User Manual of 3D+NeuroSim Framework V1.0

Developers: Xiaochen Peng, Shanshi Huang, and Anni Lu

PI: Prof. Shimeng Yu, Georgia Institute of Technology

June 1, 2021

Index

1. Introduction 1
2. New Feature Highlights in 3D+NeuroSim V1.0 2
3. System Requirements (Linux) 4
4. Installation and Usage (Linux) 4
5. Chip Level Architectures 6
   5.1 Interconnect: H-Tree 6
   5.2 Floorplan of Neural Networks 7
   5.3 Weight Mapping Methods 9
   5.4 Pipeline System 10
6. Circuit Level: Synaptic Array Architectures 11
   6.1 Parallel Synaptic Array Architectures 11
   6.2 Array Peripheral Circuits 14
7. Algorithm Level: PyTorch Wrapper 18
8. Algorithm Level: Inference Accuracy Estimation 19
9. How to run *DNN +NeuroSim* 22
10. Reference 27

---

1. Introduction

DNN+NeuroSim is an integrated framework, which is developed in C++ and wrapped by Pytorch, to emulate the deep neural networks (DNN) inference performance (in V1.0-V1.3) or on-chip training (in V2.0-V2.2) performance on the hardware accelerator based on near-memory computing or in-memory computing architectures. This released 3D+NeuroSim is extended from DNN+NeuroSim to support electrical-thermal co-simulation of 3D integrated hardware accelerators. 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 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 accuracy as well as the circuit-level performance metrics at the run-time of inference.

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++ (consumes long run-time), this *DNN+NeuroSim* framework is an integrated simulator with Pytorch 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 guarantee efficient inference running 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 this released version, three networks (VGG-8 network for CIFAR-10 dataset, DenseNet-40 network for CIFAR-10 dataset, ResNet-18 network for ImageNet dataset) are provided as default models in the wrapper, with 8-bit synaptic weights and neural activations, while users could modify the precisions and neural network topologies. The hardware parameters (such as technology nodes, memory cell properties, operation modes, and so on) will be defined under *NeuroSim* in **Param.cpp**.

---

2. New Feature Highlights in 3D+NeuroSim V1.0.

Key features in this released are summarized as follows.

1) **Enable 3D integrated electrical-thermal co-simulation**
   In this version, we introduce a group of parameters to define various hardware design options in 3D integration. To enable monolithic 3D integration, we support automatic floor-plan to partition hardware component into top and bottom tiers; we introduce device parameters of top-tier transistors in technology library; we integrate a thermal fitting function (in Pytorch wrapper) to run electrical-thermal co-simulation. To enable heterogeneous 3D integration, we introduce vertical path of signal delivery with through-silicon via (TSV) and driver/receiver; we specify the floorplan where memory arrays are in memory tiers, and other logics are in logic tier, there is only one logic tier on the bottom, while user can define multiple memory tiers on top of the logic tier (as a memory cube). For heterogeneous 3D integration, we introduce a group of technology parameters, where users can define different technology nodes for memory tier and logic tier; meanwhile an integrated thermal fitting function is also introduced to enable electrical-thermal co-simulation.
   a) ***Monolithic 3D Integration***
      In [3], we present a work to benchmark monolithic 3D integrated CIM accelerators. To address the challenges of ADC overhead and scaling limitation caused by high write voltage in emerging non-volatile memory (eNVM), we propose partitioning the circuit modules in hybrid technology nodes on top and bottom tiers with massive inter-tier vias.

An example of the floorplan of monolithic 3D integrated CIM accelerators is shown in Fig.18, where memory arrays are placed on top tier, and logic circuits are placed on bottom tier. In this framework, we consider the parameter degradation ratio between the top tier and the bottom substrate of the silicon transistors with laser-recrystallization using the experimental data for ultra-thin-body (UTB)-FET [4].

b) Heterogeneous 3D Integration

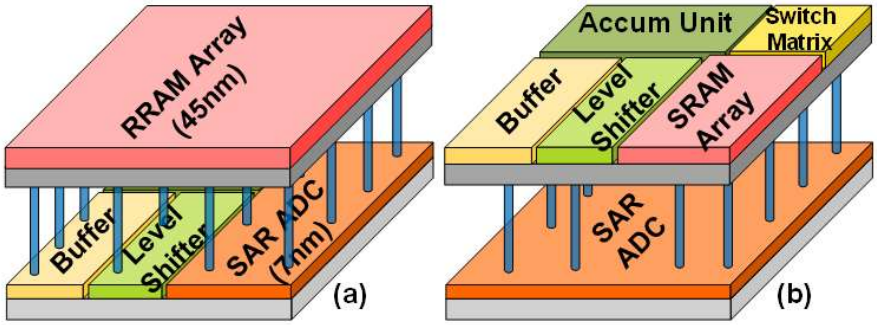

Fig. 1. Floorplan of optimized M3D CIM accelerators.

For heterogeneous 3D integration, we assume a multi-tier structure as shown in Fig.2, where the memory arrays are on top tiers, and the other logic circuits are on bottom tier. The operation strategy is assumed to be layer-by-layer scheme, such that the memory tiers can share one group of logic circuits, hence decrease the number of vertical path (via TSV) and keep the floorplan less comprehensive. As each time, one specific layer will be activated to access data from logic tier, the data will be sent to corresponding memory arrays (at different tiers) via TSV path, and after the memory arrays finish the analog computation, the output will be sent back to logic tier for further data-processing. In this framework, users are able to define the number of memory tiers, which means the signal could transfer more than one TSV in series. The NeuroSim core can automatically define each neural layer's location (i.e. the framework knows each specific neural layer is placed to which memory tier), such that we can properly estimate the signal transfer latency/energy through the TSV path.

2) Validate with real silicon data (extend from DNN+NeuroSim V1.3)

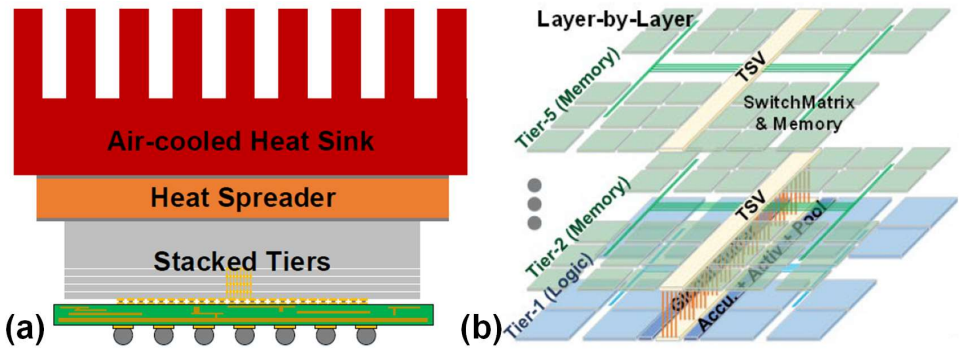

Fig. 2. (a) Die stack model of heterogeneous 3D multi-tier CIM accelerator; (b) floorplan of heterogeneous 3D multi-tier CIM accelerator (layer-by-layer operation).

In this version, we validate and calibrate the hardware performance (area, critical path delay and energy consumption) prediction against a 40nm RRAM-based CIM macro [5] post-layout simulations. Some adjustment factors are introduced to account for transistor sizing and wiring area in the layout, gate switching activity and post-layout performance drop, etc. For details, $\alpha = 1.44$ for the wire areas in level shifter; $\beta = 1.4$ for the sensing cycle as the critical path; $\gamma = 50\%$ and $\delta = 15\%$ separately for dynamic energy of DFFs and adders in shift-add or accumulators; $\epsilon = 5\%$ for dynamic energy of control circuits; and $\zeta = 1.23$ for post-layout energy increase. After these calibrations, the macro-level simulation from NeuroSim is quite accurate with error under 1%. After considering these realistic factors, the predicted performance would decrease to some degree compared to previous versions. Users could enable/disable this option or change these factor values in **Param.cpp**.

3) **Add synchronous and asynchronous mode (extend from DNN+NeuroSim V1.3)**
   In previous versions, the latency of the whole chip is accumulated by the critical path delay of each module, which is clockless and asynchronous. Considering practical circuit design, we added the synchronous mode, where the latency is measured by clock cycles. The clock period is decided by the compute sensing cycle, which is the critical path from giving input to the memory array till the ADC generating the digital partial sum, as this is an analog process and no digital buffer could be added in between. The latency of other digital modules is measured as how many cycles are needed for the processing because their timing could be adjusted by adding digital buffer. The predicted performance especially the throughput of synchronous mode is lower than the asynchronous mode. Users could change this option in **Param.cpp**.
4) **Update technology file for FinFET (extend from DNN+NeuroSim V1.3)**
   The default transistor models in NeuroSim were calibrated with the predictive technology model (PTM) [6], which is available to the public and has a wide range of technology nodes from 130nm to 7nm. However, as the PTM model (of 14nm, 10nm and 7nm) was proposed far earlier than the industry adoption of FinFET, their prediction of Fin size actually deviate from the actual values. We corrected the Fin height, width and pitch following the recent trends in leading foundries in the **Technology.cpp** and made some corresponding changes in standard cell height/width and interconnect wire pitch, and switched to the assumption of using maximum electrical width/or fin number in the standard cell for digital circuit design.
5) **Add level shifter for eNVM (extend from DNN+NeuroSim V1.3)**
   Level shifter module is added for eNVM (e.g. RRAM/PCM/FeFET) with high write voltage (>1.5V).

---

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 7.8 (Maipo), gcc v4.8.5, glibc v2.17, (2) Ubuntu 16.04, gcc v5.5.0, 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/3D_NeuroSim.git
```

Step 2: Train the network to get the model for inference

Step 3: Compile the *NeuroSim* Code

```
make
```

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

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 python3.4 + pytorch 1.1.0 (GPU), and CUDA 10.0+cuDNN v7.5.0.

---

5. 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, where the top level of chip is consist of multiple tiles, global buffer, accumulation units, activation units (sigmoid or ReLU), and pooling units. Fig. 3 (b) shows the structure of a tile, which 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, as Fig. 3 (c) shows, a PE is built up by a groups of synaptic sub-arrays, PE buffers, accumulation modules and output buffer. In Fig. 3 (d), it shows an example of synaptic sub-array, which is based on one-transistor-one-resistor (1T1R) architecture for eNVMs. At sub-array level, the array architecture is different for SRAM or FeFET (not shown in this figure).

5.1 Interconnect: H-Tree

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

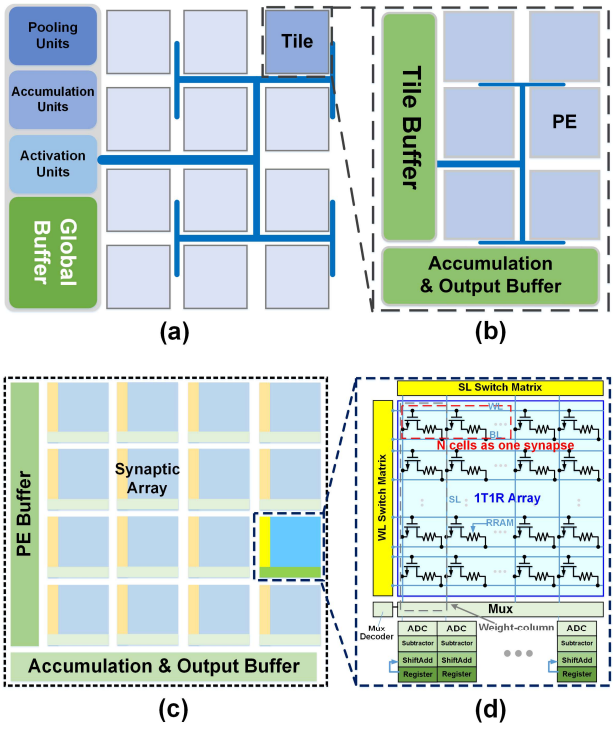

Fig. 3. The diagram of (a) top level of chip architecture, which contains multiple tiles, global buffer, accumulation units, activation units (sigmoid or ReLU) and pooling units; (b) a tile with multiple processing elements (PEs), tile buffer to load in activations, accumulation modules to add up partial sums from PEs and output buffer; (c) a PE contains a group of synaptic arrays, PE buffer and control units, accumulation modules and output buffer; (d) an example of synaptic array based on one-transistor-one-resistor (1T1R) architecture.

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 [7] is introduced as EQ (4.1) shows, where R is the resistance of a minimum-sized repeater, C is the gate

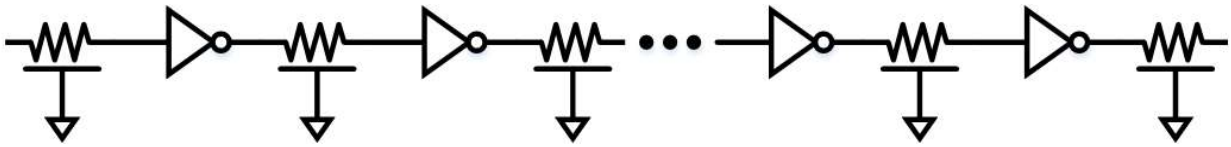

Fig. 4. The diagram of wire with repeaters.

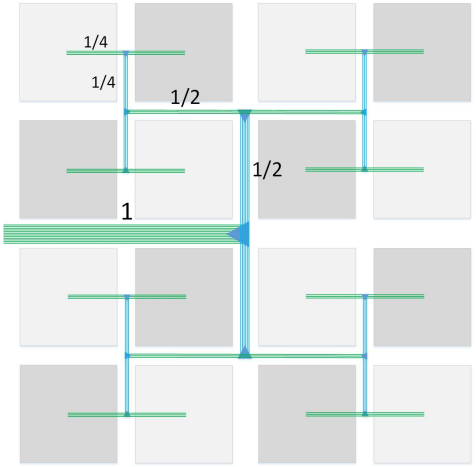

Fig. 5. An example of H-tree for a 4×4 computation-unit array.

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 decrease 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.

5.2 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 process speed.

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, as well as the VGG-8 network 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.

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%

5.3 Weight Mapping Methods

We support two mapping methods in this framework, conventional mapping and novel mapping method which was proposed in [8]. 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.

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

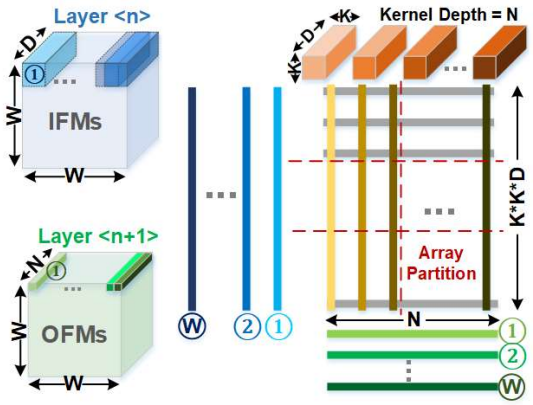

Fig. 6. An example of conventional mapping method of input and weight data.

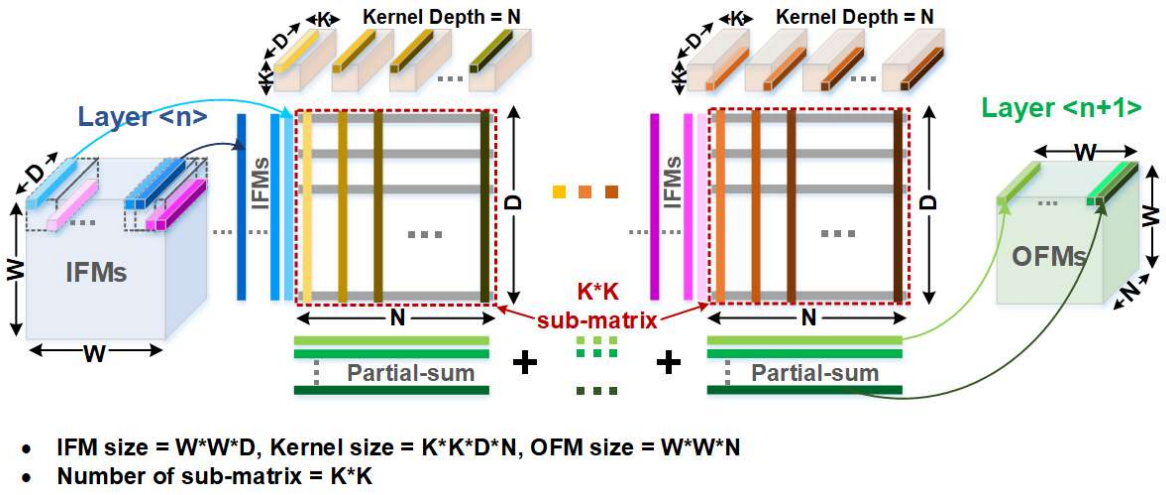

Fig. 7. An example of novel mapping method of input and weight data.

method (as Fig. 5 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.

5.4 Pipeline System

In this framework, we assume all the synaptic weights are mapped on to the inference 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

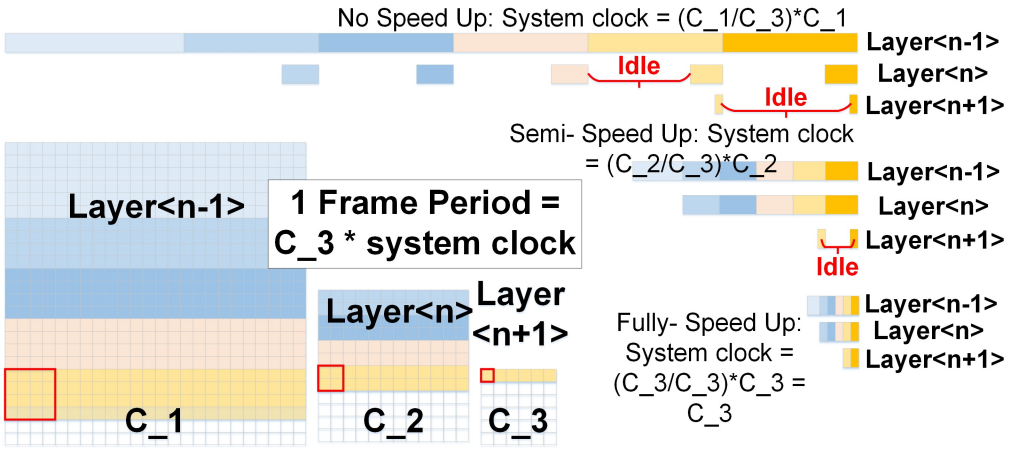

Fig. 8. Speed-up in Pipeline System.

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.

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.

---

6. 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**.

6.1 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 [9]. 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

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 [10], the input vectors will be fetched in via WL switch matrix, the partial-sums will be

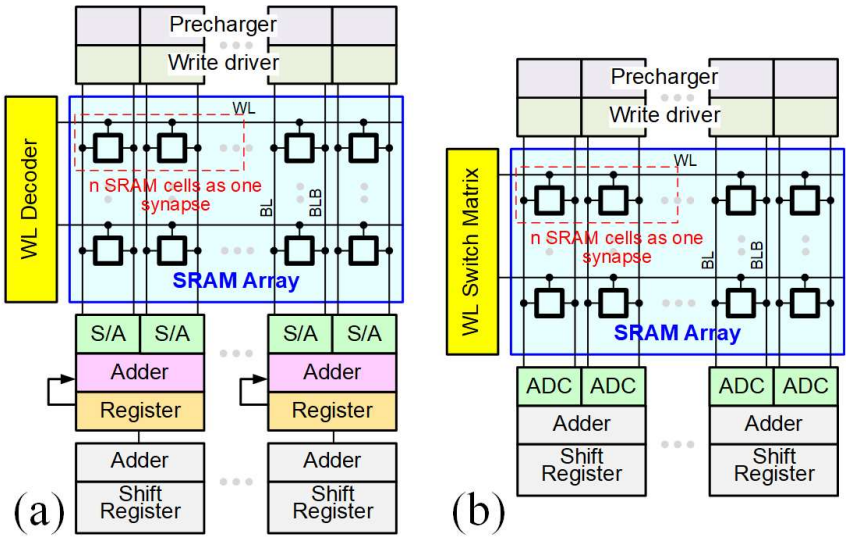

Fig. 9. The diagram of SRAM-based (a) sequential-read-out; (b) parallel-read-out synaptic arrays.

collected along columns simultaneously at one time with high-precision flash-ADCs based on multilevel 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).

2) Analog eNVM 1T1R synaptic array

Fig. 10 (a) and (b) shows the structure of 1T1R based eNVM array. 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

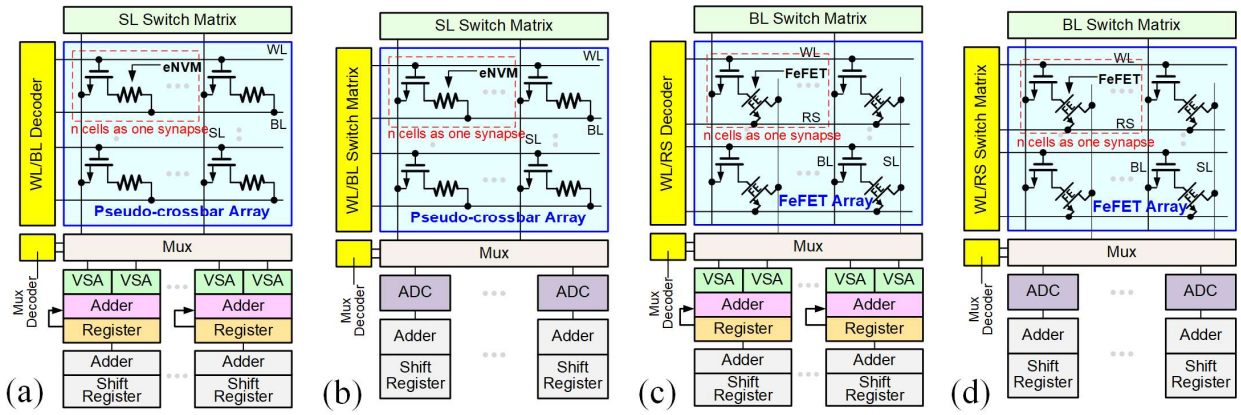

Fig. 10. (a) sequential-read-out and (b) parallel-read-out analog eNVM pseudo-1T1R synaptic arrays; (c) sequential-read-out and (c) parallel-read-out analog FeFET synaptic arrays;

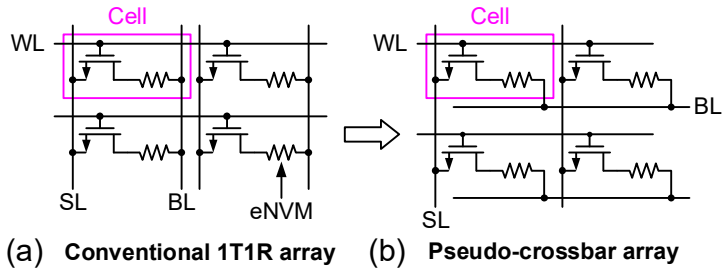

Fig. 11. Transformation from (a) conventional 1T1R array to (b) pseudo-crossbar array by 90° rotation of BL to enable weighted sum operation.

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.

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), as shown in Fig. 10 (c). 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. 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.

4) Analog FeFET array

As shown in Fig. 10 (c) and (d), the analog FeFET array is in the pseudo-crossbar fashion as proposed in [11], 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 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.

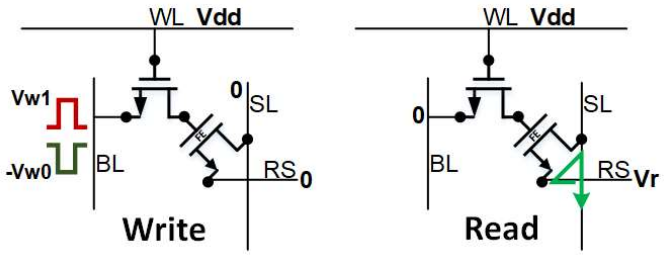

Fig. 12. Operations of (a) write and (b) read in FeFET cell.

6.2 Array Peripheral Circuits

The periphery circuit modules used in the synaptic arrays in Fig.7 and Fig. 8 are described below:

1) Level shifter

Level-shifter is normally required for RRAM (or PCM/FeFET) array to support the need of higher write voltage (e.g. >1.5V which is higher than logic VDD). In the simulator, we take a conventional level shifter as shown in Figure.13. If the validation mode is selected, a wiring area factor **α = 1.44** will be imposed on this module for calibration.

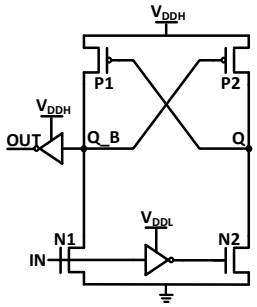

Fig. 13. Schematic of level shifter.

2) Switch matrix

Switch matrices are used for fully parallel voltage input to the array rows or columns. Fig. 14 (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 14 (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

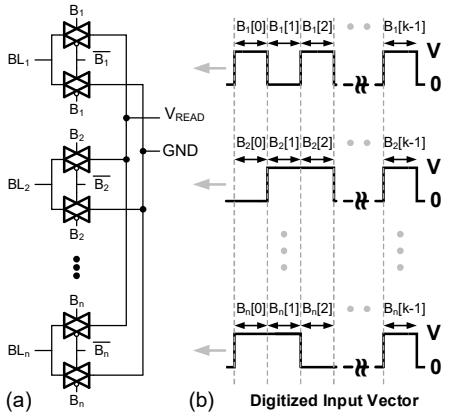

Fig. 14 (a) Transmission gates of the BL switch matrix in the weighted sum operation. A vector of control signals (B1 to Bn) from the registers (not shown here) decide the BLs to be connected to either a voltage source or ground. (b) Control signals in a bit stream to represent the precision of the input vector.

matrices (**slSwitchMatrix**, **blSwitchMatrix** and **wlSwitchMatrix**) are instantiated from **SwitchMatrix** class in **SwitchMatrix.cpp**, this module is used in parallel-read-out synaptic arrays.

3) 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. 15. 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.

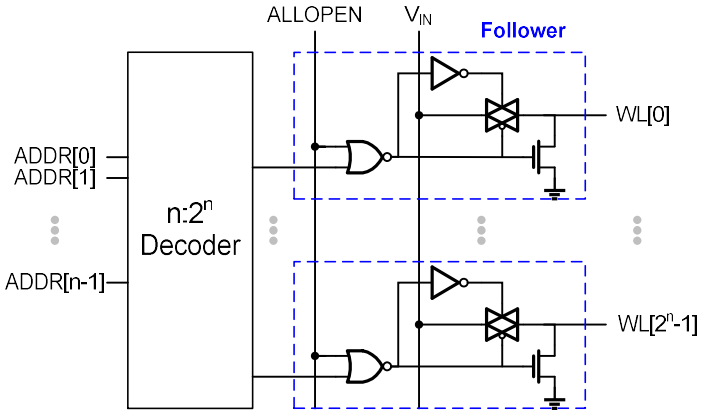

Fig. 15 Circuit diagram of the crossbar WL decoder. Follower circuit is attached to every row of the decoder to enable activation of all WLs when ALLOPEN=1.

4) 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.

5) 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. 16 shows the circuit diagram of new decoder driver and switch matrix for eNVM pseudo-1T1R synaptic array, which could be used to control both WL and BL (or RS) at the same time. In Fig. 16 (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. 16 (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

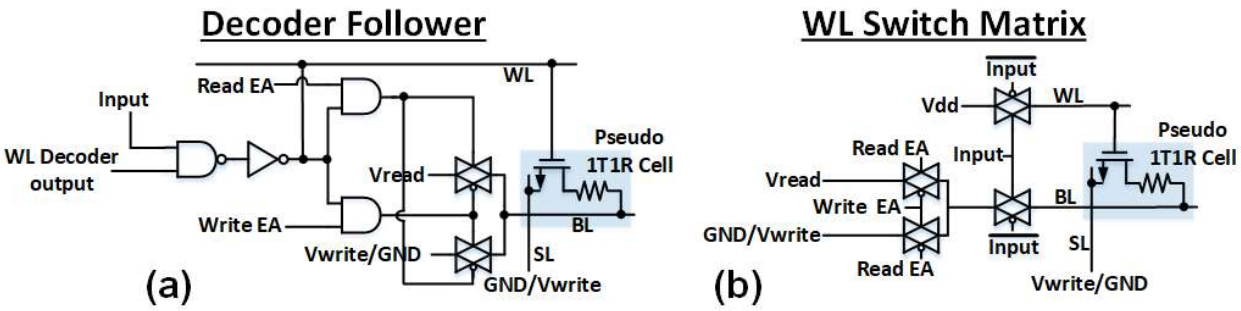

Fig. 16. Circuit diagram of (a) decoder follower and (b) WL switch matrix, which are used to control both WLs and BLs simultaneously, for pseudo-1T1R synaptic arrays.

BL. In 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-1T1R and FeFET synaptic arrays.

6) 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**.

7) Analog-to-digital converter (ADC)

To read out the partial-sums and further process them in the subsquent logic modules (such as activation and pooling), ADCs are used at the end of SLs to generate digital outputs. In the simulator, different types of ADC are supported such as Flash ADC using multilevel voltage-mode sense amplifiers (VSA) or current-mode sense amplifier (CSA), and successive-approximation-register (SAR) ADC as shown in Fig. 17. They

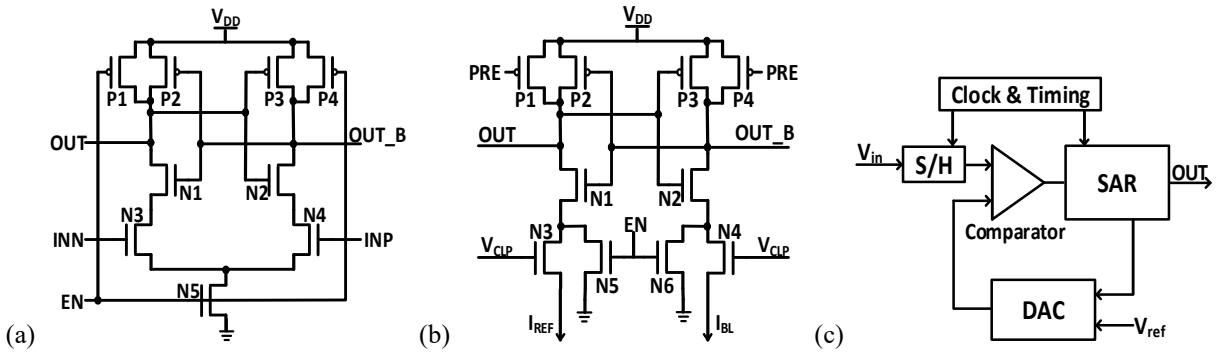

Fig. 17 Schematic of (a) voltage sense amplifier (VSA); (b) current sense amplifier (CSA); (c) successive-approximation-register (SAR) ADC.

have trade-offs in the area/power and latency. Taking the balance between energy consumption and latency into consideration, flash-ADC has better performance for lower resolution (3-bit or below) while SAR-ADC performs better for higher resolution (4-bit or above) especially when with high R_{ON} (e.g. >100k Ω), but the break-even point can be changed to 8-bit with low R_{ON} (e.g. <5k Ω).

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 Cadence 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.

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 Cadence 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. 18 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 Cadence, the fitting function yields reasonable mismatch with much faster simulation compared with Cadence.

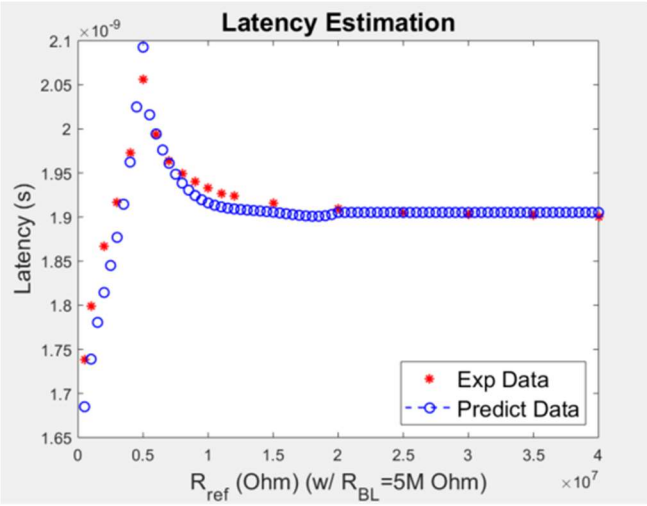

Fig. 18. An example of latency estimation based on fitting functions compared with Cadence results.

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. 19 shows, we perform 8-bit inference of VGG-8 network on CIFAR-10 dataset, to investigate the effects of truncating ADC precision on the classification accuracy. We set the sub-array size to be 128×128, and investigate three schemes with 1-bit cell, 2-bit cell and 4-bit 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. 19 suggest that higher weight-precision generally requires higher ADC-precision. With larger synaptic array size or higher cell precision, higher ADC precision is demanded.

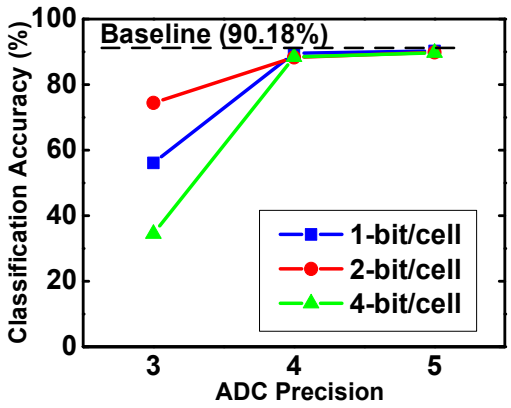

Fig. 19. Classification accuracy of CIFAR-10 for an 8-bit CNN as a function of the ADC precision for partial sums.

8) 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**.

9) 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 14 (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: PyTorch Wrapper

The algorithm we use to get the quantized DNN model for inference is the WAGE from [13]. The Pytorch code is modified based on [14-16]. The same algorithm is realized except that we move the scale term from weight to output to make it more suitable for hardware architecture. 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 to be released in future version.

Here we considered inference with offline training. 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 quantized training 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 maybe useful for online training to be release later). 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 changed 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 offline 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.

---

8. Algorithm Level: Inference Accuracy Estimation

In this framework, the neural network model (or weights) is assumed to be pre-trained off-chip, 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 inference version (V1.0-V1.3) but are consider in the training version (V2.0-V2.2). In 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.

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. 20, 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

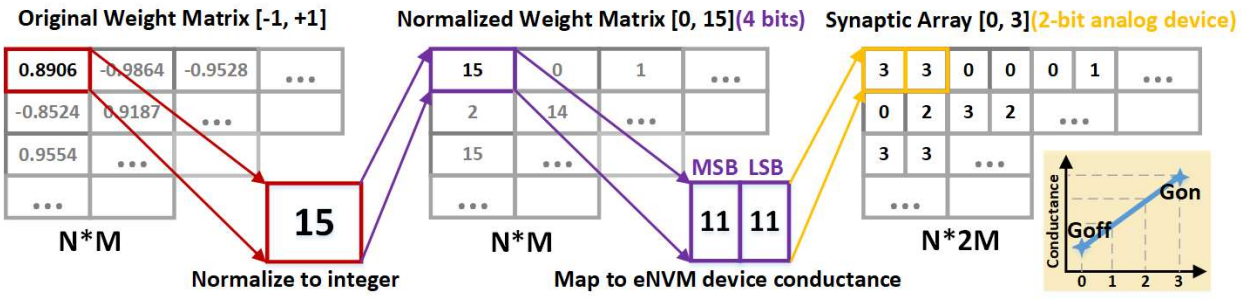

Fig. 22. Mapping weight from algorithm to synaptic device conductance in CIM architecture.

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).

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.

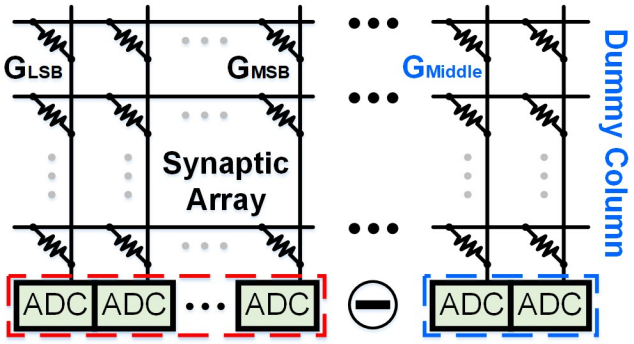

Fig. 23. Introduce dummy column to cancel out the off-state current effects.

2) Conductance Variation

It is well known that the 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 programming of the synaptic arrays.

In this framework, the conductance variation is introduced as a percentage of variation of desired conductance, 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.

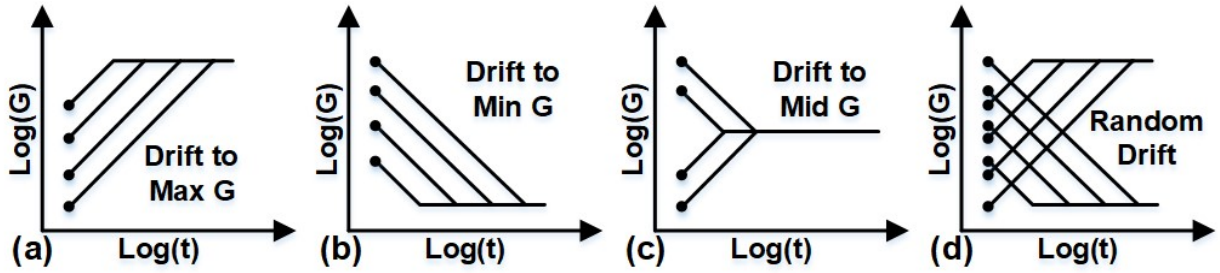

Fig. 24. Different scenarios of conductance drift.

For a chip, the conductance variation could be different from array to array, and device to device, so we module 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 reported 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 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.

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 Effects

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.

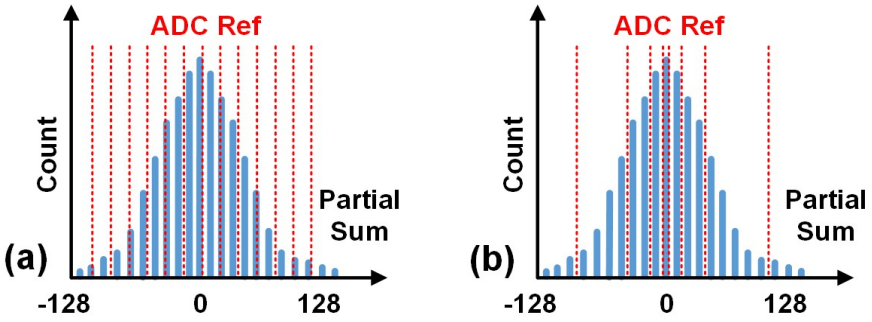

Fig. 25. Linear and non-linear ADC quantization.

However, since it is impractical to use very high-precision ADC at the edge of eNVM sub-arrays, we have to truncate the precision of ADC (for partial sums) to minimize the area and energy overhead.

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 array; 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".

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?
Layer 1	32	32	3	3	3	128	0
Layer 2	32	32	128	3	3	128	1
Layer 3	16	16	128	3	3	256	0
Layer 4	16	16	256	3	3	256	1

Layer 5	8	8	256	3	3	512	0
Layer 6	8	8	512	3	3	512	1
Layer 7	1	1	8192	1	1	1024	0
Layer 8	1	1	1024	1	1	10	0

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 analog, 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. We list some recommended device parameters as below:

Table III Scaling trend of SRAM cell area with technology nodes (assuming F is the same as the technology node)

SRAM	22nm	14nm	10nm	7nm
6T SRAM cell size (F^2)	200	300	400	600
8T SRAM cell size (F^2)	280	480	720	1080

Table IV Recommended eNVM device parameters.

	22nm RRAM (Intel) [17]	90nm RRAM (Winbond) [18]	130nm RRAM (Tsinghua) [19]	90nm PCM (IBM) [20]	22nm FeFET (GF) [21]	22nm STT-MRAM (Intel) [22]
Ron (Ω)	6k	6k	100k	40k	240k	1.4k
On/off ratio	17	150	10	12.5	100	2.8
Cell size (F^2)	5×12	6×6	4×4	4×4	4×6	10×10

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*

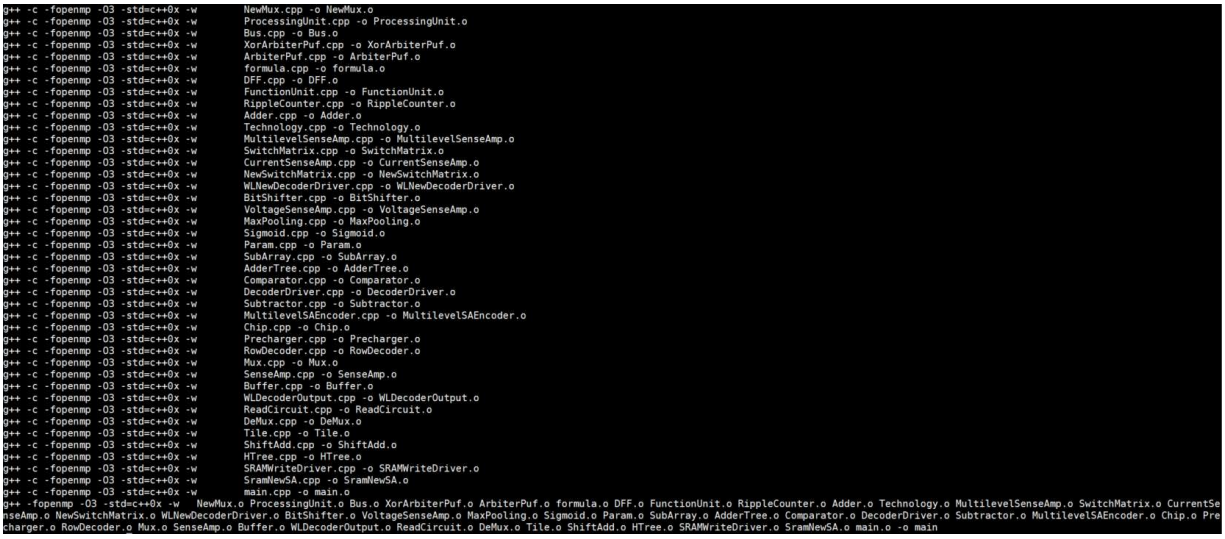

Fig. 26 Output of compilation.

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. 26 can be expected.

4) Run the program with PyTorch wrapper

After compilation of *NeuroSim*, go back to the PyTorch wrapper, in the wrapper, there are three networks (VGG-8 network for CIFAR-10 dataset, DenseNet-40 network for CIFAR-10 dataset, ResNet-18 network for ImageNet dataset) 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
    - The model will be saved at a hierarchical folders based one the option value.
  - Inference
    - Python inference.py
    - Set model_path to the saved model *.pth file

```
model_path = './log/default/batch_size=200/decreasing_lr=200,250/grad_scale=8/seed=117/type=cifar10/wl_activate=8/wl_error=8/wl_grad=8/wl_weight=2/latest.pth'
# data loader and model
```

Fig. 27 example of load path.

The program will print out the results for each layer of the network during the simulation. The simulation 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. Fig. 28 and Fig. 29 show the examples of final output for monolithic and heterogeneous 3D integration respectively. The output from the simulation include hardware inference accuracy, memory utilization, 3D partition results, and latency/energy/leakage breakdown for 1-image inference, and the equivalent energy efficiency in terms of TOPS/W, and throughput in terms of frames per second (FPS), compute efficiency in terms of TOPS/mm^2, and power density in terms of W/ mm^2.

```
---------------------- Monolithic 3D Partition Results ----------------------

Top Tier: Other Peripheries (e.g. decoder, mux ...), Interconnects and Array
Bottom Tier: ADC, Accumulation Circuits
```

```
Chip buffer readLatency is: 3.30184e+06ns
Chip buffer readDynamicEnergy is: 685650pJ
Chip ic readLatency is: 302300ns
Chip ic readDynamicEnergy is: 6.43191e+06pJ

********************** Breakdown of Latency and Dynamic Energy ************************

---------- ADC (or S/As and precharger for SRAM) readLatency is : 344941ns
---------- Accumulation Circuits (subarray level: adders, shiftAdds; PE/Tile/Global level: accumulation units) readLatency
is : 1.25998e+06ns
---------- Other Peripheries (e.g. decoders, mux, switchmatrix, buffers, IC, pooling and activation units) readLatency is :
 3.61467e+06ns
---------- ADC (or S/As and precharger for SRAM) readDynamicEnergy is : 2.04793e+07pJ
---------- Accumulation Circuits (subarray level: adders, shiftAdds; PE/Tile/Global level: accumulation units) readDynamicE
nergy is : 8.82402e+06pJ
---------- Other Peripheries (e.g. decoders, mux, switchmatrix, buffers, IC, pooling and activation units) readDynamicEnerg
y is : 1.52214e+07pJ

********************** Breakdown of Latency and Dynamic Energy ************************


---------------------------- Performance ------------------------------
Chip Operation Temperature (K): 311
Energy Efficiency TOPS/W (Layer-by-Layer Process): 9.53034
Throughput TOPS (Layer-by-Layer Process): 0.1082
Throughput FPS (Layer-by-Layer Process): 191.586
Compute efficiency TOPS/mm^2 (Layer-by-Layer Process): 0.00547446
Power Density W/mm^2 (Layer-by-Layer Process): 0.000574425
------------------------------------ Hardware Performance Done -------------------------------------

---------------------------- Simulation Performance ------------------------------
Total Run-time of NeuroSim: 110 seconds
---------------------------- Simulation Performance ------------------------------
```

Fig. 28 Example of final output for monolithic 3D integration.

```
------------------------------ Heterogeneous 3D FloorPlan ------------------------------

For layer-by-layer scheme, we assumed multiple memory tiers (like memory cube) on top of a logic tier (at bottom)

User-defined SubArray Size: 128x128

----------------- # of memory array used for each layer -----------------
layer1: 64
layer2: 144
layer3: 288
layer4: 288
layer5: 576
layer6: 576
layer7: 2048
layer8: 64

----------------- Tier # of each layer in the memory cube, larger value means in higher tier -----------------
layer1: 1
layer2: 1
layer3: 1
layer4: 1
layer5: 2
layer6: 2
layer7: 4
layer8: 4

----------------- Speed-up of each layer ------------------
layer1: 16
layer2: 4
layer3: 4
layer4: 2
layer5: 2
layer6: 1
layer7: 1
layer8: 8

---------------------------- Heterogeneous 3D FloorPlan Done -----------------------------
```

```
---------------------------- Summary -----------------------------

5-Tier Chip Area : 1.44446e+07um^2
4-Tier Memory Cube Area : 385201um^2
Shrinked IC Area : 685537um^2

Chip clock period is: 2.05141ns
Chip layer-by-layer readLatency (per image) is: 1.27019e+06ns
Chip total readDynamicEnergy is: 3.00496e+07pJ
Chip total leakage Energy is: 937699pJ
Chip total leakage Power is: 565.853uW
Chip buffer readLatency is: 1.00082e+06ns
Chip buffer readDynamicEnergy is: 135704pJ
Chip ic readLatency is: 45650.6ns
Chip ic readDynamicEnergy is: 2.41983e+06pJ

********************** Breakdown of Latency and Dynamic Energy ***********************

---------- ADC (or S/As and precharger for SRAM) readLatency is : 63462.3ns
---------- Accumulation Circuits (subarray level: adders, shiftAdds; PE/Tile/Global level: accumulation units) readLatency is : 156293ns
---------- Other Peripheries (e.g. decoders, mux, switchmatrix, buffers, IC, pooling and activation units) readLatency is : 1.05044e+06ns
---------- ADC (or S/As and precharger for SRAM) readDynamicEnergy is : 1.75976e+07pJ
---------- Accumulation Circuits (subarray level: adders, shiftAdds; PE/Tile/Global level: accumulation units) readDynamicEnergy is : 4.9124e+06pJ
---------- Other Peripheries (e.g. decoders, mux, switchmatrix, buffers, IC, pooling and activation units) readDynamicEnergy is : 7.5396e+06pJ

********************** Breakdown of Latency and Dynamic Energy ***********************


---------------------------- Performance -----------------------------
Energy Efficiency TOPS/W (Layer-by-Layer Process): 32.5843
Throughput TOPS (Layer-by-Layer Process): 0.969802
Throughput FPS (Layer-by-Layer Process): 787.283
Compute efficiency TOPS/mm^2 (Layer-by-Layer Process): 0.0671393
------------------------------------ Hardware Performance Done ------------------------------------

---------------------------- Simulation Performance -----------------------------
Total Run-time of NeuroSim: 163 seconds
---------------------------- Simulation Performance -----------------------------
```

Fig. 29 Example of final output for heterogeneous 3D integration.

10. Reference

[1]. 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.

[2]. github.com/neurosim/MLP_NeuroSim_V3.0

[3]. X. Peng et al., "*Benchmarking Monolithic 3D Integration for Compute-in-Memory Accelerators: Overcoming ADC Bottlenecks and Maintaining Scalability to 7nm or Beyond*," 2020 IEEE International Electron Devices Meeting (IEDM), 2020, pp. 30.4.1-30.4.4, doi: 10.1109/IEDM13553.2020.9372091.

[4]. C. -H. Shen et al., "*Monolithic 3D chip integrated with 500ns NVM, 3ps logic circuits and SRAM*," 2013 IEEE International Electron Devices Meeting, 2013, pp. 9.3.1-9.3.4, doi: 10.1109/IEDM.2013.6724593.

[5]. W. Li, S. Huang, X. Sun, H. Jiang, S. Yu, "Secure-RRAM: A 40nm 16kb compute-in-memory 354 macro with reconfigurability, sparsity control, and embedded security," *IEEE Custom Integrated* 355 *Circuits Conference (CICC),* 2021.

[6]. Predictive Technology Model (PTM). Available at http://ptm.asu.edu/

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

[8]. 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.

[9]. 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.

[10]. 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.

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

[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]. P. Jain, U. Arslan, M. Sekhar, B.C. Lin, L. Wei, T. Sahu, J. Alzate-vinasco, A. Vangapaty, M. Meterelliyoz, N. Strutt, A.B. Chen, "*A 3.6 Mb 10.1 Mb/mm 2 Embedded Non-Volatile ReRAM Macro in 22nm FinFET Technology with Adaptive Forming/Set/Reset Schemes Yielding Down to 0.5 V with Sensing Time of 5ns at 0.7 V*," IEEE International Solid-State Circuits Conference (ISSCC), 2019.

[18]. W. He, S. Yin, Y. Kim, X. Sun, J.J. Kim, S. Yu and J.S. Seo, "*2-Bit-per-Cell RRAM based In-Memory Computing for Area-/Energy-Efficient Deep Learning*," IEEE Solid-State Circuits Letters, vol. 3, pp. 194-197, 2020.

[19]. W. Wu, H. Wu, B. Gao, P. Yao, X. Zhang, X. Peng, S. Yu, H. Qian, "*A methodology to improve linearity of analog RRAM for neuromorphic computing*," IEEE Symposium on VLSI Technology (VLSI), 2018.

[20]. W. Kim, R.L. Bruce, T. Masuda, G.W. Fraczak, N. Gong, P. Adusumilli, S. Ambrogio, H. Tsai, J. Bruley, J.P. Han, M. Longstreet, "*Confined PCM-based analog synaptic devices offering low resistance-drift and 1000 programmable states for deep learning*," IEEE Symposium on VLSI Technology, 2019.

[21]. K. Ni, B. Grisafe, W. Chakraborty, A.K. Saha, S. Dutta, M. Jerry, J.A. Smith, S. Gupta, S. Datta, "*In-memory computing primitive for sensor data fusion in 28 nm HKMG FeFET technology*," IEEE International Electron Devices Meeting (IEDM), 2018.

[22]. L. Wei, J.G. Alzate, U. Arslan, et al. "*A 7Mb STT-MRAM in 22FFL FinFET technology with 4ns read sensing time at 0.9 V using write-verify-write scheme and offset-cancellation sensing technique*" IEEE International Solid-State Circuits Conference (ISSCC), 2019.