User Manual of MLP simulator NeuroSimV3.0

Developers: Pai-Yu Chen, Xiaochen Peng and Yandong Luo

PI: Prof. Shimeng Yu, Georgia Institute of Technology

If you have logistic questions or comments on the model, please contact Prof. Shimeng Yu, and if you have technical questions or comments, please contact Xiaochen Peng or Yandong Luo.

This research is supported by NSF CAREER award, NSF/SRC E2CDA program, and ASCENT, one of the SRC/DARPA JUMP centers.

If you use the tool or adapt the tool in your work or publication, you are required to cite the following reference:

P.-Y. Chen, X. Peng, S. Yu, "NeuroSim+: An integrated device-to-algorithm framework for benchmarking synaptic devices and array architectures", IEEE International Electron Devices Meeting (IEDM), 2017, San Francisco, USA.

Index

---

1. Introduction

MLP simulator (*+NeuroSim*) is developed in C++ to emulate the online learning/offline classification scenario with MNIST handwritten dataset in a 2-layer multilayer perceptron (MLP) neural network based on SRAM, emerging non-volatile memory (eNVM) and ferroelectric FET (FeFET) array architectures. The eNVM in this simulator refers to a special subset of resistive memory devices that can tune the conductance into multilevel states with voltage stimulus. *NeuroSim* is a circuit-level macro model for benchmarking neuro-inspired architectures in terms of circuit-level performance metrics, such as chip area, latency, dynamic energy and leakage power. Without *NeuroSim*, *MLP simulator* can be regarded as a standalone functional simulator that is able to evaluate the learning accuracy and the circuit-level performance (but only for the synapse array) during learning. With NeuroSim, *MLP simulator* (*+NeuroSim*) becomes an integrated framework with hierarchical organization from the device level (transistor, eNVM and FeFET device properties) to the circuit level (array architectures with periphery circuit modules) and then to the algorithm level (neural network topology), enabling instruction-accurate evaluation on the learning accuracy as well as the circuit-level performance metrics at the run-time of learning.

The target users for this simulator are device engineers who wish to quickly estimate the system-level performance with his/her own analog synaptic device data. The users are expected to have the weight update characteristics (conductance vs. # pulse) ready in hand. Device-level parameters such as number of levels, weight update nonlinearity, device-to-device variation, and cycle-to-cycle variations, could be extracted using the MATLAB script that is provided. At the circuit level, several design options are available, such as the analog synaptic array architecture (eNVM crossbar, eNVM pseudo-crossbar or FeFET), or digital synaptic array architecture (eNVM crossbar, eNVM 1T1R, or SRAM). At the algorithm level, a simple 2-layer MLP neural network is provided for evaluation, thus only limited options are available to the users to modify, such as the size of each layer and the size of weight matrices.

---

2. New Feature Highlights in Version 3.0.

Key changes in this released v3.0 are summarized as follows. The details can be referred to the later sections.

1) Change the algorithm weight from (0,1) in V2.0 to (-1,1) in V3.0

In this NeuroSimV3.0, the weight range is changed to -1 to 1 to make it consistent with the typical definition in most of neural network algorithms, while the weight range is 0 to 1 in the NeuroSimV2.0. Now, In V3.0 each programming pulse changes the weight two times more than that in V2.0 although the device conductance is modified by the same amount. In other words, the programming pulse becomes more “coarse” in V3.0. Therefore, for analog eNVMs with relatively fewer conductance state, an accuracy drop is observed in V3.0 compared with V2.0

This feature is automatically embedded into the training and test process. You do not need to specify any parameters for this new feature.

2) Add array architecture to support parallel read-out for digital eNVMs

In NeuroSimV2.0, the read-out for NVM based digital synapse array (e.g. STT-MRAM) is conducted row by row. In this V3.0, a parallel read-out architecture is proposed as Fig. 0 (a). The WL-BL switch matrix turns on all the WL and BL simultaneously. The total weight of a column is converted into digital bits by the readout circuits (e.g. sense amplifiers). The two reference columns (that are programmed to be on and off-states, respectively) are needed for STT-MRAM because of its low on-

off ratio. The detailed parallel read-out scheme is illustrated in **section 6.2.** To enable the parallel read-out, please set the parameter "parallel" to "true" in the constructor for digitalNVM under Cell.cpp.

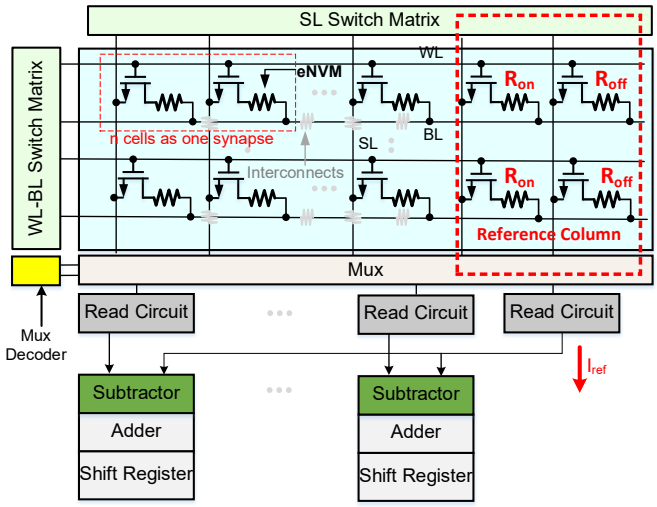


Fig. 0 (a) A parallel read-out scheme for eNVM based digital synapse

3) Add a virtual reference current to represent negative weight

To represent negative weight, the digital bits converted from the reference columns are subtracted from the digital bits converted from the actual columns. This applies to all kinds of eNVM based synaptic array architectures automatically.

4) Add more training algorithms into the MLP simulator (optimization_type in param.cpp)

Training algorithms such as "Momentum", "Adgrad", "RMSprop", "Adam" is added into the MLP simulator. These algorithms may show better convergence and accuracy.

This feature can be specified by changing the parameter "optimization_type" under param.cpp. The available options are as follows:

a. "SGD": the stochastic gradient descent which is the algorithm used by V1.0 and V2.0.

b. "Momentum": the momentum method.

c. "Adagrad": the self-adaptive gradient descent method. The learning rate will monotonically decay as the training goes on.

d. "RMSprop". It is a unpublished adaptive learning rate method proposed by Geoffrey Hinton The learning rate of weight w_i is divided by the average of w_i's recent gradient.

e. "Adam": Adaptive Moment Estimation.

The details for the above algorithms can be found in the following link:
http://ruder.io/optimizing-gradient-descent/index.html#momentum

The accuracy with these algorithms is shown below in Fig. 00. It is noticed that the convergence for analogNVMs (such as Ag-Si) with poor linearity can be improved by properly choosing the training algorithm. For analog NVMs with good linearity (such as EpiRAM), the accuracy can be further improved.

EpiRAM is referred to this paper: S. Choi, S. H. Tan, Z. Li, Y. Kim, C. Choi, P.-Y. Chen, H. Yeon, S. Yu, J. Kim, "SiGe epitaxial memory for neuromorphic computing with reproducible high performance based on engineered dislocations," Nature Materials, vol.17, pp. 335-340, 2018.

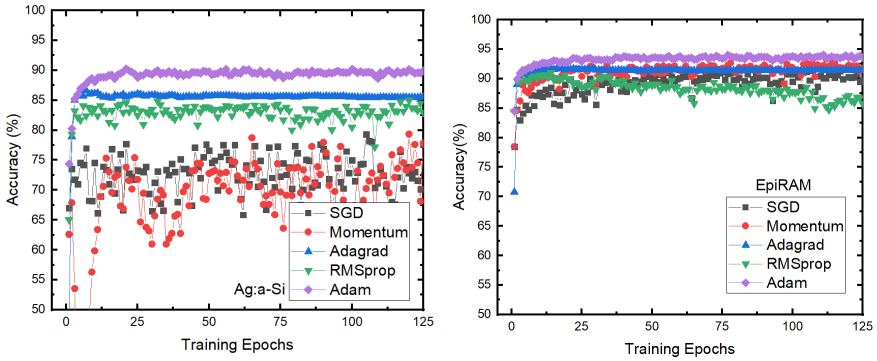


Figure 00. training accuracy vs. epochs for Ag-Si and EpiRAM

It should be noted that the hardware support for these algorithms have not been added into NeuroSimV3.0. This feature is only at the algorithm level. Hardware support may be added into the future version.

---

3. 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.

---

4. Installation and Usage (Linux)

Step 1: Get the tool from GitHub

```
git clone https://github.com/neurosim/MLP_NeuroSim_V3.0.git
```

Step 2: Extract **MNIST_data.zip** to it's current directory

```
unzip MNIST_data.zip
```

Step 3: Compile the codes

```
make
```

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 codes and build the "main" program
`make clean`	Clean up the directory by removing the object files and the "main" executable
`./main`	Run simulation (after `make`)
`make run`	Run simulation (after `make`), and the results will be saved to a log file (filename appended with the current time info). This command does not work if "stdbuf" is not found.

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

---

5. Device Level: Synaptic Device Characteristics

5.1 Non-ideal Analog eNVM Device Properties

As shown in Fig. 1, the framework considers the following non-ideal analog eNVM device properties:

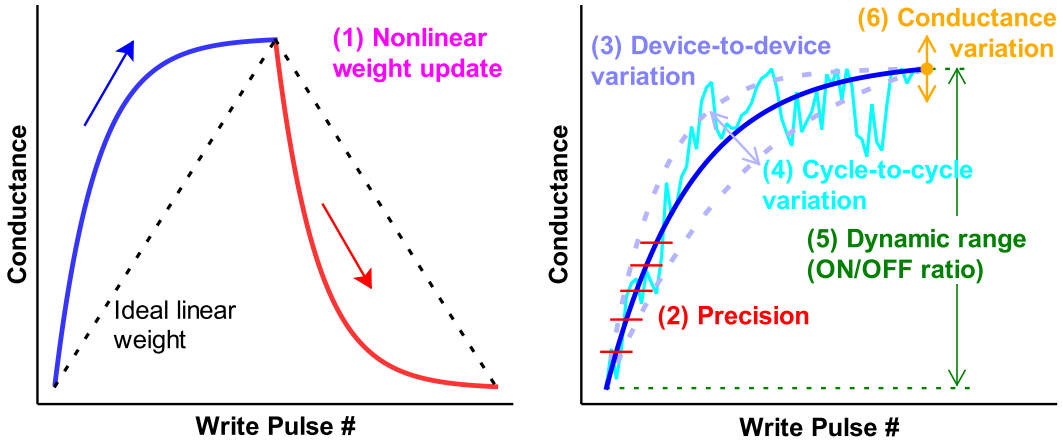


Fig. 1 Summary of non-ideal analog eNVM device properties.

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:

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

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

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

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. 1, 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. (1)-(3), a set of nonlinear weight increase (blue) and weight decrease (red) behavior can be obtained by adjusting A as shown in Fig. 2, where each nonlinear curve is labeled with a nonlinearity value from +6 to -6. It can be proved that Eq. (1) and (2) are equivalent with a different sign of A, thus we will just use Eq. (1) to calculate both nonlinear LTP and LTD weight update. Different than Fig. 1, all LTD curves are mirrored and shifted horizontally to make sure the curve starting from the pulse number 0 for simpler formulization.

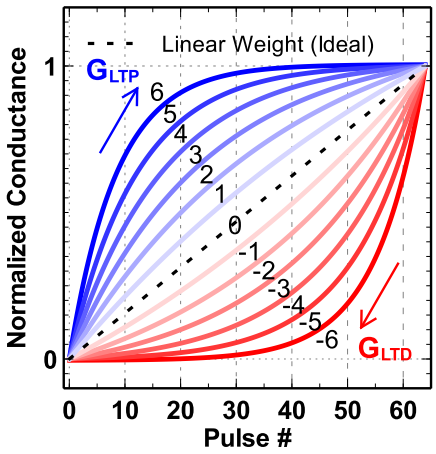


Fig. 2 Analog eNVM device behavioral model of the nonlinear weight update with the nonlinearity labeled from -6 to 6.

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. (1)-(3).

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 Fig. 2.

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.

5) Dynamic range (ON/OFF ratio)

Ideally, the weight values are represented by a normalized conductance of analog eNVM devices with the range from 0 to 1. However, the minimum conductance can be regarded as 0 only when the ratio between

the maximum and minimum conductance (ON/OFF ratio) approaches infinity. With limited ON/OFF ratio, the cells with weight=0 still have leakage.

6) Conductance variation

Different devices may observe different ON/OFF ratios if the conductance range has a variation. The conductance variation (σ) is typically expressed in terms of the percentage of the highest conductance state (ON state) as it changes the conductance range most.

5.2 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 the following reference as an example.

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.

Before the fitting, the user has to make sure the experimental weight update data are pre-processed in a format that is similar to Fig. 2. 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. (1)-(3). The user can look at the pre-processed data of Ag:a-Si devices as an example in the MATLAB script.

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 **Figure 1** (normalized conductance vs. normalized number of pulses) to find the best fit, as shown in Fig. 3. 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 Fig. 4. It should be noted that the device-to-device weight update variation is not considered in this script.

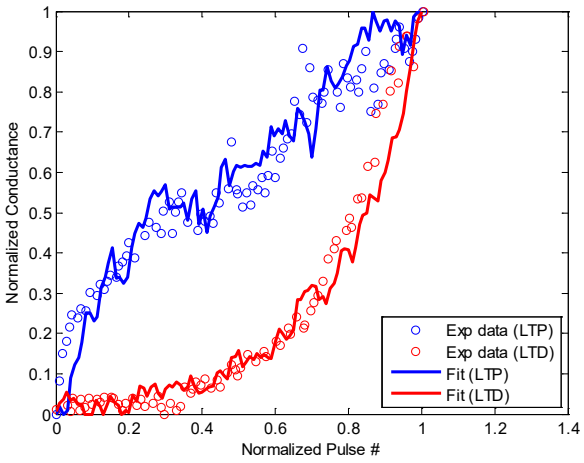


Fig. 3 Fitting of Ag:a-Si weight update data with normalized *A* in the plot of normalized conductance vs. normalized number of pulses.

```
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. 4 Code snippet of parameters in **nonlinear_fit.m**

After the fitting is done, the user can look up the magnitude of the normalized *A* value to figure out its corresponding nonlinearity label. We provide the information of one-to-one mapping of the nonlinearity label values to the normalized *A* values in the file **Nonlinearity-NormA.htm**. The nonlinearity label value is from 0 to 9 with a step of 0.01, which is precise enough. If the normalized *A* value is negative, the nonlinearity label value will also simply be negative. In the example of Ag:a-Si, the fitted normalized *A* values are 0.5 and -0.2, and we can figure out their corresponding nonlinearities are 2.40 and -4.88 for LTP and LTD, respectively, as shown in Table 1. In the next section, *MLP simulator* (*+NeuroSim*) will take these two values as eNVM cell parameters.

Nonlinearity	Norm. A	Nonlinearity	Norm. A	Nonlinearity	Norm. A	Nonlinearity	Norm. A	Nonlinearity	Norm. A
2.31	0.5207	2.38	0.5038	2.45	0.4879	4.84	0.2030	4.91	0.1983
2.32	0.5183	2.39	0.5015	2.46	0.4856	4.85	0.2023	4.92	0.1976
2.33	0.5158	2.40	0.4992	…	…	4.86	0.2016	4.93	0.1970
2.34	0.5134	2.41	0.4969	…	…	4.87	0.2010	4.94	0.1963
2.35	0.5110	2.42	0.4946	…	…	4.88	0.2003	4.95	0.1957
2.36	0.5086	2.43	0.4932	4.82	0.2044	4.89	0.1996	4.96	0.1950
2.37	0.5062	2.44	0.4901	4.83	0.2037	4.90	0.1990	4.97	0.1944

Table 1 Snippet of **Nonlinearity-NormA.htm**

5.3 Device Types and Parameters

In the simulator, we provide several types of the synaptic devices, which can be analog or digital. Analog synaptic devices represent the weight with their "analog" multi-level conductance states, while digital synaptic devices are binary memory devices and they have to be grouped together to represent the weight precision. Available device classes and their detailed device properties are enclosed in **Cell.cpp**, and the users will choose which device class to instantiate in **main.cpp**. Below is the summary of these device classes:

Device class	**Type**	**Description**
`RealDevice` (default)	Analog	This analog eNVM class has all the options of non-ideal synaptic device properties discussed earlier. The default values in this class are for the Ag:Si example.

`IdealDevice`	Analog	This device class is a subset of `RealDevice` that does not have non-ideal synaptic device properties in weight update.
`MeasuredDevice`	Analog	This device class uses the look-up table method rather than parameters to reproduce the weight update curves, thus the weight update variations are not applicable here.
`DigitalNVM`	Digital	The binary eNVM class.
`SRAM`	Digital	The binary SRAM class.

All the parameters in **Cell.cpp** have the comments to describe their meaning, and here we introduce the important or common ones that are used in more than one device class. For resistive synaptic devices, the **maxConductance** and **minConductance** are defined as $1/R_{ON}$ and $1/R_{OFF}$, respectively. **readVoltage** and **readPulseWidth** are on-chip read voltage (V) and read pulse width (s). The specified value of **readPulseWidth** does not matter because it will be modified later by the read circuit module when it calculates the required pulse width for each integration cycle based on the ADC precision. **writeVoltageLTP** and **writePulseWidthLTP** are the write voltage (V) and the write pulse width (s) during LTP or weight increase. **writeVoltageLTD** and **writePulseWidthLTD** are also defined in the same way. For the non-ideal device properties, we describe their implementation below one by one with the associated parameters:

1) Nonlinear weight update

To enable this property, the users need to make sure **nonlinearWrite=true**. Then, the users have to provide the value of **NL_LTP** and **NL_LTD**, which represent the weight update nonlinearity for LTP and LTD, respectively. In the example of Ag:a-Si, **NL_LTP** and **NL_LTD** are set to 2.40 and -4.88, as obtained from the MATLAB fitting results.

2) Limited precision

The number of conductance states can be specified in **maxNumLevelLTP** and **maxNumLevelLTD** for the LTP and LTD of the analog synaptic device, respectively. On the other hand, the weight precision with digital synaptic devices is specified in **numWeightBit** in **Param.cpp**.

3) Device-to-device weight update variation

The standard deviation (σ) of device-to-device variation is specified in **sigmaDtoD**. If this property is not considered, **sigmaDtoD** should be set to 0.

4) Cycle-to-cycle weight update variation

The standard deviation (σ) of cycle-to-cycle variation is specified in **sigmaCtoC**. It is multiplied with (**maxConductance** - **minConductance**) because it is expressed in terms of the percentage of entire conductance range as mentioned earlier. Currently the simulator only takes one value of the cycle-to-cycle weight update variation for **sigmaCtoC**. It is encouraged that the user selects the larger one in LTP and LTD for conservative estimation. If this property is not considered, **sigmaCtoC** should be set to 0.

5) Dynamic range (ON/OFF ratio)

The dynamic range is solely determined by **maxConductance** and **minConductance**. There is no additional parameter to enable this property. However, if the users would not like to take this effect into account, it is fine to set **minConductance=0** to obtain an infinite ON/OFF ratio.

6) Conductance variation

To enable this property, the users need to make sure **conductanceRangeVar=true**. Then, the users have to provide the value of **maxConductanceVar** and **minConductanceVar**, which represents the standard deviation (σ) of conductance variation at maximum and minimum conductance state in terms of percentage, respectively. If the ON/OFF ratio is large, setting **maxConductanceVar** alone is good enough.

7) Read noise

To enable this property, the users need to make sure **readNoise=true**. Then, the users have to provide the value of **sigmaReadNoise**, which is the standard deviation of read noise in gaussian distribution.

For other modes or parameters, **cmosAccess** is used to choose the cell structure, or synaptic core type in other words. **cmosAccess=true** means the pseudo-crossbar/1T1R array, while **cmosAccess=false** means the true crossbar array. If the cell is pseudo-crossbar/1T1R, we need to define **resistanceAccess**, which is the turn-on resistance value of the transistor in 1T1R array. The **FeFET** option is for the ferroelectric FET configuration. We do not have a dedicated device class for FeFET because it is similar to the analog eNVM from the non-ideal device properties' point of view. Its default configuration is **FeFET=false**. If **FeFET=true**, we need to provide the value of **gateCapFeFET**, which is the gate capacitance of FeFET. If the cell is crossbar. I-V nonlinearity **NL** can be specified as the current ratio between write voltage and half write voltage considering if a selector is added. To enable this property, the users have to set **nonlinearIV=true**. The **nonIdenticalPulse** option is for non-identical write pulse scheme where the write pulse amplitude or width linearly increases or decreases with the pulse number. As shown in Fig. 5, **VinitLTP**, **VstepLTP**, **VinitLTD**, **VstepLTD**, **PWinitLTP**, **PWstepLTP**, **PWinitLTD** and **PWstepLTD** are essential parameters that need to be defined by the users when **nonIdenticalPulse=true**.

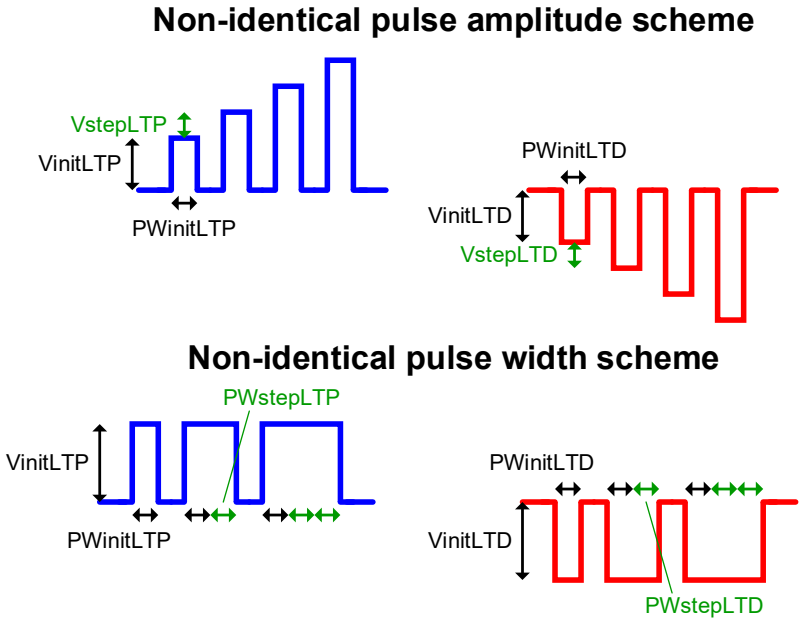


Fig. 5 Non-identical write pulse scheme for weight update

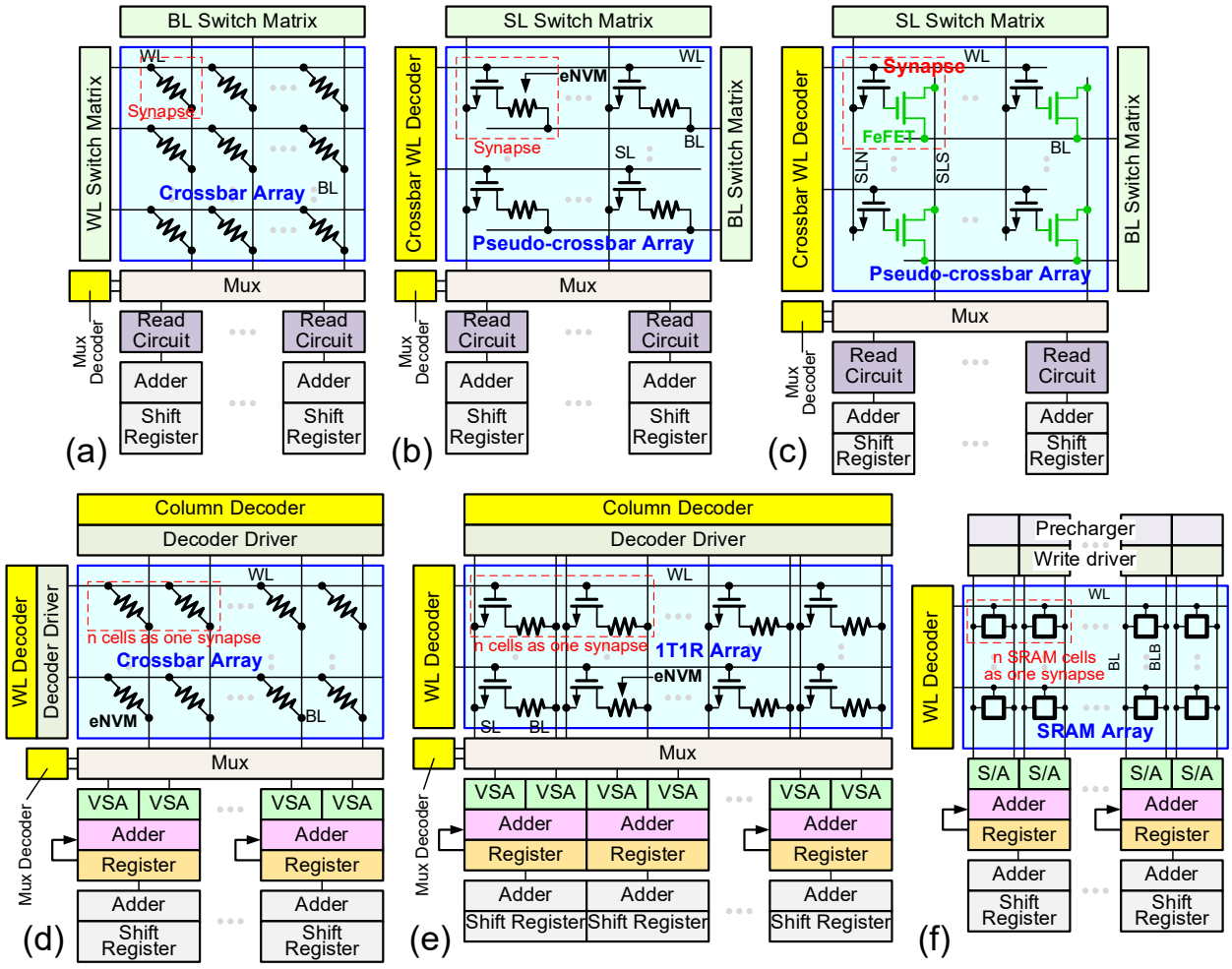
For the **MeasuredDevice** class, the users have to fill in the LTP and LTD conductance data in **rawDataConductanceLTP** and **rawDataConductanceLTD**, respectively. Example inputs have been provided in **Cell.cpp**. If the LTP and LTD data are symmetric, the users can just fill in the

rawDataConductanceLTP and set **symLTPandLTD=true**. It should be noted that this device class is still experimental and some restrictions on the data may apply. For example, the conductance range of LTP and LTD should be consistent, and it would be better that the LTP and LTD conductance data are monotonically increasing and decreasing, respectively. The simulator will automatically check if the provided data satisfy the above conditions. If the users do not bother having these checks, they can comment them out at the bottom of the **MeasuredDevice** class. Also, the weight update variations cannot be simulated in this class.

---

6. Circuit Level: Synaptic Cores and Array Architectures

In this framework, we consider two synaptic cores of 2-layer MLP in the evaluation for circuit-level metrics such as area, latency, energy, etc. A synaptic core is a computation unit that is specifically designed for weighted sum and weight update. It consists of the synaptic array and array periphery. In the simulator, a synaptic core can be instantiated from **SubArray** class in **SubArray.cpp**. In this released version, there are six available design options for the synaptic core, as shown in Fig. 6. The details of these architectures and their peripheral circuits are introduced below.



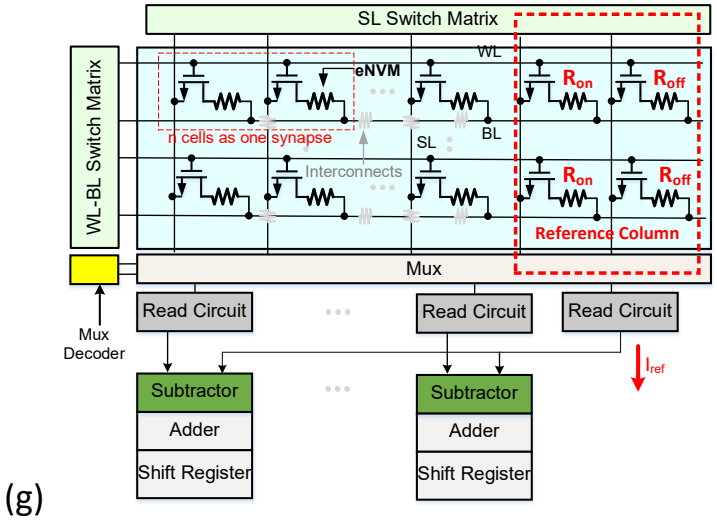


Fig. 6 Synaptic cores based on (a) analog eNVM crossbar parallel read-out, (b) analog eNVM pseudo-crossbar parallel read-out, (c) analog FeFET parallel read-out, (d) digital eNVM crossbar row-by-row read read-out, (e) digital eNVM pseudo-crossbar row-by-row read read-out, (f) digital SRAM array row-by-row read out and (g) digital eNVM pseudo-crossbar parallel read-out

6.1 Analog Synaptic Array Architectures

Analog synaptic devices can represent the weight with their "analog" multi-level conductance states. These include analog eNVM or FeFET devices. In the simulator, the analog synaptic array architectures will be instantiated if either **RealDevice**, **IdealDevice** or **MeasuredDevice** is designated in **main.cpp**.

1) 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). If the input vector is encoded by read voltage signals, the weighted sum operation (matrix-vector multiplication) can be performed in a parallel fashion with the crossbar array. However, as there is no isolation between cells, it is necessary to apply some voltage (smaller than the programming voltage, e.g. V/2) at all the unselected rows and columns to prevent the write disturbance on unselected cells during weight update. The voltage bias schemes for weight update is shown in Fig. 7. In addition, a two-terminal threshold switching selector device is desired to minimize the write disturbance and sneak path problem. In our model, a simple I-V nonlinearity (**NL**) in **Cell.cpp** is used to define the effect of selector or built-in self-selection. As the weight increase and decrease need different programming voltage polarities, the weight update process requires 2 steps with different voltage bias schemes. In weight update, the selected cells will be on the same row, and programming pulses or biases (if no update) are provided from the BL, allowing the selected cells to be tuned differently in parallel. To perform weight update for the entire array, a row-by-row operation is necessary. Ideally, the entire row is selected at a time to ensure the maximum parallelism. In the simulator, the crossbar array architecture can be designated by setting **cmosAccess=false** in **Cell.cpp**.

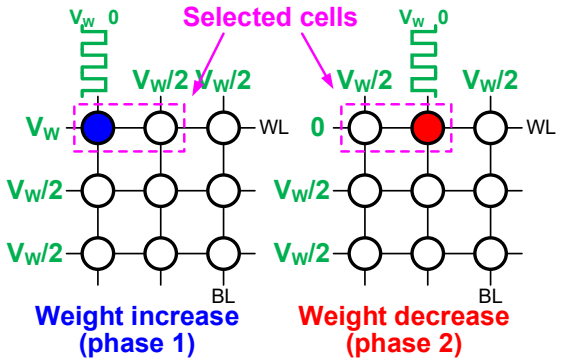


Fig. 7 Voltage bias scheme in the write operation of analog eNVM crossbar array. Two separate phases for weight increase and decrease are required. In this example, the left cell of the selected cells will be updated in phase 1, while the right one will be updated in phase 2.

2) Analog eNVM pseudo-crossbar array

Another common design solution to the write disturbance and sneak path problem is to add a cell selection transistor in series with the eNVM device, forming the one-transistor one-resistor (1T1R) array architecture, as shown in Fig. 8(a). 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, 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. 8(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. The weight update operation in pseudo-crossbar array is similar to that in crossbar array, as shown in Fig. 9. As the unselected WLs can turn off the transistors on unselected rows, no voltage bias is required for these unselected BLs thus pseudo-crossbar array can have save a lot of weight update energy compared to the crossbar array. In the simulator, the pseudo-crossbar array architecture can be designated by setting **cmosAccess=true** in **Cell.cpp**.

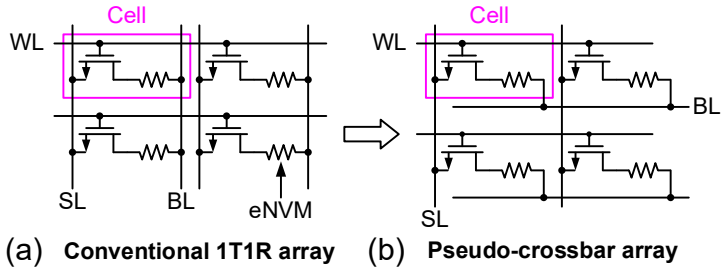


Fig. 8 Transformation from (a) conventional 1T1R array to (b) pseudo-crossbar array by 90° rotation of BL to enable weighted sum operation.

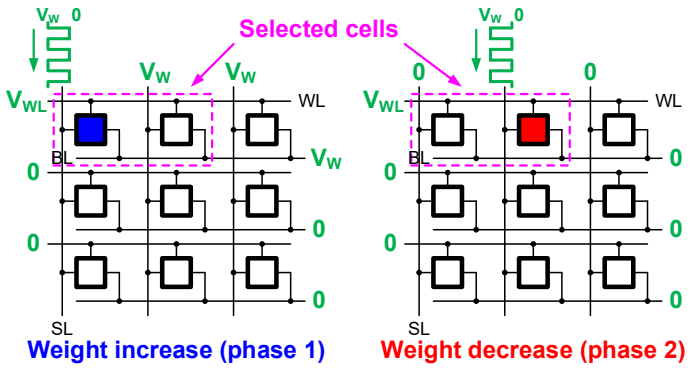


Fig. 9 Voltage bias scheme in the write operation of analog eNVM pseudo-crossbar array. Two separate phases for weight increase and decrease are required. In this example, the left cell of the selected cells will be updated in phase 1, while the right one will be updated in phase 2.

3) Analog FeFET array

As shown Fig. 6(c), the analog FeFET array is in the pseudo-crossbar fashion, which is similar to the analog eNVM pseudo-crossbar one. It also has an access transistor for each cell to prevent programming on other unselected rows during row-by-row weight update. As FeFET is a three-terminal device, it needs two separate SLs for the weighted sum (SLS) and weight update (SLN), respectively. Its weight update operation is shown in Fig. 10. In the simulator, the FeFET array architecture can be designated by setting **cmosAccess=true** and **FeFET=true** in **Cell.cpp**.

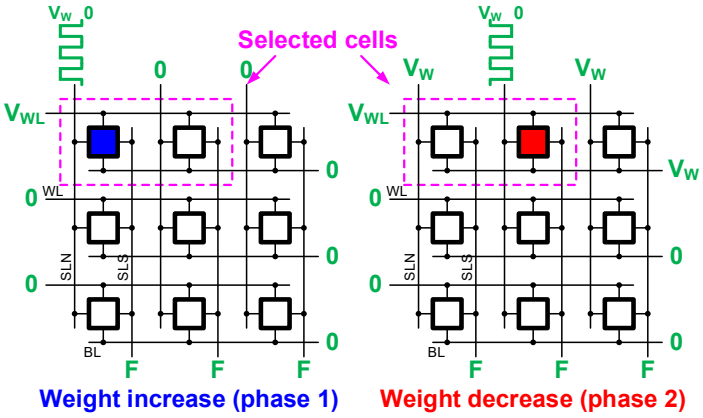


Fig. 10 Voltage bias scheme in the write operation of analog FeFET pseudo-crossbar array. Two separate phases for weight increase and decrease are required. In this example, the left cell of the selected cells will be updated in phase 1, while the right one will be updated in phase 2.

6.2 Digital Synaptic Array Architectures

As the digital synaptic devices can only store binary 0 or 1, many digital synaptic devices must be grouped together to represent the weight precision. In the simulator, digital synaptic devices include binary eNVM or SRAM devices. Thus the digital synaptic array architectures will be instantiated if either **DigitalNVM** or **SRAM** is designated in **main.cpp**.

4) Digital eNVM crossbar/1T1R array with row-by-row read-out

Both digital eNVM crossbar and 1T1R array have very similar architectures, as shown in Fig. 6(d)-(e). Multiple digital eNVM devices are grouped along the row to represent one weight. Different than the analog eNVM array architecture, the weighted sum operation in the digital one is essentially row-by-row based, thus it requires the adder and register to accumulate the partial weighted sum in a row-by-row fashion. On the other hand, the weight update operation in digital eNVM crossbar/1T1R array is also row-by-row based and is similar to the write operation in conventional eNVM array for memory, which is shown in Fig. 11. In the simulator, the crossbar array architecture is selected by setting **cmosAccess=false**, and the 1T1R array architecture is selected by setting **cmosAccess=true**.

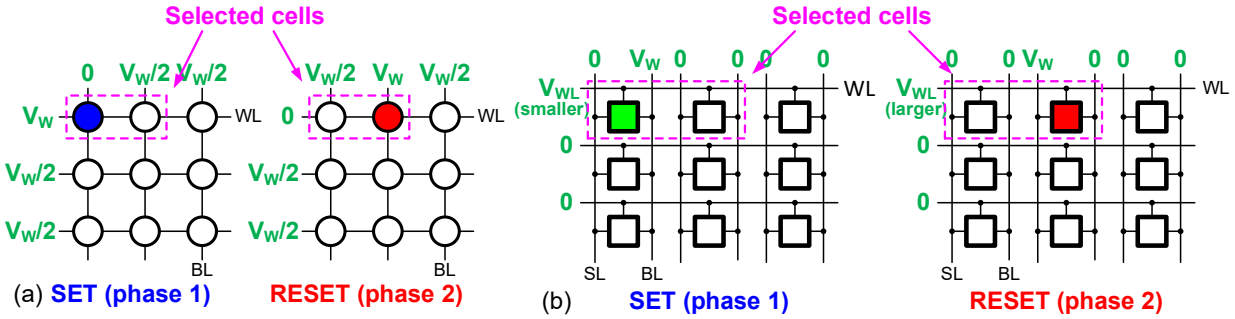


Fig. 11 Voltage bias scheme in the write operation of (a) digital eNVM crossbar and (b) digital eNVM 1T1R array. Two separate phases for SET and RESET are required. In this example, the left cell of the selected cells will be programmed to 1 in SET, while the right one will be programmed to 0 in RESET.

5) Digital eNVM 1T1R array with parallel read-out

In the parallel read-out scheme for digital eNVMs, it is based on 1T1R pseudo-crossbar array. The WL-BL Switch Matrix turn on all the WLs with input "1"s and apply read pulse at the BLs. The actual partial sum of each column is obtained by subtracting the partial sum bits of the reference column from its partial sum bits through the subtractor. The partial sum of different numerical significance is added up through the adder and shift register. The weight update in this architecture operation is still conducted row-by-row. Since the partial sum of a column is read out in parallel, it eliminates the requirement for adder and register as in the row-by-row read-out scheme described above. But the adder and shift register is still needed to add up the partial sums with difference numerical significance.

6) Digital SRAM array

Multiple digital SRAM cells are also grouped along the row to represent one weight, as shown in Fig. 6(f). The weighted sum and weight update operations are similar to the row-by-row read and write operations in conventional SRAM for memory, respectively. 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.

6.3 Array Peripheral Circuits

The periphery circuit modules used in the analog synaptic cores in Fig. 6 are described below:

1) Switch matrix

Switch matrixes are used for fully parallel voltage input to the array rows or columns. Fig. 12(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 not 1 bit, it should be encoded using multiple clock cycles, as shown in Fig. 12(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. In the simulator, all the switch matrixes (**slSwitchMatrix**, **blSwitchMatrix**, **wlSwitchMatrix**) are instantiated from **SwitchMatrix** class in **SwitchMatrix.cpp**. A new WL-BL switch matrix is designed for the parallel read-out architecture for digital eNVMs based array, as shown in Fig. 12 (c). It can be instantiated from the **NewSwitchMatrix** class in **SwitchMatrix.cpp**

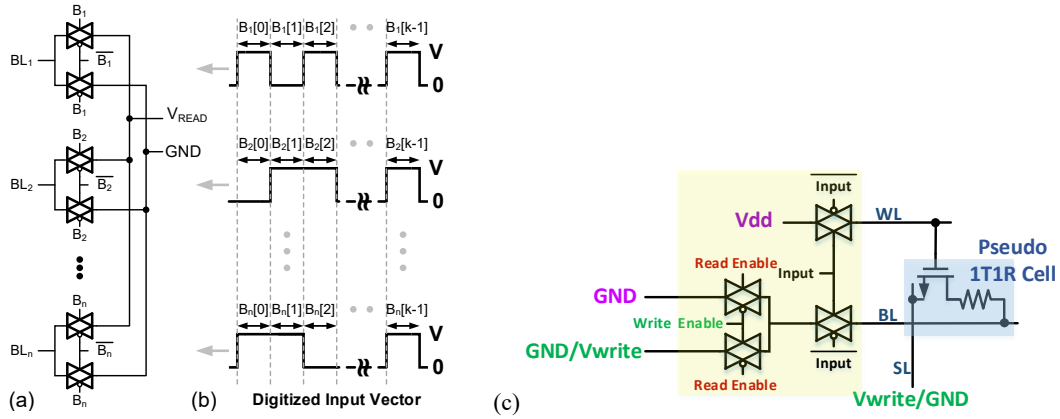


Fig. 12 (a) Transmission gates of the BL switch matrix in the weighted sum operation. A vector of control signals (B_1 to B_n) from the registers (not shown here) decide the BLs to be connected to either a voltage source or ground. (b) Control signals in a bit stream to represent the precision of the input vector. (c). The design of the new WL-BL switch matrix

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. 13. 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**.

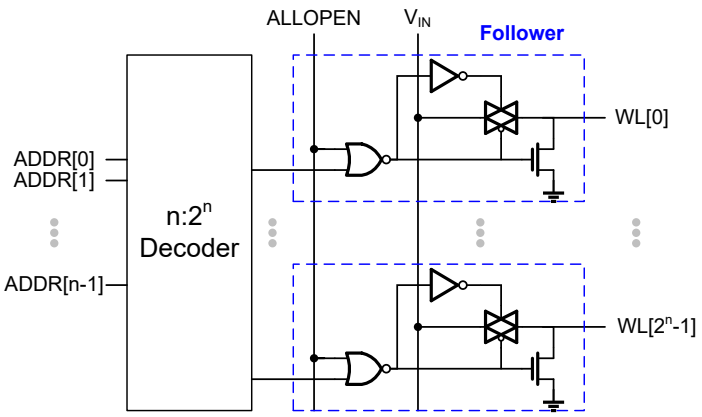


Fig. 13 Circuit diagram of the crossbar WL decoder. Follower circuit is attached to every row of the decoder to enable activation of all WLs when *ALLOPEN*=1.

3) 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 underneath 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**.

4) Analog-to-digital read circuit

To convert these analog weighted sum currents to digital outputs, we use the read circuit in the following reference to employ the principle of the integrate-and-fire neuron model, as shown in Fig. 14(a). The read circuit integrates the weighted sum current on the finite capacitance of the array column. Once the voltage charges up above a certain threshold, the read circuit fires an output pulse and the capacitance is discharged back. The simulated waveform of integrated input voltage and the digital output spikes of the read circuit is shown in Fig. 14(b). The number of output spikes is proportional to the weight sum current. The precision required for this analog-to-digital conversion (ADC) determines the pulse width in each bit of the input vector. In the simulator, a collection of read circuits (**readCircuit**) is instantiated from **ReadCircuit** class in **ReadCircuit.cpp**.

D. Kadetotad, Z. Xu, A. Mohanty, P.-Y. Chen, B. Lin, J. Ye, S. Vrudhula, S. Yu, Y. Cao, J.-S. Seo, "Parallel architecture with resistive crosspoint array for dictionary learning acceleration," *IEEE J. Emerg. Sel. Topics Circuits Syst. (JETCAS)*, vol. 5, no. 2, pp. 194-204, 2015.

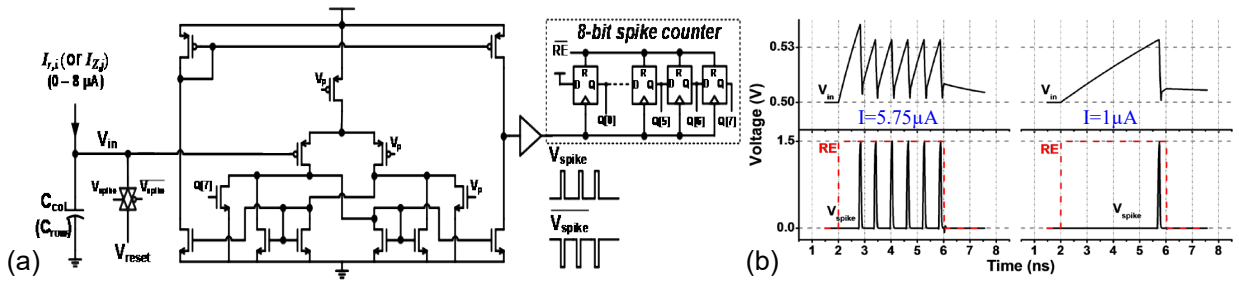


Fig. 14 (a) Design of a read circuit that employs the principle of the integrate-and-fire neuron model. (b) Simulated waveform of integrated input voltage and the digital output spikes of the read circuit.

5) WL decoder and column decoder

Both the traditional WL decoder (**wlDecoder**) and column decoder (**colDecoder**) are instantiated from **RowDecoder** class in **RowDecoder.cpp.** Their only difference is the connection to the array rows or columns, which will be determined in the initialization. If **REGULAR_ROW** is specified, then it will be a WL decoder. If **REGULAR_COL** is specified, then it will be a column decoder.

6) 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**.

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. 12(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**.

---

7. Algorithm Level: Multilayer Perceptron (MLP) Neural Network Architecture

At the algorithm level, we provide a simple 2-layer multilayer perceptron (MLP) neural network for performance benchmark. As shown in Fig. 15(a), the network consists of an input layer, hidden layer and

output layer (the input layer is not included when counting the number of layers). MLP is a fully connected neural network, where each neuron node in one layer connects to every neuron node in the following layer. The connection between two neuron nodes is through a synapse with its strength representing the weight, where W_{IH} and W_{HO} are the weight matrix between input and hidden layer and between hidden and output layer, respectively. For the input image data, we post-processed the MNIST handwritten digits by cropping the edges of each image (making it 20×20 pixels). For the favor of hardware implementation, we also convert the images to black and white data in the simulator to reduce the design complexity of input encoding. By default, the network topology is 400(input layer)-100(hidden layer)-10(output layer). 400 neurons of input layer correspond to the 20×20 MNIST image, and 10 neurons of output layer correspond to 10 classes of digits. The users could change the network topology as needed. However, a new learning rate may be required to optimize the learning accuracy.

The learning applications that the network can implement in this simulator include the online learning and offline training with classification only. In online learning, the simulator emulates hardware to train the network with images randomly picked from the training dataset (60k images) and classify the testing dataset (10k images). In offline training with classification only, the network is pre-trained by software, and the MLP simulator only emulates hardware to classify the testing dataset. The training data file is **patch60000_train.txt** and its label file (the correct answers of the training data) is **label60000_train.txt**. The testing data file is **patch10000_test.txt** and its label file (the correct answers of the testing data) is **label60000_train.txt**.

The training process consists of two key operations, the feed forward (FF) and back propagation (BP). In feed forward, the input data are fed from the input layer and they will travel in a forward direction to the output layer via a series of weighted sum operation and neuron activation function along the way. The feed forward result at the output layer will then be compared with its correct answer (the label) to calculate its prediction error (the deviation). In back propagation, this error is propagated backward from the output layer to adjust the weights of each layer in a way that the prediction error is minimized. In this simulator, we use stochastic gradient decent method to update the weights in the back propagation. Different than the traditional gradient decent, the back propagation is performed after the feed forward of every image rather than that of the entire image dataset. On the other hand, the testing (classification) process only has the feed forward operation to make predictions. The weights in this process will not be changed.

Fig. 15(b) shows a schematic of a neuron node, which encapsulates the principles discussed above. The neuron takes the weighted sum result from its inward synapses and pass it through a 1-bit low-precision activation function. In this way, the offline classification, which is purely feed forward, can be realized in 1 bit. However, the computation on the back propagation of weight update generally needs higher precision to update the small errors, thus a high-precision activation function for the back propagation is still necessary.

Fig. 15(c) shows the circuit block diagram for hardware implementation of this 2-layer MLP neural network. The weighted sum operation is performed using the synaptic cores. However, the weights used in a regular synaptic array can only represent positive values (W_H=0~1), while the weights in algorithm can be either positive or negative values (W_A=-1~1). The algorithm's weighted sum is then expressed as

$$W_A V = 2(W_H - 0.5\,J)V = 2W_H V - JV \quad (4)$$

where V is the input vector and J is the matrix of all ones that has the same dimension as W_A and W_H. In Eq. (4), $W_H V$ is the weighted sum output from the synaptic core. Therefore, we squeeze W_A from (-1~1) to the range of W_H (0~1): i.e. -1 is mapped to 0, 0 is mapped to 0.5, and 1 is mapped to 1. For each cell, the 0.5

reference weight is represented by $(I_{on}+I_{off})/2$. To reconstruct $W_A V$, we have to perform a two-step read from the array: first, we read out $W_H V$, and then subtract $0.5JV$ (basically the sum of the reference weight) from $W_H V$ through the adder at the periphery. And then multiply W_H -0.5 J by 2 using a 1-bit left-shift. The MSB (sign bit in 2's complement notation) of the adder output will be the 1-bit output of the low-precision activation function. It should be noted that we only consider the main sub-circuit modules for the neuron periphery at current stage of this simulator, and the hardware for BP error calculation as well as the detailed control logics will be included in the future release.

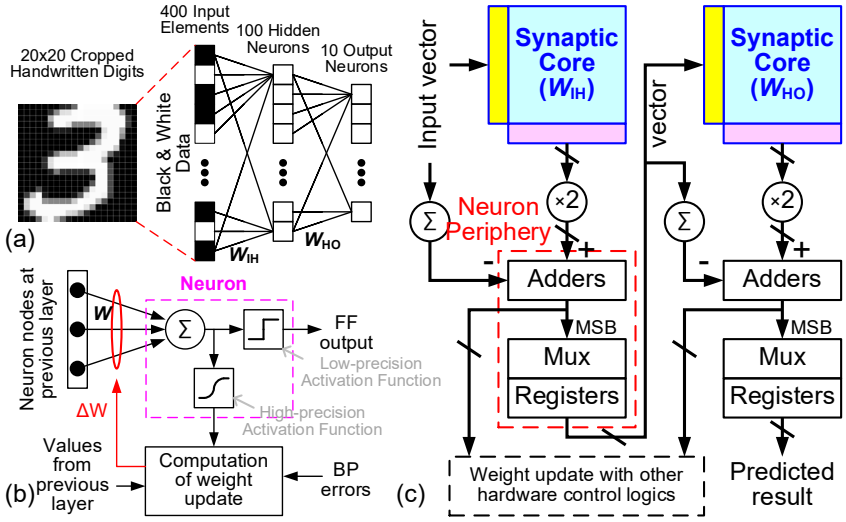


Fig. 15 (a) The 2-layer MLP neural network. (b) Schematic of a neuron node. (c) Circuit block diagram for hardware implementation of the 2-layer MLP network.

In the back propagation phase, the weight update values (ΔW) will be translated to the number of LTP or LTD write pulses (Fig. 15(b)) and applied to the synaptic array following the voltage bias scheme in Fig. 7, Fig. 9, Fig. 10 or Fig. 11. In the previous 1.0 version, we use a naïve weight update scheme, where all the selected cells in each write batch have to wait for the full number of write pulse cycles regardless of their ΔW. This naïve weight update scheme effectively reduces the hardware design complexity, but also greatly increases the weight update latency and energy due to the redundant write pulse cycles. Thus, in 2.0 version, we propose to use an optimized weight update scheme, where the cells only need to go through the maximum ΔW's number of cycles in each write batch. If all the cells in a write batch do not need an update (ΔW=0), this entire write batch can even be skipped. This could bring significant reduction in weight update latency and energy, especially considering that ΔW will usually become very small or even zero after the first few epochs of learning.

8. How to run MLP simulator (+NeuroSim)

1) Select the synaptic device type in main.cpp

Firstly, the users have to select the synaptic device type for the two synaptic cores. Available device types are **RealDevice**, **IdealDevice**, **MeasuredDevice**, **DigitalNVM** and **SRAM** as listed in Section 4.3. The default configuration is **RealDevice** for both synaptic cores, as shown below in **main.cpp**:

```
arrayIH->Initialization<RealDevice>();
arrayHO->Initialization<RealDevice>();
```

2) Modify the device parameters in Cell.cpp

After selecting the synaptic device type, the users may wish to modify the device parameters in the corresponding synaptic device class in **Cell.cpp**. Essential parameters have been described in Section 4.3.

3) Modify the network and hardware parameters in Param.cpp

The users may also wish to modify the network and hardware parameters in **Param.cpp**. For the network side, **numMnistTrainImages** and **numMnistTestImages** are the number of images in MNIST during training and testing respectively, **numTrainImagesPerEpoch** means the number of training images per epoch, while **interNumEpochs** represents the internal number of epochs within each printed epoch shown on the screen. In addition, **nInput**, **nHide** and **nOutput** are the number of neurons in input, hidden and output layers in the 2-layer MLP neural network, respectively.

For the hardware side, the first four hardware parameters determine the learning configuration, which can be the following cases:

1. Online learning in hardware: **useHardwareInTrainingFF, useHardwareInTrainingWU** and **useHardwareInTestingFF** are all **true**
2. Offline learning in software and then classification only in hardware: **useHardwareInTrainingFF** and **useHardwareInTrainingWU** are **false**, while **useHardwareInTestingFF** is **true**
3. Pure learning in software: **useHardwareInTrainingFF, useHardwareInTrainingWU** and **useHardwareInTestingFF** are all **false**

For other hardware parameters, **numBitInput** means the number of bits of the input data. The hardware architecture design in this released version only allows **numBitInput=1** (black and white data), which should not be changed. **numBitPartialSum** represents the number of bits in the digital output (partial weighted sum output) of read circuit (ADC) at the periphery of the synaptic array. **numWeightBit** means the number of weight bits for pure algorithm without consideration of hardware, and **numColMuxed** means the number of columns of the synaptic array sharing one read circuit in the array. Time-multiplexing is required if **numColMuxed** is greater than 1. For example, the total weighted sum latency will be increased by roughly 16 times if **numColMuxed=16**. In the weight update, there might also be limited throughput for the weight update information to be provided from outside. In this case, time-multiplexing is implemented by setting **numWriteColMuxed**. For example, **numWriteColMuxed=16** means updating every row will need roughly 16 weight update operations.

4) Compilation of the program

Whenever any change is made in the files, the codes has to be recompiled by using `make` command as stated in **Installation and Usage (Linux)** section. If the compilation is successful, the following screenshot of Fig. 16 can be expected:

```
g++ -c -fopenmp -O3 -std=c++0x -w formula.cpp -o formula.o
g++ -c -fopenmp -O3 -std=c++0x -w Array.cpp -o Array.o
g++ -c -fopenmp -O3 -std=c++0x -w NeuroSim.cpp -o NeuroSim.o
g++ -c -fopenmp -O3 -std=c++0x -w Train.cpp -o Train.o
g++ -c -fopenmp -O3 -std=c++0x -w Param.cpp -o Param.o
g++ -c -fopenmp -O3 -std=c++0x -w IO.cpp -o IO.o
g++ -c -fopenmp -O3 -std=c++0x -w Mapping.cpp -o Mapping.o
g++ -c -fopenmp -O3 -std=c++0x -w Cell.cpp -o Cell.o
g++ -c -fopenmp -O3 -std=c++0x -w Test.cpp -o Test.o
g++ -c -fopenmp -O3 -std=c++0x -w NeuroSim/formula.cpp -o NeuroSim/formula.o
g++ -c -fopenmp -O3 -std=c++0x -w NeuroSim/FunctionUnit.cpp -o NeuroSim/FunctionUnit.o
g++ -c -fopenmp -O3 -std=c++0x -w NeuroSim/SwitchMatrix.cpp -o NeuroSim/SwitchMatrix.o
g++ -c -fopenmp -O3 -std=c++0x -w NeuroSim/SubArray.cpp -o NeuroSim/SubArray.o
g++ -c -fopenmp -O3 -std=c++0x -w NeuroSim/Technology.cpp -o NeuroSim/Technology.o
g++ -c -fopenmp -O3 -std=c++0x -w NeuroSim/Mux.cpp -o NeuroSim/Mux.o
g++ -c -fopenmp -O3 -std=c++0x -w NeuroSim/VoltageSenseAmp.cpp -o NeuroSim/VoltageSenseAmp.o
g++ -c -fopenmp -O3 -std=c++0x -w NeuroSim/DFF.cpp -o NeuroSim/DFF.o
g++ -c -fopenmp -O3 -std=c++0x -w NeuroSim/Precharger.cpp -o NeuroSim/Precharger.o
g++ -c -fopenmp -O3 -std=c++0x -w NeuroSim/DecoderDriver.cpp -o NeuroSim/DecoderDriver.o
g++ -c -fopenmp -O3 -std=c++0x -w NeuroSim/RowDecoder.cpp -o NeuroSim/RowDecoder.o
g++ -c -fopenmp -O3 -std=c++0x -w NeuroSim/SenseAmp.cpp -o NeuroSim/SenseAmp.o
g++ -c -fopenmp -O3 -std=c++0x -w NeuroSim/Adder.cpp -o NeuroSim/Adder.o
g++ -c -fopenmp -O3 -std=c++0x -w NeuroSim/WLDecoderOutput.cpp -o NeuroSim/WLDecoderOutput.o
g++ -c -fopenmp -O3 -std=c++0x -w NeuroSim/ReadCircuit.cpp -o NeuroSim/ReadCircuit.o
g++ -c -fopenmp -O3 -std=c++0x -w NeuroSim/ShiftAdd.cpp -o NeuroSim/ShiftAdd.o
g++ -c -fopenmp -O3 -std=c++0x -w NeuroSim/SRAMWriteDriver.cpp -o NeuroSim/SRAMWriteDriver.o
g++ -c -fopenmp -O3 -std=c++0x -w main.cpp -o main.o
g++ -fopenmp -O3 -std=c++0x -w formula.o Array.o NeuroSim.o Train.o Param.o IO.o Mapping.o Cell.o Test.o NeuroSim/formula.o NeuroSim/FunctionUnit.o NeuroSim/
SwitchMatrix.o NeuroSim/SubArray.o NeuroSim/Technology.o NeuroSim/Mux.o NeuroSim/VoltageSenseAmp.o NeuroSim/DFF.o NeuroSim/Precharger.o NeuroSim/DecoderDrive
r.o NeuroSim/RowDecoder.o NeuroSim/SenseAmp.o NeuroSim/Adder.o NeuroSim/WLDecoderOutput.o NeuroSim/ReadCircuit.o NeuroSim/ShiftAdd.o NeuroSim/SRAMWriteDriver
.o main.o -o main
```

Fig. 16 Output of `make`

5) Run the program

Then, use `./main` or `make run` to run the program. For both commands, the program will print out the results at every epoch during the simulation. Compared to `./main`, `make run` can save the results to a log file with filename appended with the current time info, as shown in Fig. 17. With the default value of **totalNumEpochs=125** and **numTrainImagesPerEpoch=8000** for a total 1 million MNIST images, the simulation will approximately take about 1 hour with an Intel i7 CPU and 32 GB RAM.

```
Total SubArray (synaptic core) area=5.0841e-09 m^2
Total Neuron (neuron peripheries) area=1.2082e-09 m^2
Total area=6.2923e-09 m^2
Leakage power of subArrayIH is : 7.1391e-05 W
Leakage power of subArrayHO is : 1.6238e-05 W
Leakage power of NeuronIH is : 1.5533e-05 W
Leakage power of NeuronHO is : 2.4850e-06 W
Total leakage power of subArray is : 8.7629e-05 W
Total leakage power of Neuron is : 1.8018e-05 W
```

```
Accuracy at 125 epochs is : 71.85%
        Read latency=7.2030e-02 s
        Write latency=3.1997e+04 s
        Read energy=4.1946e-04 J
        Write energy=1.3019e-02 J
```

Fig. 17 Output of the simulation

At the end of simulation, it is expected to have **similar** results for the Ag:a-Si example in Table 2:

Accuracy	Area (m^2)	Read Latency (s)	Write Latency (s)	Read Energy (J)	Write Energy (J)
71.85%	6.2923^{-9}	$7.2030e^{-2}$	$3.1997e^{4}$	$4.1964e^{-4}$	$1.3019e^{-2}$

Table 2 Final results of learning accuracy and circuit-level performance

For the accuracy of pure software learning as the reference, the users could change the learning modes in **Param.cpp** as discussed above, and it is 96~97% for a network topology 400-100-10.

9. The Benchmark Table for Different Synaptic Devices

This section shows the benchmark table for different synaptic device. In V3.0, the technology node is shifted to 32nm from 14nm for all devices, which is more realistic considering the current industry integration of eNVMs at mostly at 40nm or 28nm. Besides, In V3.0, more devices are added into the benchmark table. In analog eNVM devices, the EpiRAM proposed by the MIT team and the TaOx/HfO2 stack proposed by the Tsinghua team are included. The MUX transistor sizes are relaxed for those devices with small on-state resistance (Ron) to allow voltage transfer to the synaptic arrays, therefore, the area of those arrays significantly increases. For digital synapse, STT-MRAM is added as a comparison to SRAM. Both row-by-row read-out and parallel read-out are simulated for STT-MRAM device.

Table I. The benchmark table in NeuroSimV2.0 (at 14nm node)

	Analog eNVM synapses						Analog FeFET	Digital synapse
Device type	**Ag:a-Si**	**TaO_x/TiO_2**	**PCMO**	**AlO_x/HfO_2**	**GST PCM**	**EpiRAM**	**HZO FeFET**	**6-bit SRAM**
# of conductance states	97	102	50	40	100-120	64	32	--
Nonlinearity (weight increase/decrease)	2.4/-4.88	1.85/-1.79	3.68/-6.76	1.94/-0.61	0.105/2.4	0.5/-0.5	1.75/1.46	--
R_{ON}	26 MΩ	5 MΩ	23 MΩ	16.9 kΩ	4.71 kΩ	81 kΩ	559.28 kΩ	--
ON/OFF ratio	12.5	2	6.84	4.43	19.8	50.2	45	--
Weight increase pulse	3.2V/300µs	3V/40ms	-2V/1ms	0.9V/100µs	0.7V (avg.)/ 6µs	5V/5µs	3.65V (avg.)/ 75ns	--
Weight decrease pulse	-2.8V/300µs	-3V/10ms	2V/1ms	-1V/100µs	3V (avg.)/ 125ns	-3V/5µs	-2.95V (avg.)/ 75ns	--
Cycle-to-cycle variation (σ)	3.5%	<1%	<1%	5%	1.5%	2%	<1%	--
Online learning accuracy	~73%	~10%	10%	~41%	~87%	~91.5%	~90%	~94%
Area	1072.0 μm^2	1071.3 μm^2	1071.3 μm^2	3657.2 μm^2	7233.0 μm^2	1546.3 μm^2	1190.4 μm^2	10311 μm^2
Latency (naïve)	4.20E8 s	3.57E10 s	7.00E8 s	5.60E7 s	4.39E6 s	4.48E6 s	3.36E4 s	7.76 s
Energy (naïve)	87.94 mJ	65.86 mJ	29.4 mJ	150 mJ	1.52 J	183.7mJ	98.01 mJ	6.98 mJ
Latency (optimized)	64200 s	2.845 s	5.2507 s	443.98 s	413.0 s	499.8s	1.2924 s	3.7049 s
Energy (optimized)	14.81 mJ	0.17 mJ	0.17 mJ	146.19 mJ	1.34 J	67.2mJ	0.21 mJ	3.3 mJ
Leakage power	35.29 µW	35.29 µW	35.29 µW	35.29 µW	35.29 µW	35.29 µW	35.29 µW	1.1 mW

Ref. P.-Y. Chen, S. Yu, "Technological benchmark of analog synaptic devices for neuro-inspired architectures," IEEE Design & Test, **DOI:** 10.1109/MDAT.2018.2890229

Table II. The Benchmark Table in NeuroSimV3.0 (at 32nm node)

	Analog eNVM synapses						Analog FeFET	Digital synapse	
Device type	**Ag:a-Si [1]**	**TaOx/HfOx [2]**	**PCMO [3]**	**$AlOx/HfO_2$ [4]**	**GST PCM [5]**	**EpiRAM (Ag:SiGe) [6]**	**HZO FeFET [7]**	**6-bit SRAM**	**6-bit STT-MRAM**
# of conductance states	97	128	50	40	100-120	64	32	--	--
Nonlinearity (weight increase/decrease)	2.4/-4.88	0.04/-0.63	3.68/-6.76	1.94/-0.61	0.105/2.4	0.5/-0.5	1.75/1.46	--	--
R_{ON}	26 MΩ	100 KΩ	23 MΩ	16.9 KΩ	4.71 KΩ	81 KΩ	559.28 KΩ	--	3.5 KΩ

ON/OFF ratio	12.5	10	6.84	4.43	19.8	50.2	45	--	2.3	
Weight increase pulse	3.2V/300µs	1.6V/50ns	-2V/1ms	0.9V/100µs	0.7V (avg.)/ 6µs	5V/5µs	3.65V (avg.)/ 75ns	--	1V/10ns	
Weight decrease pulse	-2.8V/300µs	1.6V/50ns	2V/1ms	-1V/100µs	3V (avg.)/ 125ns	-3V/5µs	-2.95V (avg.)/ 75ns	--	1V/10ns	
Cycle-to-cycle variation (σ)	3.5%	3.7%	<1%	5%	1.5%	2%	<0.5%	--	--	
Online learning accuracy	~72%	~80%	~33%	~20%	89%	92%	88%	~94%	~94%	~94%
Area	6292.3 μm^2	8663.1 μm^2	6292.4 μm^2	21760 μm^2	46565μm^2	9144.3μm^2	7032.6μm^2	65728 μm^2	70254 μm^2	66632 μm^2
Latency (optimized)	31997s	10.15s	12218s	470.42S	203.0s	229.6s	2.73s	5.98 s	6.9s (parallel)	90.1s (row-by row)
Energy (optimized)	13.44mJ	4.01mJ	2.53mJ	15.26mJ	35.0mJ	31.01mJ	1.9mJ	15.56 mJ	0.1467J	0.1462J
Leakage power	105.65µW	105.65µW	105.65µW	105.65µW	105.65µW	105.65µW	105.65µW	2.80 mW	124.8 µW	84.0 µW

Some remarks

1. The area cost increases by about 6 times at 32nm node, which is slightly larger than $(32nm/14nm)^2 = 5.22$. It can be explained by additional periphery modules to support the reference weight.
2. In terms of latency and energy consumption, they are determined by both the operations of periphery circuit and the read/write within the synaptic array. The latency and energy consumption of the periphery circuit will be degraded by shifting technology from 14nm to 32nm. The latency and energy consumption within the synaptic array is mainly determined by the write pulse width and write voltage, which is assumed to be independent on technology node here. Therefore, three trends are observed for analog eNVMs based synaptic device:
    a. For devices with relatively OK linearity and long programming pulse width (e.g. EpiRAM, Ag-Si, PCM with 1µs-100µs programming pulse width), both latency and energy consumption is reduced because the weight range is changed from 0-1 to (-1)-1. In this case, the same ΔW corresponds to 1/2 the number of programming pulses in V2.0 and therefore, the write latency in the synaptic array is reduced significantly. The increase of latency and energy consumption of the periphery is comparatively small. Therefore, both latency and energy consumption is reduced.
    b. For devices with relatively OK linearity and short programming pulse width (e.g. FeFET with ~75ns programming pulse width), both energy consumption and latency is increased due to the impact of technology node shift on periphery circuit.
    c. For devices with relatively bad linearity (e.g. PCMO), both latency and energy consumption is increased. In V2.0, devices with bad linearity will not learn (~10% learning accuracy) and therefore, the number of programming pulse applied is relatively less. In V3.0, the learning accuracy is slightly improved, which means more programming pulses are applied to change the weight of the synapse device.
3. For online learning accuracy, good accuracy can be obtained if the following three conditions are obtained:
    a. Good linearity. In general, devices with good linearity shows good online learning accuracy. For example, for EpiRAM and PCMs, it maintains the accuracy in both V2.0 and V3.0
    b. Small cycle to cycle variation. For TaO_x/HfO_x even though it has good linearity, the online accuracy is low because of it relatively large cycle to cycle variation.

c. Enough conductance states. This is not obvious in V2.0. But in V3.0, since the weight range is extended from (0,1) to (-1,1), the weight distance between each conductance becomes larger. For device with good linearity, small variation but relatively fewer number of conductance state (e.g. HZO FeFET), an accuracy drop is observed in V3.0. For example, the online learning accuracy for HZO FeFET is dropped from 90% to 88% in V3.0, the online learning accuracy. For AlO_x/HfO_X is dropped from 41% to 20% due to its limited number of conductance states.

4. For devices with low on-off ratio, the online training accuracy can be improved by introducing the reference current in V3.0.
5. For digital eNVM based synapses, by parallelizing the read-out operation, STT-MRAM can achieve comparable latency as SRAM and much lower leakage power than SRAM. But the energy consumption for STT-MRAM is still larger due to its relatively large write current.

References for device parameters in V3.0:

[1]. S. H. Jo, et al. "Nanoscale memristor device as synapse in neuromorphic systems." Nano Letters, 10.4 (2010): 1297-1301.

[2]. W. Wu, et al. "A methodology to improve linearity of analog RRAM for neuromorphic computing." IEEE Symposium on VLSI Technology, 2018.

[3]. S. Park, et al. "Neuromorphic speech systems using advanced ReRAM-based synapse." IEEE International Electron Devices Meeting, 2013.

[4]. J. Woo, et al. "Improved synaptic behavior under identical pulses using AlOx/HfO2 bilayer RRAM array for neuromorphic systems." IEEE Electron Device Letters, 37.8 (2016): 994-997.

[5]. D. Kuzum, et al. "Nanoelectronic programmable synapses based on phase change materials for brain-inspired computing," Nano Letters, 12.5 (2011): 2179-2186.

[6]. S. Choi, et al. "SiGe epitaxial memory for neuromorphic computing with reproducible high performance based on engineered dislocations." Nature Materials, 17.4 (2018): 335-340.

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