higher cell precision, higher ADC precision is demanded. For flash-ADC, higher than 3-bit may still result in significant area overhead, thus more compact ADC design (e.g. SAR-ADC) is still under development for future release.

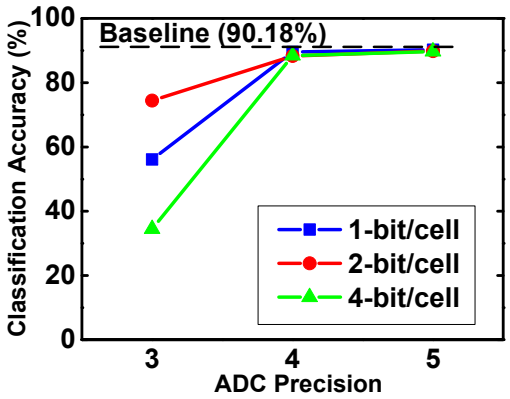


Fig. 21. Inference accuracy of CIFAR-10 for an 8-bit CNN as a function of the ADC precision for partial sums.

7) Adder and register

As mentioned earlier, the adders and registers are used to accumulate the partial weighted sum results during the row-by-row weighted sum operation in digital synaptic array architectures. The group of adders is instantiated from **Adder** class in **Adder.cpp** and the group of registers (**dff**) is instantiated from **DFF** class in **DFF.cpp**.

8) Adder and shift register

The adder and shift register pair at the bottom of synaptic core performs shift and add of the weighted sum result at each input vector bit cycle (B_1 to B_n in Fig 16 (b)) to get the final weighted sum. The bit-width of the adder and shift register needs to be further extended depending on the precision of input vector. If the values in the input vector are only 1 bit, then the adder and shift register pair is not required. In the simulator, a collection of the adder and shift register pairs (**ShiftAdd**) is instantiated from **ShiftAdd** class in **ShiftAdd.cpp**, where **ShiftAdd** further contains a group of adders (**adder**) instantiated from **Adder** class in **Adder.cpp** and a group of registers (**dff**) instantiated from **DFF** class in **DFF.cpp**.

6. Algorithm Level: PyTorch and TensorFlow Wrapper

The algorithm we use to get the quantized DNN model for inference is the WAGE from [13]. The TensorFlow code is modified based on the author released code at [14]. The Pytorch code realizes the same algorithm except that we move the scale term from weight to output to make it more suitable for the hardware architecture. We referenced [15] [16] for our Pytorch code. This algorithm could directly train a quantized network with user defined bit width for weight, activation, gradient and error. The partial sum quantization (according to ADC precision) is also introduced by our modification.

In V1.1, we considered inference on hardware (with offline training by software). In general, users could either train the network with floating point and find the quantization level with statistics for weight and activation or introduce quantization with desired quantization level during training directly. We choose the second scheme using WAGE since WAGE quantize both weight and activation using fixed quantization level, which is [-1, 1] with scale of 2^{-b}. This mechanism is friendly to hardware implementation, which normally represent data use 2's complimentary. WAGE also apply quantization to gradient and error, which is not necessary for inference stage (but will be useful for on-chip training design in V2.0 release). Users could set the bit-width to -1 to make these two floating-point for inference. Users need to pay attention that some hyper-parameters need to be fine-tuned if the bit-width is changed for WAGE algorithm.

The key parameters that will be transferred from the DNN algorithm to *NeuroSim* are weight precision (determining the synaptic weight cell design), partial sum precision (determining the ADC precision), and the activation precision (determining the input clock cycle number). For inference, the weight patterns are pre-defined by software training, and they will be transferred to *NeuroSim* only once (acting as one-time programming), and then the input dataset (e.g. 1 test image) is loaded for the hardware performance estimation. For on-chip training, we assume both of the inference and training are operating on-chip, the feed-forward and error computation will be done in the transposable synaptic arrays, while weight gradient computation is done in the weight gradient computational units. The python wrapper will transfer both of the old and new weight data to *NeuroSim* core for real-traced hardware estimation, as the back-propagation involves huge amount of data transfer, we also consider DRAM access in V2.0 (as discussed in ***Section 4***).

---

7. Algorithm Level: Inference Accuracy Estimation in V1.1

In this framework, the neural network model (or weights) is assumed to be pre-trained by software, and then mapped to the compute-in-memory (i.e. CIM) inference chip. Thus, the non-ideal effects of synaptic devices (such as nonlinearity, asymmetry and endurance during weight-update operation) are not considered in this V1.1 version. On the contrast, the main factors that we introduced into the accuracy estimation of inference chip are: on/off ratio, ADC quantization effects, conductance variation and retention.

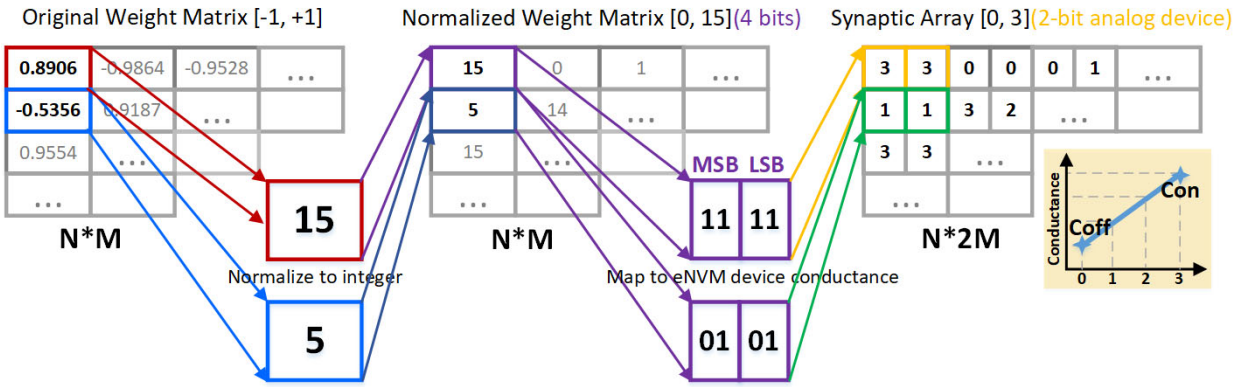


Fig. 22. Mapping weight from algorithm to synaptic device conductance in CIM architecture.

As Fig.22 shows, to represent the weights from algorithm (floating-point) on the CIM architectures, due to the limited precision of synaptic devices, one ideal way is to normalize the weights to decimal integers, and then digitalize the integers to conductance levels. For example, as shown in Fig. 22, if we define the synaptic weight precision to be 4-bit (decimal integer 0 to 15), and represented by 2-bit (conductance level 0 to 3) synaptic devices, from algorithm, the floating-point weight "+0.8906" will be normalized to 15, and thus be mapped to two synaptic devices, one as LSB and one as MSB, and each of them are on conductance level 3 (i.e.15/4=3, 15%4=3). Similarly, with a negative number "-0.5356", it will be firstly normalized to 5, and then mapped to two synaptic devices, and each of the LSB and MSB will be on conductance level 1 (i.e. 5/4=1, 5%4=1).

1) Conductance On/Off Ratio

Ideally, the conductance levels of synaptic devices range from 0 to 2^N, where N is the precision of synaptic devices. However, the minimum conductance can be regarded as 0 only if the conductance on/off ratio (=maximum conductance/minimum conductance) of synaptic devices is infinity, which is not feasible in current technology.

Thus, in reality, the minimum conductance level is not an ideal "0". For example, if we use a normalized synaptic device conductance range as 0~1 (as $0\sim2^N/2^N$), where the "1" can be represented as maximum conductance, and "0" is minimum conductance, in algorithm aspect, the conductance level "1" represent ideal "1", while the conductance level "0" actually represent a non-zero value "1/(on/off ratio)". In this case, small on/off ratio will introduce such non-ideal zeros into the calculation, and significantly distort the inference accuracy.

One approach to remedy this situation is to eliminate the effect of the OFF-state current in every weight element with the aid of a dummy column. In this framework, as Fig. 23 shows, we map the algorithm

weights (range [-1, +1]) to synaptic devices (conductance range [G_{min}, G_{max}]) in the synaptic arrays, while we set a group of dummy columns beside each synaptic array, and the devices in dummy columns are set to the middle conductance (G_{min}+G_{max})/2. Such that, by subtracting the real outputs with the dummy outputs, the truncated conductance range will be [-(G_{max}-G_{min})/2, +(G_{max}-G_{min})/2], which is zero-centered as [-1, +1], and the off-state current effects are perfectly removed.

The conductance on/off ratio is defined as one argument "args.onoffratio" in "inference.py" file.

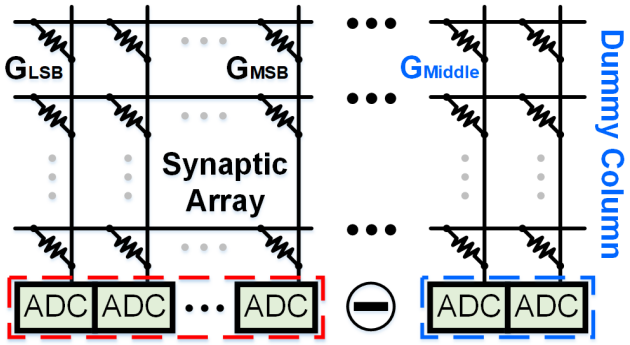


Fig. 23. Introduce dummy column to cancel out the off-state current effects.

2) Conductance Variation

It is well known that the eNVM based synaptic devices involving drift and diffusion of the ions/vacancies show considerable variation from device to device, and even from pulse to pulse within one device. Thus, in inference chip, although the weight-update operation is not required, conductance variation is still a concern during initialization or this one-time programming of the synaptic arrays.

In this framework, the conductance variation is introduced as a percentage of variation of desired conductance (instead of an absolute value), for example, if the desired conductance is 0.5, with +0.1 conductance variation, the actual conductance will be 0.55, similarly, with -0.2 conductance variation, the actual conductance will be 0.4.

For a particular chip, the conductance variation could be different from array to array, and device to device, so we model such variation as a function of random generator, to generate conductance variation of different cells, while the standard deviation of this random generator will be the argument in "inference.py" file, as "args.vari".

3) Retention

Retention refers to the ability of memory device to retain its programmed state over a long period of time. Typical retention specification for NVM in memory application is more than 10 years at 85°C. Many binary eNVM devices have been able to meet this requirement. However, there are no sufficient reported experimental data for analog eNVM that shows such retention, which can be attributed to the instability of intermediate conductance states.

To be general, we consider four scenarios of conductance drift for the retention analysis, as show in Fig. 24, where the conductance can either drift toward its maximum, minimum or intermediate states, or just

randomly drift. The formula for modeling the conductance drift behavior is assumed (as phase change memory) to follow the one shown below:

$$G=G_0\left(\frac{t}{t_0}\right)^v$$

where "G_0" is the initial conductance, "t" is the retention time, "v" is the drift coefficient and "t_0" is the time constant which is assumed to be 1 second in this framework.

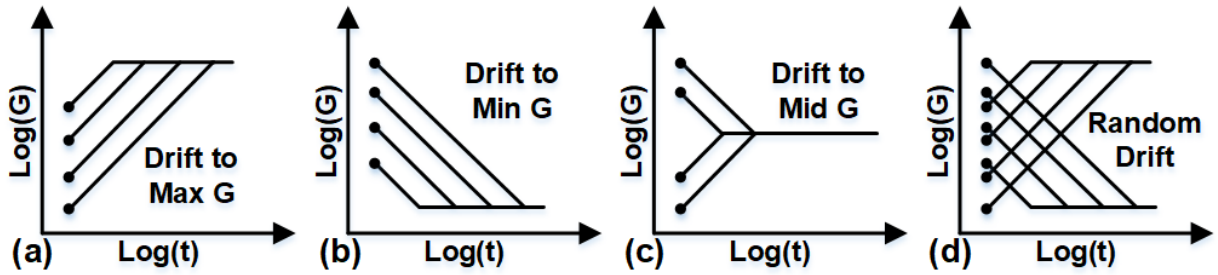


Fig. 24. Different scenarios of conductance drift.

To estimate the retention effect on inference accuracy, we define a function called "Retention" in file "wage_quantizer.py", where the retention time and drift coefficient are defined as "args.t" and "args.v" separately, while "args.detect" is used to define the drift scenario, if "args.detect" is 1, then the drift scenario is drifting to a fixed value (otherwise, it is random drift), and the targeted value is then defined as "args.target", the range is defined from 0 to 1. Those arguments can be defined in file "inference.py".

4) ADC Quantization Effect

For CIM architecture, there are mainly two read-out schemes. A sequential processing method of the matrix-vector multiplication is to read out the dot-products in a row-by-row manner, which leads to extra energy and latency for accumulations along the rows. A more efficient method is parallel processing, where multiple rows are activated simultaneously by a switch matrix, and the current summation will be read out by an ADC. Therefore, the row-by-row accumulation periphery of sequential scheme is eliminated. However, since it is impractical to use very high-precision ADC at the edge of synaptic arrays, we have to truncate the precision of ADC (for partial sums) to minimize the area and energy overhead.

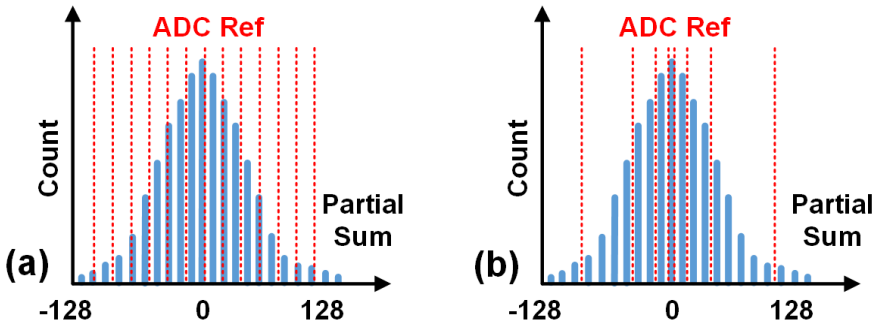


Fig. 25. Linear and non-linear ADC quantization.

To minimize the ADC precision while guarantee the inference accuracy, it is necessary to run the simulation of ADC quantization before hardware design. In this framework, we support two quantization methods: linear and non-linear quantization. As Fig. 25 shows, in linear quantization, the ADC references are distributed linearly across possible partial-sum value range in the synaptic arrays; while in non-linear quantization, the ADC references are non-linearly distributed, according to the distribution of partial-sums, the references are more spread in high-probability area, while less in low-probability part.

Normally, non-linear quantization can save ~1-bit ADC precision compared with linear quantization, however, the choice of non-linear references and quantized outputs is quite sensitive, the detection of partial-sum distribution is necessary. In this framework, we defined two functions called "NonLinearQuantizeOut" and "LinearQuantizeOut" in file "wage_quantizer.py", while the users can define the "args.ADCprecision" in file "inference.py", the user could also customize the ADC implementation for each layer in file "quantization_cpu_np_infer.py" as below:

```
115                 # Add ADC quanization effects here !!!
116                 outputPartialQ = wage_quantizer.LinearQuantizeOut(outputPartial, self.ADCprecision)
117                 outputDummyPartialQ = wage_quantizer.LinearQuantizeOut(outputDummyPartial, self.ADCprecis
```

```
150             # Add ADC quanization effects here !!!
151             outputPartialQ = wage_quantizer.LinearQuantizeOut(outputPartial, self.ADCprecision)
152             outputDummyPartialQ = wage_quantizer.LinearQuantizeOut(outputDummyPartial, self.ADCprecision
```

Fig. 26. Example of ADC implementation in each layer.

8. Algorithm Level: On-chip Training Accuracy Estimation in V2.0

To extend the framework to estimate the on-chip training performance, we introduced the non-ideal properties into the python wrapper, including nonlinearity and asymmetry of the eNVMs, device-to-device and cycle-to-cycle variations, as shown in Fig. 27.

1) Nonlinear weight update

Ideally, the amount of weight increase (or long-term potentiation, LTP) and weight decrease (or long-term depression, LTD) should be linearly proportional to the number of write pulses. However, the realistic devices reported in literature do not follow such ideal trajectory, where the conductance typically changes rapidly at the beginning stages of LTP and LTD and then gradually saturates. We have built a device behavioral model to capture nonlinear weight update behavior, where the conductance change with number of pulses (P) is described with the following equations:

$$G_{LTP} = B\left(1 - e^{\left(-\frac{P}{A}\right)}\right) + G_{min} \tag{8.1}$$

$$G_{LTD} = -B\left(1 - e^{\left(\frac{P-P_{max}}{A}\right)}\right) + G_{max} \tag{8.2}$$

$$B = (G_{max} - G_{min})/(1 - e^{\frac{-P_{max}}{A}}) \tag{8.3}$$

G_{LTP} and G_{LTD} are the conductance for LTP and LTD, respectively. G_{max}, G_{min}, and P_{max} are directly extracted from the experimental data, which represents the maximum conductance, minimum conductance and the maximum pulse number required to switch the device between the minimum and maximum conductance states. A is the parameter that controls the nonlinear behavior of weight update. A can be positive (blue) or negative (red). In Fig. 27, the A of LTP and LTD has the same magnitude but different signs. B is simply a function of A that fits the functions within the range of G_{max}, G_{min}, and P_{max}. All these parameters can be different in LTP and LTD in the fitting by the MATLAB script. However, for simplicity, the simulator currently uses the smaller value of G_{max} and the larger value of G_{min} in LTP and LTD.

Using Eq. (8.1)-(8.3), a set of nonlinear weight increase (blue) and weight decrease (red) behavior can be obtained by adjusting A as shown in **Error! Reference source not found.**8, where each nonlinear curve is labeled with a nonlinearity value from +6 to -6. It can be proved that Eq. (8.1) and (8.2) are equivalent with a different sign of A, thus we will just use Eq. (8.1) to calculate both nonlinear LTP and LTD weight update.

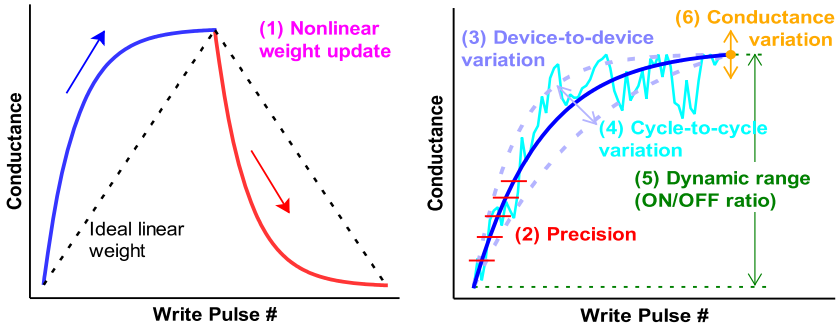


Fig. 27. Summary of non-ideal analog eNVM device properties.

Different than **Error! Reference source not found.**, all LTD curves are mirrored and shifted horizontally to make sure the curve starting from the pulse number 0 for simpler formulization.

In this framework, we define the nonlinearity as "args.nonlinearityLTP" and "args.nonlinearityLTD" in file "train.py" from python wrapper.

2) Limited precision

The precision of an analog eNVM device is determined by the number of conductance states it has, which is P_{max} in Eq. (8.1)-(8.3). Thus, in the framework, we define weight precision "args.wl_weight", gradient precision "args.wl_grad" to be same as the cell precision "args.cellBit", which should be guaranteed by the number of conductance states "args.max_level".

3) Device-to-device weight update variation

The effect of device-to-device weight update variation can be analyzed by introducing the variation into the nonlinearity baseline. This variation is defined as the nonlinearity baseline's standard deviation (σ) respect to 1 step of the 6 steps in **Error! Reference source not found.**8. In the framework, we introduced the device-to-device variation as "args.d2dVari" in file "train.py" from python wrapper.

4) Cycle-to-cycle weight update variation

The cycle-to-cycle weight update variation is referred to as the variation in conductance change at every programming pulse. This variation (σ) is expressed in terms of the percentage of entire conductance range. In the framework, we introduced the cycle-to-cycle variation as "args.c2cVari" in file "train.py" from python wrapper.

5) Fitting by MATLAB script (nonlinear_fit.m)

In this section, we will fit the experimental weight update data and extract the device parameters that will be used in the simulator. We have developed a MATLAB script **nonlinear_fit.m** to do such a task, where it has been set up for fitting Ag:a-Si devices in [17].

Before the fitting, the user has to make sure the experimental weight update data are pre-processed in a format that is similar to **Error! Reference source not found.**8. Namely, the LTD data should be mirrored horizontally and both LTP and LTD data should start from the pulse number 0 so that the data can be fit by Eq. (8.1)-(8.3). The user can look at the pre-processed data of Ag:a-Si devices as an example in the MATLAB script.

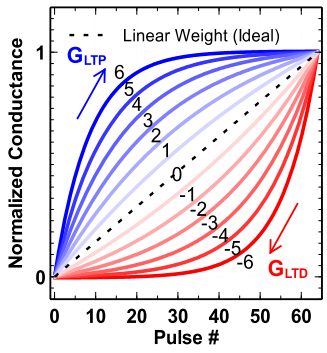


Fig. 28. Analog eNVM device behavioral model of the nonlinear weight update with the nonlinearity labeled from -6 to 6.

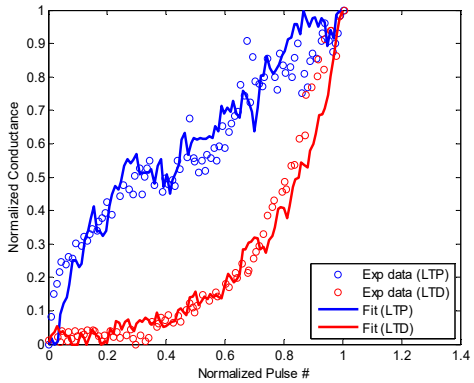


Fig. 29. Fitting of Ag:a-Si weight update data with normalized A in the plot of normalized conductance vs. normalized number of pulses.

The shape of nonlinear weight update curves can look very different with the same A but different P_{max} and G_{min}, because A has to be scaled according to different P_{max}. First, we normalize P_{max} to be 1 by default definition, then we can tune the normalized A and the cycle-to-cycle weight update variation for both LTP and LTD in the generated Fig. 28 (normalized conductance vs. normalized number of pulses) to find the best fit, as shown in **Error! Reference source not found.**. A good procedure is that the user first finds out a reasonable normalized A for LTP and LTD curves without variation (by setting the variation in LTP and LTD to zero), and then try to fit the LTP and LTD data with good variation values and pseudorandom seeds (for example, **rng(103)** and **rng(898)** in script). In the script, the A values are defined as **A_LTP** and **A_LTD**, and P_{max} is defined as **xf** which is set to be 1. These parameters are shown in **Error! Reference source not found.**. It should be noted that the device-to-device weight update variation is not considered

```
192 -    xf = 1;
193 -    A_LTP = 0.5;
194 -    B_LTP = 1./(1-exp(-1./A_LTP));
195 -    A_LTD = -0.2;
196 -    B_LTD = 1./(1-exp(-1./A_LTD));
198      % LTP fitting
199 -    var_amp = 0.035;    % LTP cycle-to-cycle variation
200 -    rng(103);
217      % LTD fitting
218 -    var_amp = 0.025;    % LTD cycle-to-cycle variation
219 -    rng(898);
```

Fig. 30. Code snippet of parameters in “nonlinear_fit.m”.

in this script.

9. How to run *DNN +NeuroSim*

1) Define Network Structure in NetWork.csv

Firstly, the users have to define network structure in the NetWork.csv file, such that the NeuroSim will process the floorplan and define the hardware design. Taking the default VGG-8 with 8 layers as an example, the definition of each cell in the excel table in shown below, in the NetWork.csv file, only the numbers are supposed to be filled in, i.e. the texts cannot be written in the file, it is important to accurately modify the table to avoid segmentation fault.

Table II NetWork.csv

	IFM Length	IFM Width	IFM Channel Depth	Kernel Length	Kernel Width	Kernel Depth	Followed by pooling or not?	Stride
Layer 1	32	32	3	3	3	128	0	1
Layer 2	32	32	128	3	3	128	1	1
Layer 3	16	16	128	3	3	256	0	1
Layer 4	16	16	256	3	3	256	1	1
Layer 5	8	8	256	3	3	512	0	1
Layer 6	8	8	512	3	3	512	1	1
Layer 7	1	1	8192	1	1	1024	0	1
Layer 8	1	1	1024	1	1	10	0	1

In the default VGG-8 network, layer 1 to layer 6 are convolutional layers, and layer 7 to layer 8 are fully-connected layers. In the Table II, the dimensions of each layer are defined in different rows, from layer 1 to layer 8 (row 1 to row 8), while the first three columns (column 1 to column 3) are used to define the dimension of input feature maps (IFMs) of each layer. For example, the input image size of layer 1 is 32×32×3, thus, in first row, the first three cells should be filled by 32, 32 and 3 respectively, which indicated the length, width and depth of the IFM. The next three columns (column 4 to column 6) are used to define the dimension of kernels. For example, the kernel size of layer 3 is 3×3×128×256 (i.e. each single 3D kernel is 3×3×128, the kernel depth is 256), since it is well known that the third dimension of kernel is defined by the IFM channel depth, it is not necessary to define the third dimension again, thus, from the Table II, in row 3, the fourth, fifth and sixth cell should be filled by 3, 3 and 256, which represent the length, width and kernel depth (first, second and fourth dimension of kernel) respectively. One should notice that, the fully-connected layer can also be represented in the similar way, by considering it as a special convolutional layer, which has unit length and width for IFM and kernels. The last column is used to define whether the current layer is followed by pooling, it will be read by *NeuroSim*, and properly estimate the hardware performance for pooling function, in this framework, the activation function is considered to be integrated in every layer.

2) Modify the hardware parameters in Param.cpp

After setting up the network structure, the users need to define the hardware parameters in **Param.cpp**. In this file, the users could define the parameters, such as technology node (**technode**), device type (**memcelltype**: SRAM, eNVM or FeFET), operation mode (**operationmode**: parallel or sequential read-out, synaptic sub-array size (**numRowSubArray, numColSubArray**), synaptic device precision (**cellBit**), mapping method (conventional or novel), activation type (sigmoid or ReLU), cell height/width in feature size (F), clock frequency and so on.

In this framework, all the hardware parameters that users need to define are summarized in the **Param.cpp**, thus, to successfully run the simulator, the two main files users need to visit are **NetWork.csv** and **Param.cpp**.

3) Compilation of *NeuroSim*

After modifying the **NetWork.csv** and **Param.cpp** files, or whenever any change is made in the files, the codes have to be recompiled by using `make` command as stated in **Installation and Usage (Linux)** section. If the compilation is successful, a screenshot like Fig. 31 can be expected.

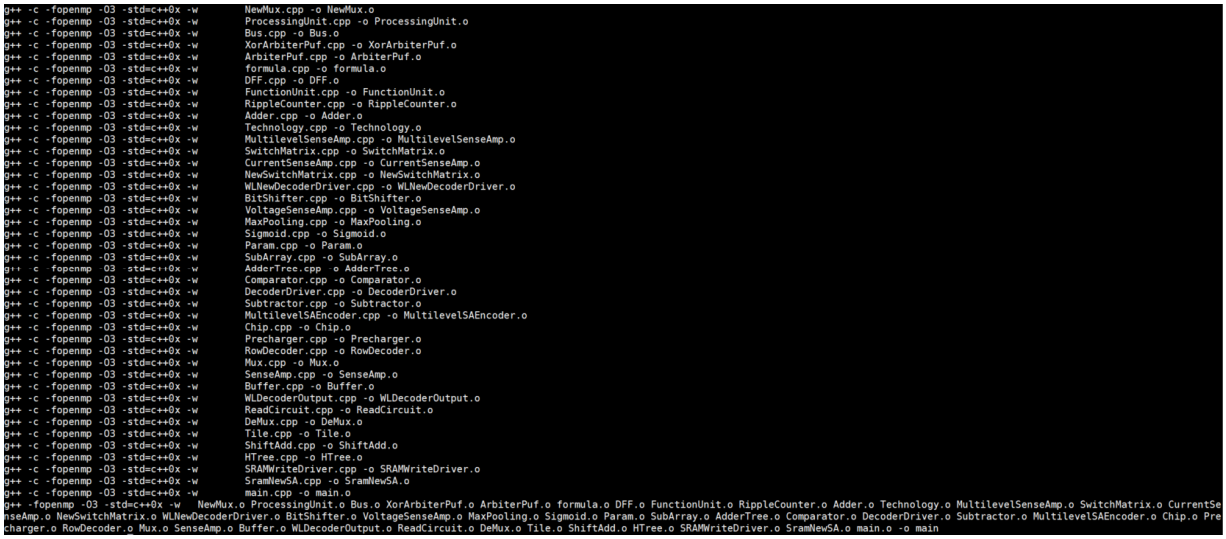


Fig. 31 Output of compilation.

4) Run the program with PyTorch/TensorFlow wrapper

After compilation of *NeuroSim*, go back to the Python wrapper, in the wrapper, there is a VGG-8 as default, the users can modify their network structures, and run the simulator correspondingly.

Instructions to run the wrapper:

- PyTorch (https://pytorch.org/)
  - The bitwidth could be set use optional parameter
  - Train
    - Python train.py
    - *NeuroSim* simulation is integrated with "train.py", to limit simulation time, by default, we call *NeuroSim* once per epoch (256 epochs in total in default defindation)
    - The model will be saved at a hierarchical folders based one the option value.
    - Please expect whole simulation to take ~12 hours
  - Inference
    - Python inference.py (for inference engine benchmarking as in V1.1)
    - Set model_path to the saved model *.pth file
    - Python inference.py (integrated with *NeuroSim*)

The program will print out the results for each layer of the network during the simulation. The simulation using V1.1 for inference will approximately take 5 minutes with a computer workstation (Intel 8-core CPU with 3.2 GHz and NVidia Titan V GPU) for the VGG-8 network. The simulation using V2.0 for on-chip training (with hardware constraints) will approximately take 12 hours (with 256 epochs) for the VGG-8 network. To limit simulation time, by default, we call NeuroSim once per epoch (each *NeuroSim* run will involve huge data process of weight, delta weight and input activations). The estimation results of both training accuracy (from Python wrapper) and hardware performance (from *NeuroSim* core) will be printed out in the terminal at each epoch, meanwhile, more detailed results (including weight distribution, breakdown of main hardware components) will be automatically saved in different csv files under the simulation path. An example of V2.0 output is shown below in Fig. 32, Fig. 33 and Fig. 34.

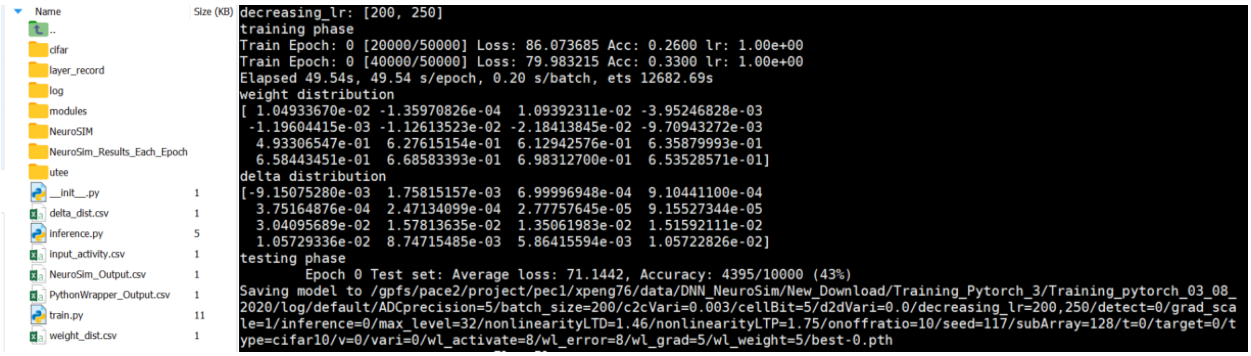


Fig. 32 Output (file list, training accuracy and data distribution) of DNN+NeuroSim V2.0.

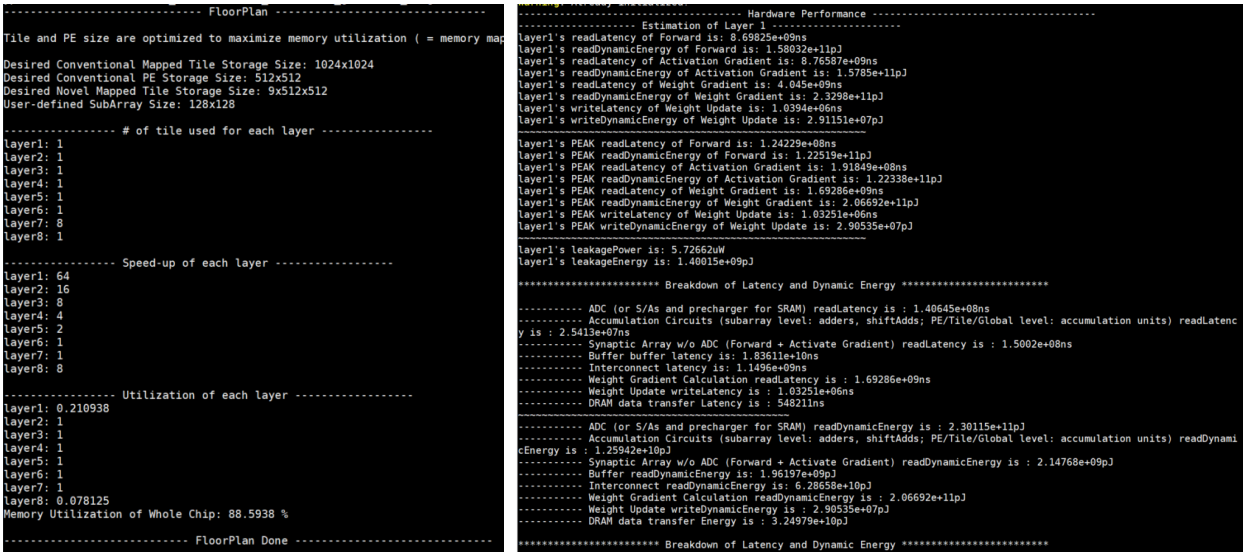


Fig. 33 Output (*NeuroSim*: floorplan, breakdown of each layer) of DNN+NeuroSim V2.0.

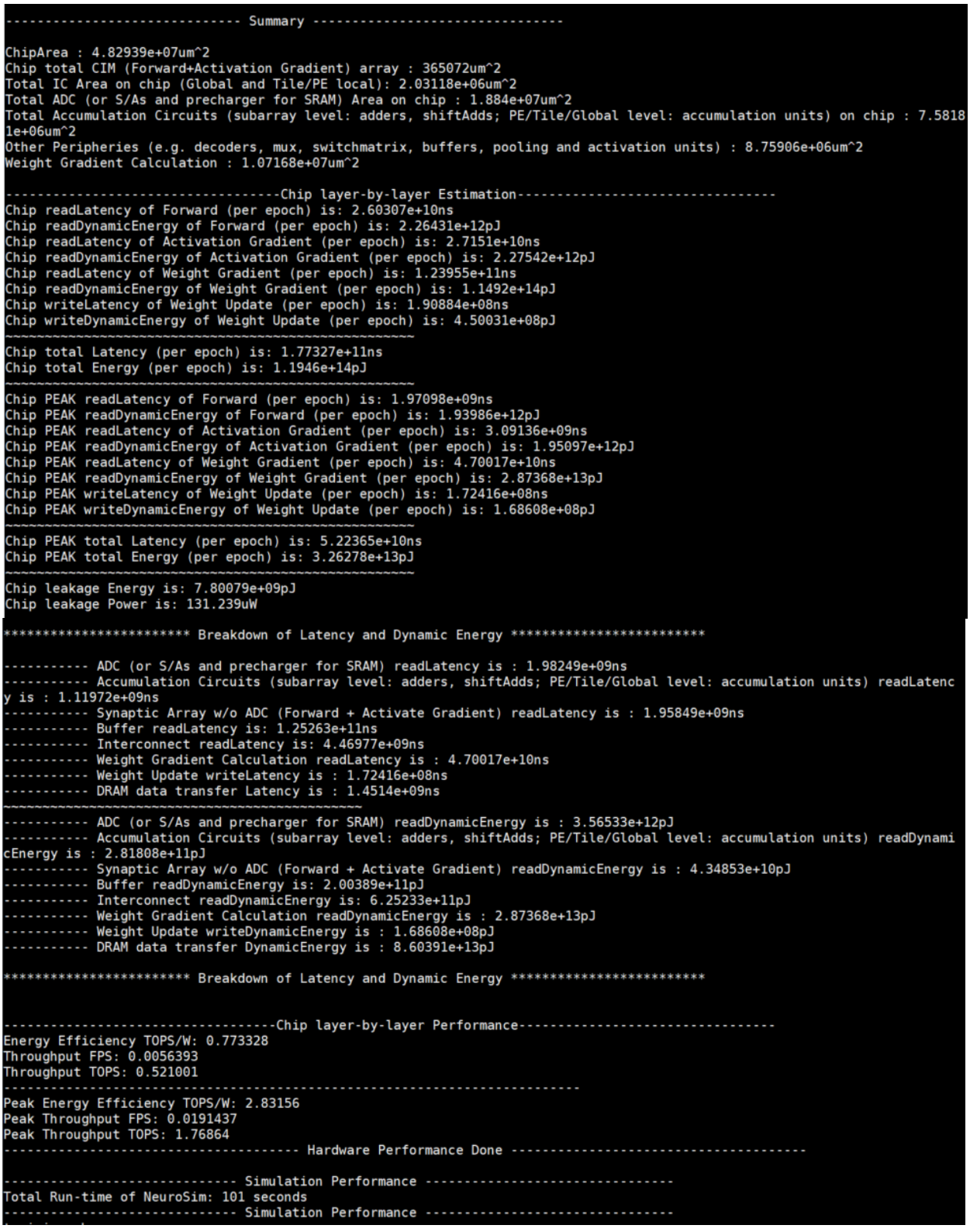


Fig. 34Output (*NeuroSim*: hardware estimation summary) of DNN+NeuroSim V2.0.

10. Reference

[1]. 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), 2019.

[2]. 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, 2018.

[3]. M. O'Connor et al., "*Fine-Grained DRAM: Energy-Efficient DRAM for Extreme Bandwidth Systems*," 2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Boston, MA, USA, 2017, pp. 41-54.

[4]. X. Peng, S. Huang, H. Jiang, A. Lu and S. Yu, "*DNN+ NeuroSim V2.0: An End-to-End Benchmarking Framework for Compute-in-Memory Accelerators for Training*," arXiv, 2020.

[5]. N. E. Weste and D. Harris, "*CMOS VLSI Design – A Circuit and Systems Perspective, 4th edition*," 2007.

[6]. 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), 2019.

[7]. P.-Y. Chen, et al., "*Technology-design co-optimization of resistive cross-point array for accelerating learning algorithms on chip*," ACM/IEEE Design, Automation & Test in Europe Conference & Exhibition (DATE), 2015.

[8]. W. Khwa et al., "*A 65nm 4Kb algorithm-dependent computing-in-memory SRAM unit-macro with 2.3ns and 55.8TOPS/W fully parallel product-sum operation for binary DNN edge processors*," IEEE International Solid State Circuits Conference (ISSCC), 2018.

[9]. J. Seo et al., "*A 45nm CMOS neuromorphic chip with a scalable architecture for learning in networks of spiking neurons*," 2011 IEEE Custom Integrated Circuits Conference (CICC), San Jose, CA, 2011, pp. 1-4.

[10]. M. Jerry, et al., "*Ferroelectric FET analog synapse for acceleration of deep neural network training*," IEEE International Electron Devices Meeting (IEDM), 2017.

[11]. P. Wang, W. Shim, Z. Wang, J. Hur, S. Datta, A. I. Khan, S. Yu, "*Drain-erase scheme in ferroelectric field effect transistor-Part II: 3D-NAND architecture for in-memory computing*," IEEE Trans. Electron Devices, vol. 67, no. 3, pp. 962-967, 2020.

[12]. X. Sun, S. Yin, X. Peng, R. Liu, J.-S. Seo, S. Yu, "*XNOR-RRAM: A scalable and parallel resistive synaptic architecture for binary neural networks*," ACM/IEEE Design, Automation & Test in Europe Conference (DATE), 2018.

[13]. S. Wu, et al. "*Training and inference with integers in deep neural networks*," arXiv: 1802.04680, 2018.

[14]. github.com/boluoweifenda/WAGE

[15]. github.com/stevenygd/WAGE.pytorch

[16]. github.com/aaron-xichen/pytorch-playground

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