User Manual of DNN+NeuroSim Framework V1.0

Developers: Xiaochen Peng and Shanshi Huang

PI: Prof. Shimeng Yu, Georgia Institute of Technology

June 1, 2019

Index

---

1. Introduction

DNN+NeuroSim is an integrated framework, which is developed in C++ and wrapped by Pytorch and TensorFlow, to emulate the deep neural networks (DNN) inference performance (in V 1.0) or on-chip training (to-be-released) performance on the hardware accelerator based on near-memory computing or in-memory computing architectures. Various device technologies are supported, including SRAM, emerging non-volatile memory (eNVM) based on resistance switching (e.g. RRAM, PCM, STT-MRAM), and ferroelectric FET (FeFET). SRAM is by nature 1-bit per cell, eNVMs and FeFET in this simulator could support either 1-bit or multi-bit per cell. *NeuroSim* [1] is a circuit-level macro model for benchmarking neuro-inspired architectures (including memory array, peripheral logic, and interconnect routing) in terms of circuit-level performance metrics, such as chip area, latency, dynamic energy and leakage power. With Pytorch and TensorFlow wrapper, *DNN +NeuroSim* framework can support hierarchical organization from

the device level (transistors from 130 nm down to 7 nm, eNVM and FeFET device properties) to the circuit level (periphery circuit modules such as analog-to-digital converters, ADCs), to chip level (tiles of processing-elements built up by multiple sub-arrays, and global interconnect and buffer) and then to the algorithm level (different convolutional neural network topologies), enabling instruction-accurate evaluation on the inference 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 and TensorFlow wrapper (i.e. C++ wrapped by python). With the wrapper, users are able to define various network structures, precisions of synaptic weights and neural activations, which 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 the released simulator, an 8-layer VGG (VGG-8) network for CIFAR-10 dataset is provided as a default model in the wrapper, with 8-bit synaptic weights and neural activations, while users could modify the precisions and neural network topologies. The hardware parameters (such as technology nodes, memory cell properties, operation modes, and so on) will be defined under *NeuroSim* in **Param.cpp**.

---

2. System Requirements (Linux)

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

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

---

3. Installation and Usage (Linux)

Step 1: Get the tool from GitHub

```
git clone https://github.com/neurosim/DNN_NeuroSim.git
```

Step 2: Extract **framework** to its current directory

```
tar -zxvf Inference_pytorch.tar.gz
tar -zxvf Inference_tensorflow.tar.gz
```

Step 3: Compile the *NeuroSim* Code

```
make
```

Step 4: Run Pytorch/TensorFlow 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 CUDA 9.0 + cuDNN v7.0.5, python2.7 + tensorflow 1.5.0 (GPU) and python 3.5 + pytorch 1.0(GPU).

---

4. Chip Level Architectures

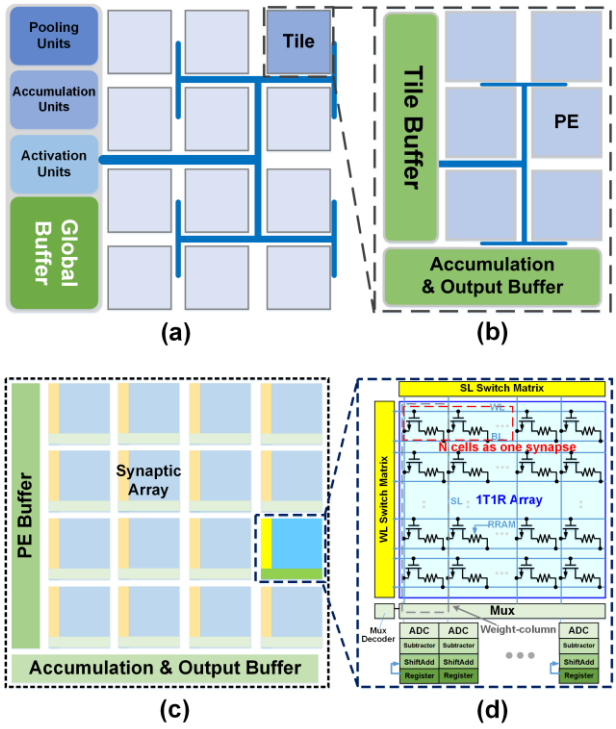

Fig. 1. The diagram of (a) top level of chip architecture, which contains multiple tiles, global buffer, accumulation units, activation units (sigmoid or ReLU) and pooling units; (b) a tile with multiple processing elements (PEs), tile buffer to load in activations, accumulation modules to add up partial 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.

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. 1 shows the modeled chip hierarchy, where the top level of chip is consist of multiple tiles, global buffer, accumulation units,

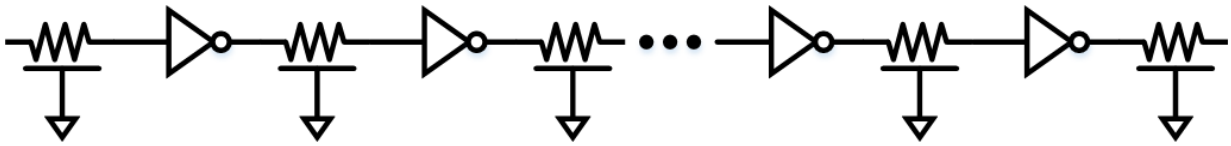

Fig. 2. The diagram of wire with repeaters.

activation units (sigmoid or ReLU), and pooling units. Fig. 1 (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. 1 (c) shows, a PE is built up by a groups of synaptic sub-arrays, PE buffers, accumulation modules and output buffer. In Fig. 1 (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).

4.1 Interconnect: H-Tree

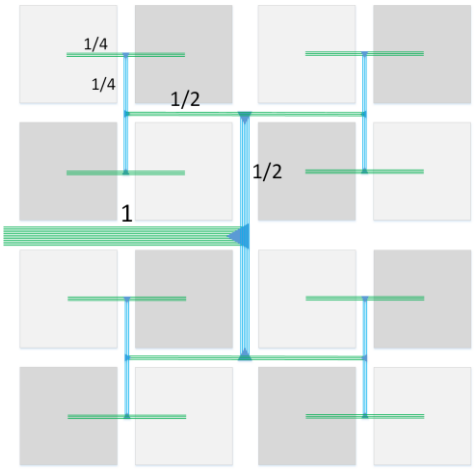

Fig. 3. An example of H-tree for a 4×4 computation-unit array.

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. 2 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 [3] is introduced as EQ (4.1) shows, where R is the resistance of a minimum-sized repeater, C is the gate capacitance, and diffusion capacitance Cp_{inv}, R_w and C_w are the unit resistance and capacitance of wire, respectively.

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

The repeater size should use an NMOS transistor width of

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

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

Fig. 3 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.

4.2 Floorplan of Neural Networks

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

The floorplan starts from tile sizing to PE sizing, while the size of synaptic array is defined by users in **Param.cpp**. With pre-defined network structure and weight mapping method, *NeuroSim* automatically calculate weight-matrix size for each layer (especially for convolutional ones, where 3D kernels will be unrolled to 2D matrixes), 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, and the default 8-layer VGG network in the simulator for CIFAR-10 dataset. The results were based on assumption that one memory cell is sufficient to map one synaptic weight (i.e. an 8-bit cell to map an 8-bit synapse), and synaptic array size is 128×128. With various hardware configuration (such as two 4-bit memory cells form one 8-bit synaptic weight), the memory utilization could be slightly different.

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%

4.3 Weight Mapping Methods

We support two mapping methods in this framework, conventional mapping and novel mapping method which was proposed in [4]. Fig. 4 shows the example of conventional mapping for one convolutional layer, where each 3D kernel (weight) is unrolled into a long column, since the partial sums in each 3D will be summed up to get the final output. Thus, the total kernels in each convolutional layer will form a group of such long columns, i.e., a large weight matrix.

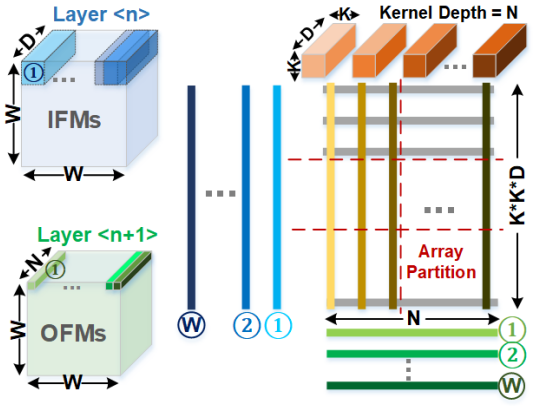

Fig. 4. An example of conventional mapping method of input and weight data.

To get the output feature maps (OFMs), as Fig. 3 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

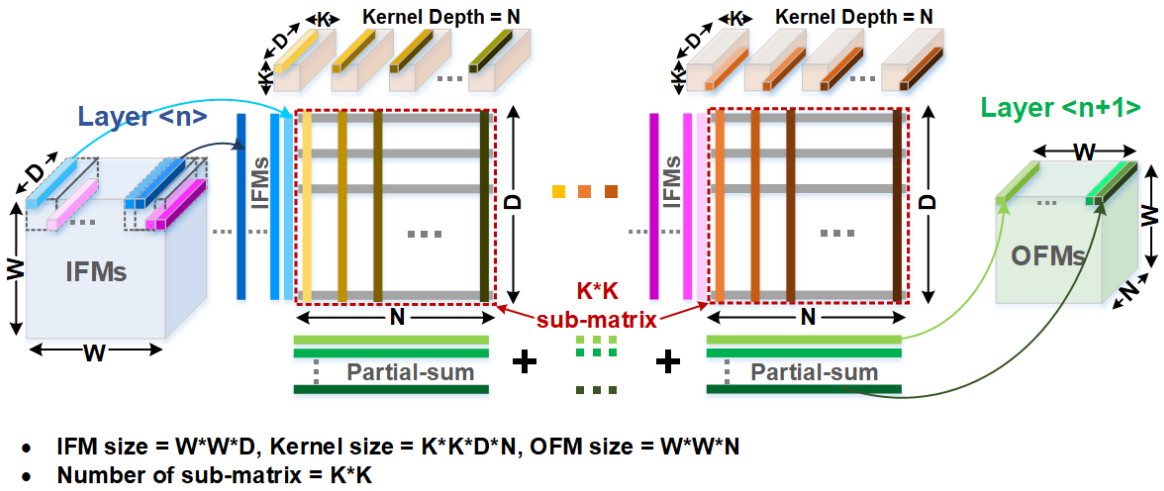

Fig. 5. An example of novel mapping method of input and weight data.

"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. 5 shows an example of novel mapping for the same convolutional layer. Instead of unrolling 3D kernels into a large matrix, the weights at different spatial location of each kernel are mapped into different sub-matrices. According to the spatial location of partitioned kernel data in each kernel, we define which group of these partitioned kernel data should belong to. Hence, K×K sub-matrices are needed for the kernels (whose first and second dimension equal to K and K), since each sub-matrix has size D×N, the size of total weight matrix will be K×K×D×N, which equals to the size of unrolled matrix from conventional mapping method (as Fig. 3 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. Circuit Level: Synaptic Array Architectures

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

5.1 Parallel Synaptic Array Architectures

Fig. 6 and Fig. 7 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 [5]. 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. 6. 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. 6 (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 [6], the input vectors will be fetched in via WL switch matrix, the partial-sums will be collected along columns simultaneously at one time with high-precision flash-ADCs based on multilevel 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).

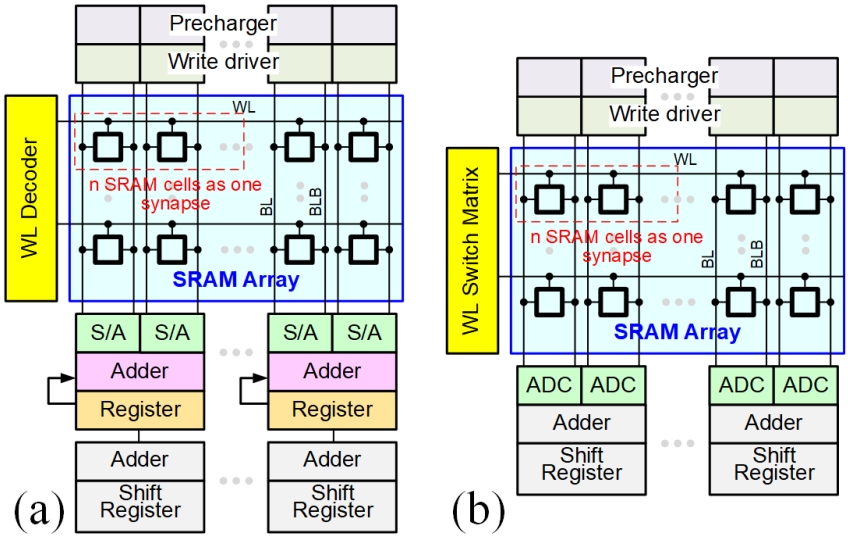

Fig. 6. The diagram of SRAM-based (a) sequential-read-out; (b) parallel-read-out synaptic arrays.

2) Analog eNVM 1T1R synaptic array

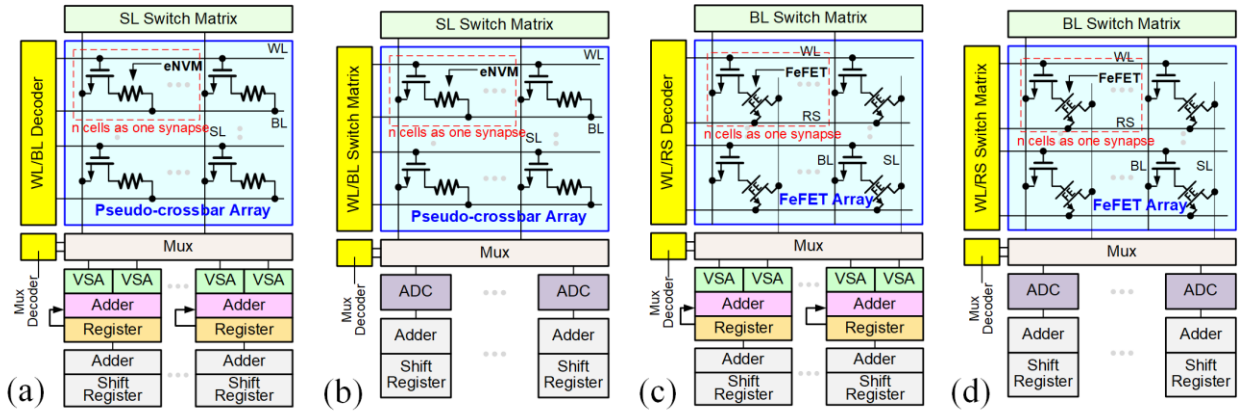

Fig. 7. (a) sequential-read-out and (b) parallel-read-out analog eNVM pseudo-1T1R synaptic arrays; (c) sequential-read-out and (c) parallel-read-out analog FeFET synaptic arrays;

Fig. 7 (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 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. Then the weighted sum currents are digitalized by a current-mode sense amplifier (S/A), and a flash-ADC with multilevel S/A by varying references is used.

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. 6 (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

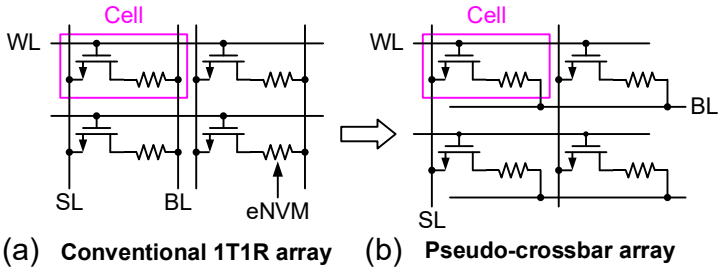

Fig. 8. Transformation from (a) conventional 1T1R array to (b) pseudo-crossbar array by 90° rotation of BL to enable weighted sum operation.

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. 7 (c) and (d), the analog FeFET array is in the pseudo-crossbar fashion as proposed in [7], 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. 9 shown below.

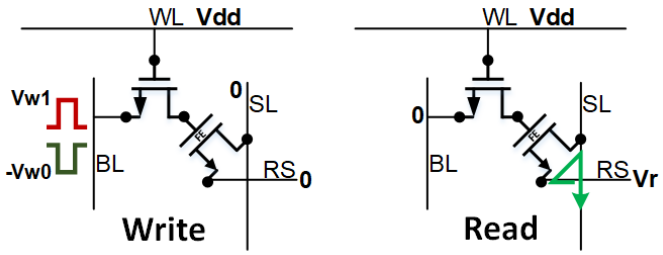

Fig. 9. Operations of (a) write and (b) read in FeFET cell.

5.2 Array Peripheral Circuits

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

1) Switch matrix

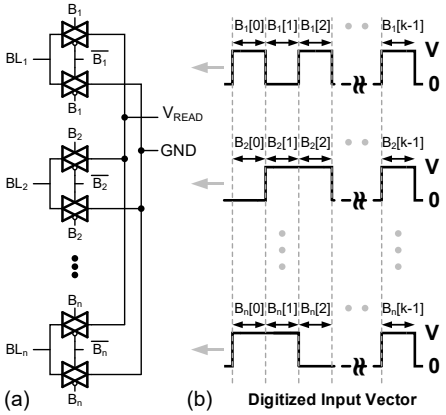

Fig. 10 (a) Transmission gates of the BL switch matrix in the weighted sum operation. A vector of control signals (B1 to Bn) from the registers (not shown here) decide the BLs to be connected to either a voltage source or ground. (b) Control signals in a bit stream to represent the precision of the input vector.

Switch matrices are used for fully parallel voltage input to the array rows or columns. Fig 10 (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 10 (b). The reason why we do not use analog voltage to represent the input vector precision is the I-V nonlinearity of eNVM cell, which will cause the weighted sum distortion or inaccuracy as discussed above. In the simulator, all the switch matrices (**slSwitchMatrix**, **blSwitchMatrix** and **wlSwitchMatrix**) are instantiated from **SwitchMatrix** class in **SwitchMatrix.cpp**, this module is used in parallel-read-out synaptic arrays

2) Crossbar WL decoder

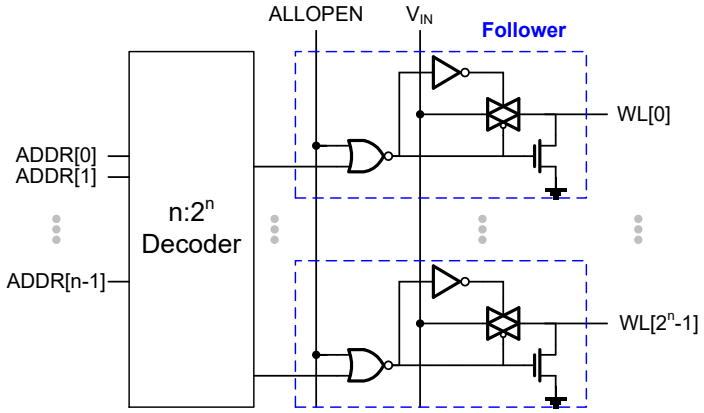

Fig. 11 Circuit diagram of the crossbar WL decoder. Follower circuit is attached to every row of the decoder to enable activation of all WLs when ALLOPEN=1.

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. 11. 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

3) Decoder driver

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

4) New Decoder Driver and Switch Matrix

One should be noticed that, for eNVM pseudo-crossbar and FeFET synaptic arrays, the WLs and BLs/RSs could be controlled by same input signals, but with different voltage values, thus, it could significantly save the area for unnecessary BL/RS switch matrix. To achieve this function, there are several extra control gates to be added into the WL decoder driver circuits, and into the WL switch matrix. Fig. 14 shows the circuit

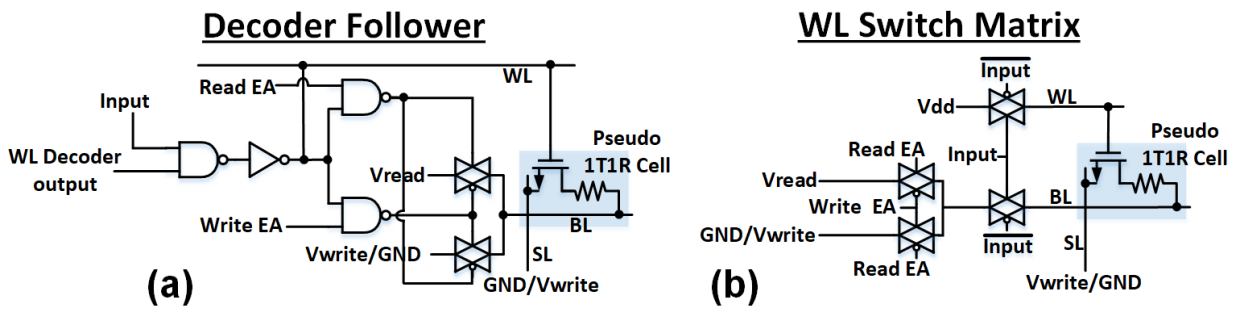

Fig. 12 Circuit diagram of (a) decoder follower and (b) WL switch matrix, which are used to control both WLs and BLs simultaneously, for pseudo-1T1R synaptic arrays.

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. 12 (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. 12 (b), the each single WL switch matrix has two extra transmission gates to be used to send two separate voltages into the corresponding WL and BL. In 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.

5) Multiplexer (Mux) and Mux decoder

The Multiplexer (Mux) is used for sharing the read periphery circuits among synaptic array columns, because the array cell size is much smaller than the size of read periphery circuits and it will not be area-efficient to put all the read periphery circuits 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**.

6) 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), a group of flash-ADCs with multilevel S/A by varying references are used at the end of SLs to generate digital outputs. In the simulator, we take a conventional current-sense-amplifier (CSA) as shown in Fig. 13, as the unit circuit module, to build up multilevel S/A. To precisely estimate the latency and energy of S/A, we run Cadence simulation across technology from 130nm to 7nm, for each technology node, we chose reasonable BL current range (considering practical device resistance range), and in the range we select multiple specific nodes I_{BL}, detect the latency and power trends of each specific I_{BL} when sweeping I_{ref} (i.e. from $0.001\times I_{BL}$ to $1000\times I_{BL}$). As a detection of multiple experiments based on 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.

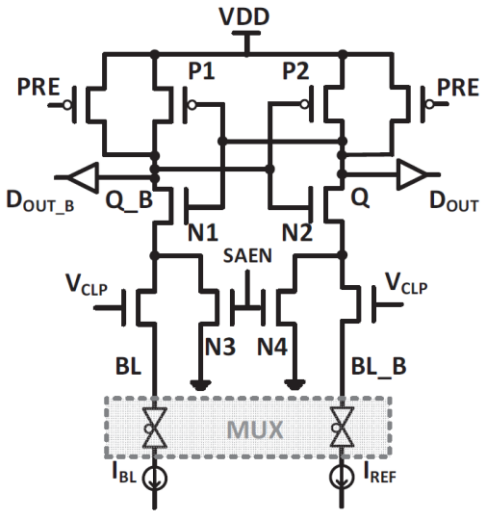

Fig. 13 Schematic of current sense amplifier (CSA).

Then, we sweep the technology nodes, at each technology node, we sweep the I_{BL}, and for each I_{BL}, we sweep the I_{ref}. We collect all the simulated data from 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. 14 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.

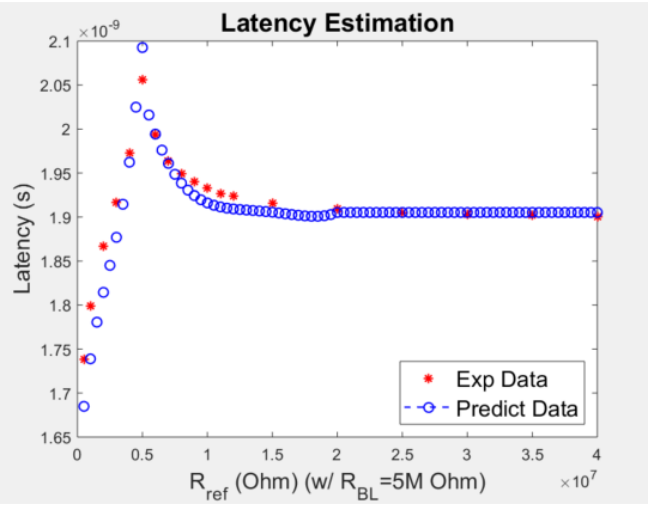

Fig. 14 An example of latency estimation based on fitting functions compared with Cadence results.

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. 15 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 [8]. 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 [8], the results in Fig. 15 surmise that higher weight-precision generally requires higher ADC-precision.

With larger synaptic array size or higher cell precision, higher ADC precision is demanded. For flash-ADC, higher than 3-bit may still result in significant area overhead, thus more compact ADC design is still under development for future release.

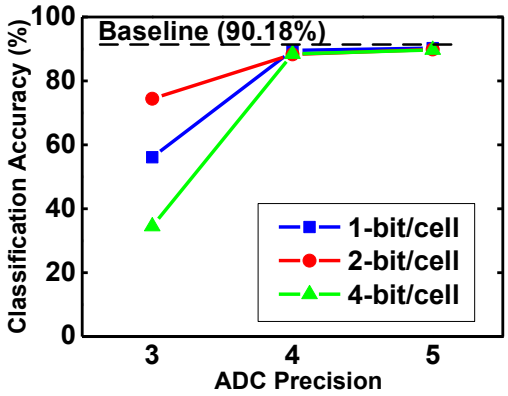

Fig. 15. Classification accuracy of CIFAR-10 for an 8-bit CNN as a function of the ADC precision for partial sums.

7) Adder and register

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

8) Adder and shift register

The adder and shift register pair at the bottom of synaptic core performs shift and add of the weighted sum result at each input vector bit cycle (B_1 to B_n in Fig 10 (b)) to get the final weighted sum. The bit-width of the adder and shift register needs to be further extended depending on the precision of input vector. If the values in the input vector are only 1 bit, then the adder and shift register pair is not required. In the simulator, a collection of the adder and shift register pairs (**ShiftAdd**) is instantiated from **ShiftAdd** class in **ShiftAdd.cpp**, where **ShiftAdd** further contains a group of adders (**adder**) instantiated from **Adder** class in **Adder.cpp** and a group of registers (**dff**) instantiated from **DFF** class in **DFF.cpp**.

6. Algorithm Level: PyTorch and TensorFlow Wrapper

The algorithm we use to get the quantized DNN model for inference is the WAGE from [9]. The TensorFlow code is modified based on the author released code at [10]. The Pytorch code realizes the same algorithm except that we move the scale term from weight to output to make it more suitable for the hardware architecture. We referenced [11] [12] for our Pytorch code. This algorithm could directly train a quantized network with user defined bit width for weight, activation, gradient and error. The partial sum quantization (according to ADC precision) is to be released in future version.

In V1.0, 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 second scheme using WAGE since WAGE quantize both weight and activation using fixed quantization level, which is [-1, 1] with scale of 2^{-b}. This mechanism is friendly to hardware implementation, which normally represent data use 2's complimentary. WAGE also apply quantization to gradient and error, which is not necessary for inference stage (but 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.

---

7. How to run *DNN +NeuroSim*

1) Define Network Structure in NetWork.csv

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

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.

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.

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**.

```
g++ -c -fopenmp -O3 -std=c++0x -w       NewMux.cpp -o NewMux.o
g++ -c -fopenmp -O3 -std=c++0x -w       ProcessingUnit.cpp -o ProcessingUnit.o
g++ -c -fopenmp -O3 -std=c++0x -w       Bus.cpp -o Bus.o
g++ -c -fopenmp -O3 -std=c++0x -w       XorArbiterPuf.cpp -o XorArbiterPuf.o
g++ -c -fopenmp -O3 -std=c++0x -w       ArbiterPuf.cpp -o ArbiterPuf.o
g++ -c -fopenmp -O3 -std=c++0x -w       formula.cpp -o formula.o
g++ -c -fopenmp -O3 -std=c++0x -w       DFF.cpp -o DFF.o
g++ -c -fopenmp -O3 -std=c++0x -w       FunctionUnit.cpp -o FunctionUnit.o
g++ -c -fopenmp -O3 -std=c++0x -w       RippleCounter.cpp -o RippleCounter.o
g++ -c -fopenmp -O3 -std=c++0x -w       Adder.cpp -o Adder.o
g++ -c -fopenmp -O3 -std=c++0x -w       Technology.cpp -o Technology.o
g++ -c -fopenmp -O3 -std=c++0x -w       MultilevelSenseAmp.cpp -o MultilevelSenseAmp.o
g++ -c -fopenmp -O3 -std=c++0x -w       SwitchMatrix.cpp -o SwitchMatrix.o
g++ -c -fopenmp -O3 -std=c++0x -w       CurrentSenseAmp.cpp -o CurrentSenseAmp.o
g++ -c -fopenmp -O3 -std=c++0x -w       NewSwitchMatrix.cpp -o NewSwitchMatrix.o
g++ -c -fopenmp -O3 -std=c++0x -w       WLNewDecoderDriver.cpp -o WLNewDecoderDriver.o
g++ -c -fopenmp -O3 -std=c++0x -w       BitShifter.cpp -o BitShifter.o
g++ -c -fopenmp -O3 -std=c++0x -w       VoltageSenseAmp.cpp -o VoltageSenseAmp.o
g++ -c -fopenmp -O3 -std=c++0x -w       MaxPooling.cpp -o MaxPooling.o
g++ -c -fopenmp -O3 -std=c++0x -w       Sigmoid.cpp -o Sigmoid.o
g++ -c -fopenmp -O3 -std=c++0x -w       Param.cpp -o Param.o
g++ -c -fopenmp -O3 -std=c++0x -w       SubArray.cpp -o SubArray.o
g++ -c -fopenmp -O3 -std=c++0x -w       AdderTree.cpp -o AdderTree.o
g++ -c -fopenmp -O3 -std=c++0x -w       Comparator.cpp -o Comparator.o
g++ -c -fopenmp -O3 -std=c++0x -w       DecoderDriver.cpp -o DecoderDriver.o
g++ -c -fopenmp -O3 -std=c++0x -w       Subtractor.cpp -o Subtractor.o
g++ -c -fopenmp -O3 -std=c++0x -w       MultilevelSAEncoder.cpp -o MultilevelSAEncoder.o
g++ -c -fopenmp -O3 -std=c++0x -w       Chip.cpp -o Chip.o
g++ -c -fopenmp -O3 -std=c++0x -w       Precharger.cpp -o Precharger.o
g++ -c -fopenmp -O3 -std=c++0x -w       RowDecoder.cpp -o RowDecoder.o
g++ -c -fopenmp -O3 -std=c++0x -w       Mux.cpp -o Mux.o
g++ -c -fopenmp -O3 -std=c++0x -w       SenseAmp.cpp -o SenseAmp.o
g++ -c -fopenmp -O3 -std=c++0x -w       Buffer.cpp -o Buffer.o
g++ -c -fopenmp -O3 -std=c++0x -w       WLDecoderOutput.cpp -o WLDecoderOutput.o
g++ -c -fopenmp -O3 -std=c++0x -w       ReadCircuit.cpp -o ReadCircuit.o
g++ -c -fopenmp -O3 -std=c++0x -w       DeMux.cpp -o DeMux.o
g++ -c -fopenmp -O3 -std=c++0x -w       Tile.cpp -o Tile.o
g++ -c -fopenmp -O3 -std=c++0x -w       ShiftAdd.cpp -o ShiftAdd.o
g++ -c -fopenmp -O3 -std=c++0x -w       HTree.cpp -o HTree.o
g++ -c -fopenmp -O3 -std=c++0x -w       SRAMWriteDriver.cpp -o SRAMWriteDriver.o
g++ -c -fopenmp -O3 -std=c++0x -w       SramNewSA.cpp -o SramNewSA.o
g++ -c -fopenmp -O3 -std=c++0x -w       main.cpp -o main.o
g++ -fopenmp -O3 -std=c++0x -w   NewMux.o ProcessingUnit.o Bus.o XorArbiterPuf.o ArbiterPuf.o formula.o DFF.o FunctionUnit.o RippleCounter.o Adder.o Technology.o MultilevelSenseAmp.o SwitchMatrix.o CurrentSe
nseAmp.o NewSwitchMatrix.o WLNewDecoderDriver.o BitShifter.o VoltageSenseAmp.o MaxPooling.o Sigmoid.o Param.o SubArray.o AdderTree.o Comparator.o DecoderDriver.o Subtractor.o MultilevelSAEncoder.o Chip.o Pre
charger.o RowDecoder.o Mux.o SenseAmp.o Buffer.o WLDecoderOutput.o ReadCircuit.o DeMux.o Tile.o ShiftAdd.o HTree.o SRAMWriteDriver.o SramNewSA.o main.o -o main
```

Fig. 16 Output of compilation.

3) Compilation of *NeuroSim*

After modifying the **NetWork.csv** and **Param.cpp** files, or whenever any change is made in the files, the codes have to be recompiled by using `make` command as stated in **Installation and Usage (Linux)** section. If the compilation is successful, a screenshot like Fig. 16 can be expected.

4) Run the program with PyTorch/TensorFlow wrapper

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

Instructions to run the wrapper:

- Tensorflow: (https://www.tensorflow.org/)
  - The bitwidth setting is under source/Option.py
  - Train:
    - cd source/
    - python Top.py (The saveModel path should be set under Option.py)
  - Inference
  - cd source/
    - python Inference.py (Set the loadModel path the same with the saved one)

```
14  loadModel = None
15  # loadModel = '../model/' + '2017-12-06' + '(' + 'vgg7 2888' + ')' + '.tf'
16  saveModel = None
17  # saveModel = '../model/' + Time + '(' + Notes + ')' + '.tf'
18
19  bitsW = 2  # bit width of we ights
20  bitsA = 8  # bit width of activations
21  bitsG = 8  # bit width of gradients
22  bitsE = 8  # bit width of errors
23
```

Fig. 17 part of Option.py

- 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.

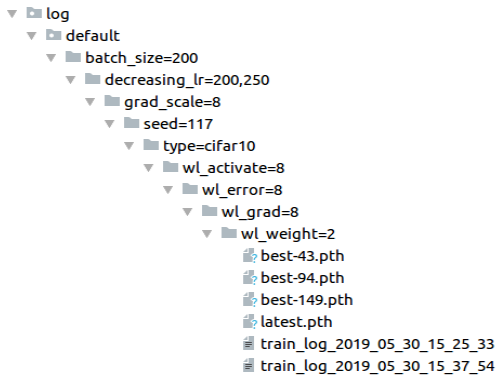

Fig. 18 example of output folder hierarchy

- 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. 19 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. 17 shows an example of final output of an 8-bit VGG-8 inference for one CIFAR-10 image, based on parallel 1T1R synaptic array, with 2-bit per cell RRAM (100KΩ and 10MΩ as R_{on} and R_{off}). The output from the simulation include hardware inference accuracy, memory utilization, 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).

```
Test set: Average loss: 1.5692, Accuracy: 9301/10000 (93%)
------------------------------ FloorPlan ------------------------------
Desired Conventional Mapped Tile Storage Size: 512x512
---------------- # of tile used for each layer ----------------
layer1: 1
layer2: 3
layer3: 3
layer4: 5
layer5: 5
layer6: 9
layer7: 32
layer8: 2

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

---------------- Utilization of each layer -----------------
layer1: 0.210938
layer2: 0.75
layer3: 0.75
layer4: 0.9
layer5: 0.9
layer6: 1
layer7: 1
layer8: 0.078125
Memory Utilization of Whole Chip: 0.914453

--------------------------- FloorPlan Done -----------------------------


------------------------------------- Hardware Performance -------------------------------------
------------------- Estimation of Layer 1 --------------------
layer1's readLatency is: 10128.4ns
layer1's readDynamicEnergy is: 643389pJ
layer1's leakageEnergy is: 181327pJ
layer1's buffer latency is: 2851.03ns
layer1's buffer readDynamicEnergy is: 120388pJ
layer1's ic latency is: 364.561ns
layer1's ic readDynamicEnergy is: 144913pJ
------------------- Estimation of Layer 2 --------------------
...
layer4's readDynamicEnergy is: 8.44954e+06pJ
layer4's leakageEnergy is: 553123pJ
layer4's buffer latency is: 1061.89ns
layer4's buffer readDynamicEnergy is: 1.47842e+06pJ
layer4's ic latency is: 694.818ns
layer4's ic readDynamicEnergy is: 668609pJ
------------------- Estimation of Layer 5 --------------------
layer5's readLatency is: 11596ns
layer5's readDynamicEnergy is: 2.83455e+06pJ
layer5's leakageEnergy is: 193526pJ
layer5's buffer latency is: 195.041ns
layer5's buffer readDynamicEnergy is: 295303pJ
layer5's ic latency is: 138.493ns
layer5's ic readDynamicEnergy is: 240647pJ
------------------- Estimation of Layer 6 --------------------
layer6's readLatency is: 21410.8ns
layer6's readDynamicEnergy is: 5.61653e+06pJ
layer6's leakageEnergy is: 331339pJ
layer6's buffer latency is: 195.041ns
layer6's buffer readDynamicEnergy is: 579484pJ
layer6's ic latency is: 212.415ns
layer6's ic readDynamicEnergy is: 436923pJ
------------------- Estimation of Layer 7 --------------------
layer7's readLatency is: 343.744ns
layer7's readDynamicEnergy is: 818426pJ
layer7's leakageEnergy is: 2920.54pJ
layer7's buffer latency is: 5.41782ns
layer7's buffer readDynamicEnergy is: 168597pJ
layer7's ic latency is: 9.55089ns
layer7's ic readDynamicEnergy is: 42464.9pJ
------------------- Estimation of Layer 8 --------------------
layer8's readLatency is: 35.6463ns
layer8's readDynamicEnergy is: 11735.7pJ
layer8's leakageEnergy is: 627.354pJ
layer8's buffer latency is: 5.41782ns
layer8's buffer readDynamicEnergy is: 4411.01pJ
layer8's ic latency is: 1.29921ns
layer8's ic readDynamicEnergy is: 542.531pJ
---------------------------- Summary -----------------------------
ChipArea : 2.85244e+06um^2
Chip total readLatency is: 142395ns
Chip total readDynamicEnergy is: 3.37863e+07pJ
Chip total leakage Energy is: 2.39987e+06pJ
Chip buffer readLatency is: 10251.8ns
Chip buffer readDynamicEnergy is: 6.75296e+06pJ
Chip ic readLatency is: 3663.72ns
Chip ic readDynamicEnergy is: 2.98604e+06pJ

--------------------------- Performance -----------------------------
Energy Efficiency TOPS/W (Layer-by-Layer Process): 17.0208
Throughput FPS (Layer-by-Layer Process): 7022.69
------------------------------------ Hardware Performance Done ------------------------------------
```

Fig. 20 Example of final output.

8. Upcoming Version

Currently, this *DNN +NeuroSim V1.0* can only support hardware estimation for on-chip inference, where the network is assumed to be processed layer-by-layer, this could cause leakage energy and longer latency, pipeline design is to be released in future versions to improve this aspect.

In addition, the hardware simulation for on-chip training will be supported in the upcoming version.

---

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

[4]. 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.

[5]. 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.

[6]. 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.

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

[8]. 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.

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

[10]. github.com/boluoweifenda/WAGE

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

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