Manual of *MNSIM_Python*: A Behavior-Level Modeling Tool for NVM-based CNN Accelerators

Zhenhua Zhu[1,*], Hanbo Sun[1], Kaizhong Qiu[1], Lixue Xia[2], Gokul Krishnan[6], Dimin Niu[2], Qiuwen Lou[3], Xiaoming Chen[4], Yuan Xie[2,5], Yu Cao[6], X. Sharon Hu[3], Yu Wang[1,*], and Huazhong Yang[1]

[1]**Dept. of EE, BNRist, Tsinghua University**
[2]**Alibaba Group**
[3]**University of Notre Dame**
[4]**Institute of Computing Technology, Chinese Academy of Sciences**
[5]**University of California, Santa Barbara**
[6]**Arizona State University**
[*]**zhuzhenh18@mails.tsinghua.edu.cn, yu-wang@tsinghua.edu.cn**

ABSTRACT

MNSIM_Python is a behavior-level modeling tool for NVM-based CNN accelerators and the version 1.0 is still a beta version. If you have any questions and suggestions about MNSIM_Python please contact us via e-mail. We hope that MNSIM_Python can be helpful to your research work, and sincerely invite every Processing-In-Memory researcher to add your ideas to MNSIM_Python to enlarge its function.

CONTENTS

1 INTRODUCTION

MNSIM_Python is a behavior-level modeling tool for NVM-based CNN accelerators, which is developed in Python. Compared with the former version MNSIM (available in: https://github.com/Zhu-Zhenhua/MNSIM_V1.1), MNSIM_Python models the CNN computing accuracy and hardware performance (i.e., area, power, energy, and latency) in behavior-level. As shown in Figure 1, this tool is developed for Non-Volatile Memory (NVM) based Processing-In-Memory (PIM) architecture designers and CNN algorithm researchers who want to fast evaluate the CNN accuracy and hardware performance of their architecture or algorithm model design. It should be noted that this tool is mainly used to estimate and compare the relative dis-/advantages of different architecture/NN design solutions. For achieving more accurate simulation results, please use circuits-level simulators.

MNSIM_Python is designed based on these papers:

[IEEE TCAD] Lixue Xia, Boxun Li, Tianqi Tang, Peng Gu, Pai-yu Chen, Shimeng Yu, Yu Cao, Yu Wang, Yuan Xie, Huazhong Yang, MNSIM: Simulation Platform for Memristor-based Neuromorphic Computing System, in IEEE TCAD, vol.37, No.5, 2018, pp.1009-1022.

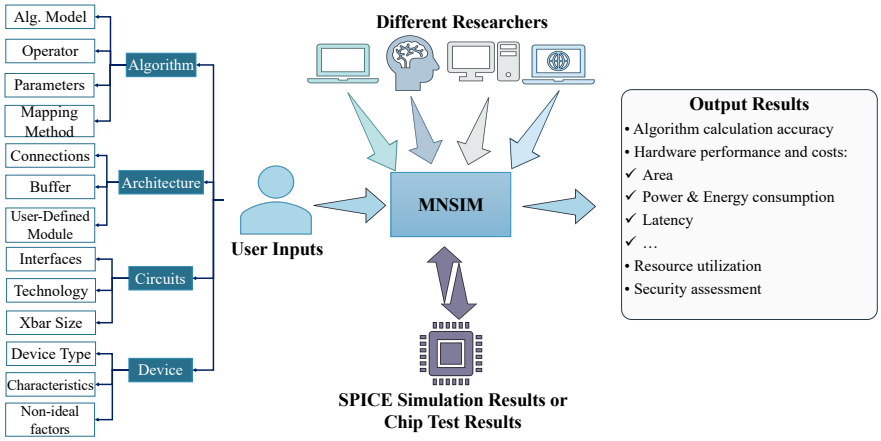

Figure 1. The overview of MNSIM_Python

[DAC'19] Zhenhua Zhu, Hanbo Sun, Yujun Lin, Guohao Dai, Lixue Xia, Song Han, Yu Wang, Huazhong Yang, A Configurable Multi-Precision CNN Computing Framework Based on Single Bit RRAM, in DAC, 2019.

[ASPDAC'20] Hanbo Sun, Zhenhua Zhu, Yi Cai, Xiaoming Chen, Yu Wang, Huazhong Yang, An Energy- Efficient Quantized and Regularized Training Framework for Processing-In-Memory Accelerators, to appear in ASP-DAC 2020, 2020.

[ASPDAC'17] Tianqi Tang, Lixue Xia, Boxun Li, Yu Wang, Huazhong Yang, Binary Convolutional Neural Network on RRAM, in ASP-DAC 2017, 2017, pp.782-787.

Thanks for using MNSIM_Python.

2 RUNNING MNSIM_PYTHON

2.1 Basic running method

1^{st}: Make sure the MNSIM_Python location is added into the system environment variables:

e.g.: export PYTHONPATH=PYTHONPATH:/Users/user1/MNSIM_Python/

2^{nd}: Download the default weights files to the file /MNSIM_Python/:
https://cloud.tsinghua.edu.cn/d/e566b3daaed44804b640/.

3^{rd}: Go to the tool directory and run MNSIM_Python:

e.g.: cd /Users/user1/MNSIM_Python/
python main.py

2.2 Parser information

The detailed parser information is listed in Table 1.

Here we show two examples about how to use the parser information:

e.g.: **Simulate the NN computing accuracy considering SAFs and device variations**
python main.py -SAF -Var

e.g.: **Simulate** AlexNet **and the weights file is stored in** [/example/AlexNet.pth]
python main.py -NN 'AlexNet' -Weights "/example/AlexNet.pth"

2.3 Hardware description and modification

In MNSIM_Python, we propose a basic PIM architecture assumption as shown in Figure 2. Users can describe their PIM architectures design with a few modifications (e.g., change the crossbar size, PE number, or add new hardware modules). More details of the architecture design will be discussed in Section 3.

[SimConfig.ini] is the hardware configuration description file, which contains eight parts:

Table 1. Parser information

Parser	Description	Default
-HWdes –hardware_description	Hardware description file location and file name	/MNSIM_Python/ SimConfig.ini
-Weights –weights	NN weights file location and file name	/MNSIM_Python/ vgg8_params.pth
-NN –NN	NN model name	vgg8
-DisHW –disable_hardware_modeling	Disable hardware modeling	False
-DisAccu –disable_accuracy_simulation	Disable accuracy simulation	False
-SAF –enable_SAF	Enable MNSIM_Python to simulate the effect of Stuck-At-Fault	False
-Var –enable_variation	Enable MNSIM_Python to simulate the effect of device variation	False
-FixRange –enable_fixed_Qrange	Enable MNSIM_Python to fix ADC quantization range (-|max|, |max|)	False
-DisPipe –disable_inner_pipeline	Disable the inner-layer pipeline structure modeling in MNSIM_Python	False
-D –device	Determine the device (platform) running MNSIM_Python (input the GPU id, None is CPU)	CPU
-DisModOut –disable_module_output	Disable module simulation results output, only output the entire system simulation results	False
-DisLayOut –disable_layer_output	Disable layer-wise simulation results output, only output the whole NN model simulation results	False

1. **[Device level]:** model the device characteristics (e.g., device area, read/write latency, etc.)
2. **[Crossbar level]:** model the crossbar configuration (e.g., crossbar size)
3. **[Interface level]:** describe characteristics of Analog-to-Digital Converters (ADCs) and Digital-to-Analog Converters (DACs) (e.g., ADDA area, resolution, power, etc.)
4. **[Process element level]:** model the PE configuration (Figure 2(3). e.g., crossbar number in PE)
5. **[Digital module level]:** model the digital module configuration (e.g., registers, shifters, adders.)
6. **[Tile level]:** model the tile configuration (Figure 2(2). e.g., PE number in one tile)
7. **[Architecture level]:** describe the architecture level configuration and buffer design (e.g., buffer type and size)
8. **[Algorithm level]:** Configure the simulation settings (needs to be updated later)

For more details about **[SimConfig.ini]**, users can refer to the default configuration file (file path: /MNSIM_Python/SimConfig.ini).

2.4 CNN description and weights file

In MNSIM_Python, we provide four basic network models and weights files, i.e., LeNet (-NN 'lenet'), AlexNet (-NN 'alexnet'), VGG-8 (-NN 'vgg8'), and VGG-16 (-NN 'vgg16'). These basic network models are trained on Cifar-10. The basic weights file can be downloaded from: https://cloud.tsinghua.edu.cn/d/e566b3daaed44804b640/.

If the users want to test their own CNN models other than the default CNN models, two steps are needed:

1. Describe the user designed network structure in /MNSIM_Python/MNSIM/ Interface/network.py **get_net(hardware_config, cate)**. For example, the code describing AlexNet is shown below.

   Here, the input variable string 'alexnet' is the NN model name used in input parser, other required information is shown in Table 2. What is more, MNSIM_Python also supports the multi-precision

```
if cate.startswith('alexnet'):
    layer_config_list.append({'type': 'conv', 'in_channels': 3, 'out_channels': 64,
'kernel_size':3, 'padding':1, 'stride':2})
    layer_config_list.append({'type': 'relu'})
    layer_config_list.append({'type': 'pooling', 'mode': 'MAX', 'kernel_size': 2, 'stride': 2})
    layer_config_list.append({'type': 'conv', 'in_channels': 64,
'out_channels': 192, 'kernel_size': 3, 'padding': 1})
    layer_config_list.append({'type': 'relu'})
    layer_config_list.append({'type': 'pooling', 'mode': 'MAX', 'kernel_size': 2, 'stride': 2})
    layer_config_list.append({'type': 'conv', 'in_channels': 192,
'out_channels': 384, 'kernel_size': 3, 'padding': 1})
    layer_config_list.append({'type': 'relu'})
    layer_config_list.append({'type': 'conv', 'in_channels': 384,
'out_channels': 256, 'kernel_size': 3, 'padding': 1})
    layer_config_list.append({'type': 'relu'})
    layer_config_list.append({'type': 'conv', 'in_channels': 256,
'out_channels': 256, 'kernel_size': 3, 'padding': 1})
    layer_config_list.append({'type': 'relu'})
    layer_config_list.append({'type': 'pooling', 'mode': 'MAX', 'kernel_size': 2, 'stride': 2})
    layer_config_list.append({'type': 'view'})
    layer_config_list.append({'type': 'fc', 'in_features': 1024, 'out_features': 4096})
    layer_config_list.append({'type': 'relu'})
    layer_config_list.append({'type': 'fc', 'in_features': 4096,'out_features': 4096})
    layer_config_list.append({'type': 'relu'})
    layer_config_list.append({'type': 'fc', 'in_features': 4096,'out_features': 10})
```

CNN (i.e., different layers have different weights', input activations', and output activations' precision). Users need to add descriptions of the precision parameters of each layer after defining the CNN structure in /MNSIM_Python/MNSIM/Interface/network.py. For example, if we want to specify that the parameters of each layer are the same (weight precision is 9-bit, activation precision is 9-bit, and the fixed-point decimal point position is -2), the code is shown below:

```
for i in range(len(layer_config_list)):
    quantize_config_list.append({'weight_bit': 9, 'activation_bit': 9, 'point_shift': -2})
    input_index_list.append([-1])
```

2. Provide the weights file (*.pth) of the user designed network. The weights file is required to be generated by PyTorch (with torch.save).

Table 2. Layer required information

Layer Type	Variable	Description
conv (Convolutional layer + batch norm operations)	in_channels	Input channel number
	out_channels	Output channel number
	kernel_size	The convolutional kernel size
	stride	The stride size of the sliding window
relu (Nonlinear activation layer)	–	Current version only supports ReLU
pooling (Pooling layer)	mode	Pooling function type: max pooling (MAX) or average (AVG) pooling
	kernel_size	Pooling window's size
	stride	The stride size of the sliding window
view	–	Change the 3D matrix to 1D vector (transition between conv and fc layer)
fc (Fully-connected layer)	in_features	The length of fc layer's input vector
	out_features	The length of fc layer's output vector

2.5 Case study: VGG8

In this section, we will show a case study: simulation and modeling of VGG8.
Firstly, download the source code from GitHub:

```
:~/zzh$ git clone https://github.com/Zhu-Zhenhua/MNSIM_Python.git
Cloning into 'MNSIM_Python'...
remote: Enumerating objects: 9, done.
remote: Counting objects: 100% (9/9), done.
remote: Compressing objects: 100% (9/9), done.
remote: Total 1172 (delta 3), reused 0 (delta 0), pack-reused 1163
Receiving objects: 100% (1172/1172), 1.33 MiB | 28.00 KiB/s, done.
Resolving deltas: 100% (782/782), done.
```

Then, add the MNSIM_Python file path into the system environment variables and go to the directory:

```
:~/zzh$ export PYTHONPATH=PYTHONPATH:../zzh/MNSIM_Python/
:~/zzh$ cd MNSIM_Python/
```

Next, run the MNSIM_Python and the data set will be downloaded at the begining.

```
:~/zzh/MNSIM_Python$ python main.py
Hardware description file location: /home/cengsl14/zzh/MNSIM_Python/SimConfig.ini
Software model file location: /home/cengsl14/zzh/MNSIM_Python/vgg8_params.pth
Whether perform hardware simulation: True
Whether perform accuracy simulation: True
Whether consider SAFs: False
Whether consider variations: False
Quantization range: dynamic range (depends on the data distribution)
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to /home/cengsl14/zzh/MNSIM_Python/MNSIM/Interface/cifar10/cifar-10-python.tar.gz
100.0%Files already downloaded and verified
==================================
```

The output information contains four parts:

1. The CNN model information:

```
==================================
CNN model information:
Layer number: 12
    Layer 0 : conv
    |----Input Size: [32, 32]
    |----Input Precision: 9
    |----Kernel Size: 3
    |----Weight Precision: 9
    |----Input Channel: 3
    |----Stride: 1
    |----Output Size: [32, 32]
    |----Output Channel: 128
```

2. Hardware utilization and performance:

```
Hardware performance finished!
Tile number: 64
Resource utilization: 0.8865585327148438
Hardware area: 869767118.8479999 um^2
               crossbar area: 386547056.64000005 um^2
               DAC area: 87031.80799999999 um^2
               ADC area: 432537600 um^2
               digital part area: 50595430.400000006 um^2
                       |---adder area: 1609760.7680000004 um^2
                       |---shift-reg area: 48454041.6 um^2
                       |---input_demux area: 265814.01599999995 um^2
                       |---output_mux area: 265814.01599999995 um^2
Hardware power: 970.8912674693336 W
               crossbar power: 5.751273813333334 W
               DAC power: 1.8452928 W
               ADC power: 952.0640000000008 W
               digital part power: 11.230700856000004 W
                       |---adder power: 0.00221836 W
                       |---shift-reg power: 11.091800000000001 W
                       |---input_demux power: 0.06813388799999999 W
                       |---output_mux power: 0.06854860799999997 W
```

3. Computing latency (latency of each layer and the entire model):

```
---------------------------------------------
Layer 10  type: pooling
Occupancy: 1.0
Buffer latency of layer 10 : 860.8664205126373 ( 65.04 %)
Computing latency of layer 10 : 0 ( 0.00 %)
     DAC latency of layer 10 : 0 ( 0.00 %)
     ADC latency of layer 10 : 0 ( 0.00 %)
     xbar latency of layer 10 : 0 ( 0.00 %)
Digital part latency of layer 10 : 2.0 ( 0.15 %)
Intra tile communication latency of layer 10 : 0 ( 0.00 %)
Inter tile communication latency of layer 10 : 460.8 ( 34.81 %)
     One layer merge latency of layer 10 : 0 ( 0.00 %)
     Inter tile transfer latency of layer 10 : 460.8 ( 34.81 %)
---------------------------------------------
Layer 11  type: fc
Occupancy: 1.0
Buffer latency of layer 11 : 28.542017331982017 ( 5.41 %)
Computing latency of layer 11 : 405.0181980181818 ( 76.72 %)
     DAC latency of layer 11 : 80.0 ( 15.15 %)
     ADC latency of layer 11 : 181.8181818181818 ( 34.44 %)
     xbar latency of layer 11 : 63.2000162 ( 11.97 %)
Digital part latency of layer 11 : 94.0 ( 17.81 %)
Intra tile communication latency of layer 11 : 0.37109375 ( 0.07 %)
Inter tile communication latency of layer 11 : 0.0 ( 0.00 %)
     One layer merge latency of layer 11 : 0.0 ( 0.00 %)
     Inter tile transfer latency of layer 11 : 0.0 ( 0.00 %)
---------------------------------------------
Entire latency: 1892733.8156580816 ns
```

4. CNN classification accuracy (accuracy results based on GPUs and PIM systems):

```
====================================
Accuracy simulation will take a few minutes on GPU
Original accuracy: 0.5990909090909091
PIM-based computing accuracy: 0.5581818181818182
```

3 ARCHITECTURE DESIGN USED IN MNSIM_PYTHON

In order to model the computing accuracy and hardware performance of PIM accelerators under different architecture design parameters, we propose a basic architecture assumption for MNSIM_Python, which is shown in Figure 2. The architecture design refers to our DAC'19 paper: *Zhenhua Zhu, Hanbo Sun, Yujun*

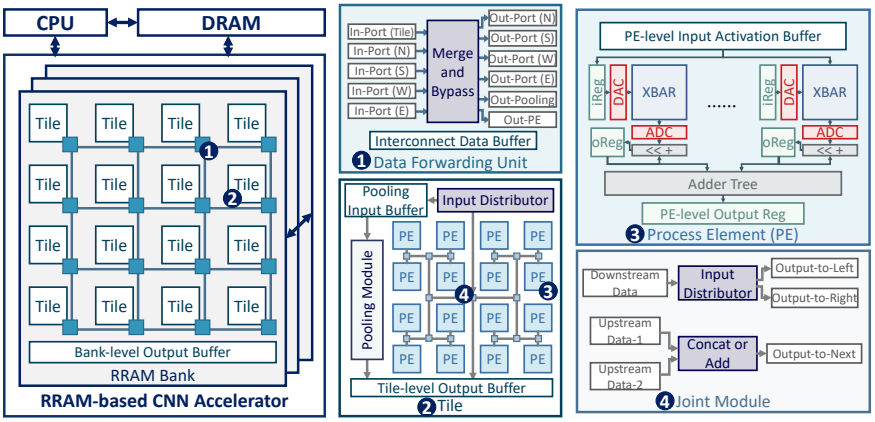

Figure 2. The overall architecture assumption used in MNSIM_Python

Lin, Guohao Dai, Lixue Xia, Song Han, Yu Wang, Huazhong Yang, A Configurable Multi-Precision CNN Computing Framework Based on Single Bit RRAM, in Design Automation Conference (DAC), 2019.
In this paper, we demonstrated that multi-precision CNN quantization can improve the classification accuracy while reducing the storage burden and computing latency further. To support the acceleration of multi-precision CNNs in limited precision device-based PIM accelerators (e.g., 1-bit RRAM), a data splitting scheme is proposed as shown in Figure 3. In our architecture design, we use multiple crossbars to store multi-bit weights. For the input activation, due to the limited resolution of DACs, multiple cycles are needed for fetching these data.

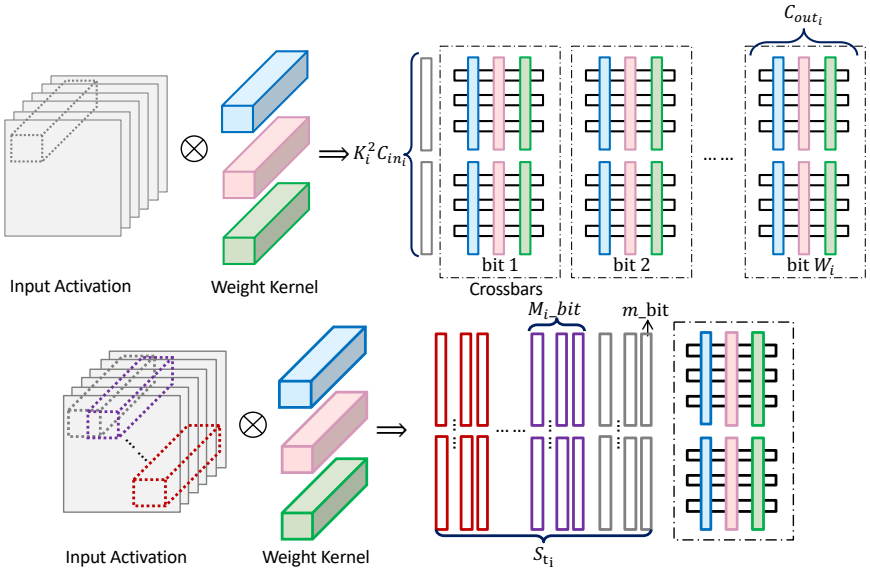

Figure 3. Data splitting scheme

The architecture is mainly composed of several **NVM banks**. In each **NVM bank**, an array of **NVM tiles** is organized and connected in a way similar to Network-on-Chip (NoC). To reduce the complexity of control logic and data path, we specify each tile will only process one layer of CNN, while for some large-scale layers, matrix splitting and multiple tiles will be needed. One **NVM tile** is adjacent to a **data forwarding unit**, which receives data from other tiles, merges (i.e., add or concatenate) them, and outputs the result to the local tile or other tiles. According to the layer type, i.e., CONV layers, pooling layers, or FC layers, the **NVM tile** can be configured as a pooling module or an MVM module, which are realized by **the pooling module** and **the crossbar process elements (PEs) array**. The **NVM PEs** in one tile are linked as an H-Tree structure to reduce the intra tile interconnection overhead. Each connection node

of the H-Tree is a **joint module**, which manages the data forwarding and summations of PE results. To solve the limited NVM device precision problem and to support multi-precision algorithms, multiple low precision **NVM crossbars** represent and store a part of high precision weight values. For example, eight 1-bit NVM crossbars are required for storing 8-bit CONV kernels. Computing results of different crossbars are merged together by shifter and adder tree.

4 ENTIRE MODELING FLOW

The entire modeling flow is shown in Figure 4. The input variables include specific weights & input feature (*.pth), CNN structure (/Interface/network.py), and architecture design (SimConfig.ini). The modeling process can be divided into two parts: accuracy simulation and hardware performance modeling.

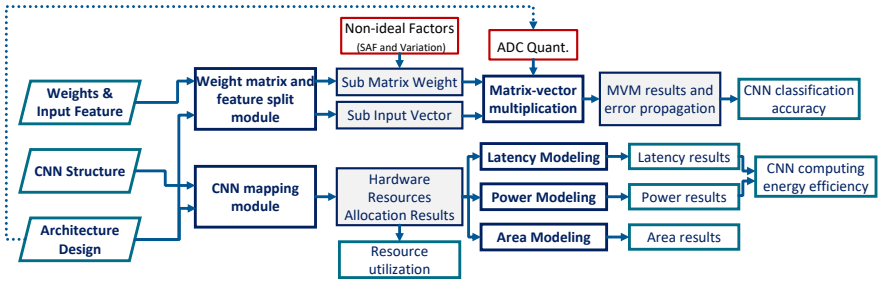

Figure 4. Entire modeling flow

Figure 5 illustrates the detailed accuracy evaluation process of PIM-based CNN computing accuracy, which contains five steps (/MNSIM_Python/MNSIM/Interface/). Firstly, considering crossbar size, NVM device precision, and DAC resolution, we split the weight matrix and feature data into sub-matrices and sub-vectors. Secondly, non- ideal factors are introduced to update the sub-matrix values. Here we only take Stuck- At-Faults (SAFs) and resistance variations into consideration, other non-ideal factors will be updated in the latter version. Thirdly, Matrix-Vector Multiplications (MVMs) are performed between updated sub-matrices and sub vectors. Fourthly, the MVMs results are quantized according to the ADC resolution. In MNSIM_Python, we provide two quantization modes:

1. Normal fixed quantization range: determine the quantization range according to the crossbar size $M \times N$, device precision (p_{NVM}), and DAC resolution(p_{DAC}):

$$[0, 2^{p_{DAC}+\log_2 M+p_{NVM}} - 1]$$

2. Dynamic quantization range: determine the quantization range according to the data distribution through NN training with training dataset. For this mode, please refer to our ASPDAC'20 paper for more information: *Hanbo Sun, Zhenhua Zhu, Yi Cai, Xiaoming Chen, Yu Wang, Huazhong Yang, An Energy-Efficient Quantized and Regularized Training Framework for Processing-In-Memory Accelerators, to appear in the 25th Asia and South Pacific Design Automation Conference (ASP-DAC 2020), 2020.*

Finally, the quantized MVM results are merged into the CONV results and propagated to the later layers to get the final classification accuracy.
The hardware modeling part is based on our previous work: *Lixue Xia, Boxun Li, Tianqi Tang, Peng Gu, Pai-yu Chen, Shimeng Yu, Yu Cao, Yu Wang, Yuan Xie, Huazhong Yang, MNSIM: Simulation Platform for Memristor-based Neuromorphic Computing System , in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), vol.37, No.5, 2018, pp.1009-1022.* Firstly, according to CNN structure and architecture design, the hardware resource usage and tile-level data dependency description are generated (/MNSIM_Python/ MNSIM/Mapping_Model/). Secondly, in terms of the mapping results, the power and area are modeled from the bottom level (e.g., device) to the top level (e.g., tile) (/MNSIM_Python/MNSIM/Hardware_Model/). Please note that the area and power results are based on the behavior-level modeling analysis. The parameters in the modules come from circuits-level simulation results, existing paper results, and other simulators (i.e., CACTI [4, 3]

and NVSIM [2]). Thirdly, computing latency is estimated w/ or w/o considering inner-layer pipeline (/MNSIM_Python/MNSIM/Latency_Model/). The inner-layer pipeline structure is discussed in our paper: *Tianqi Tang, Lixue Xia, Boxun Li, Yu Wang, Huazhong Yang, Binary Convolutional Neural Network on RRAM, in Proceedings of the 22nd Asia and South Pacific Design Automation Conference (ASP-DAC), 2017, pp.782-787.* Finally, the latency results and power results are used to calculate the computing energy efficiency.

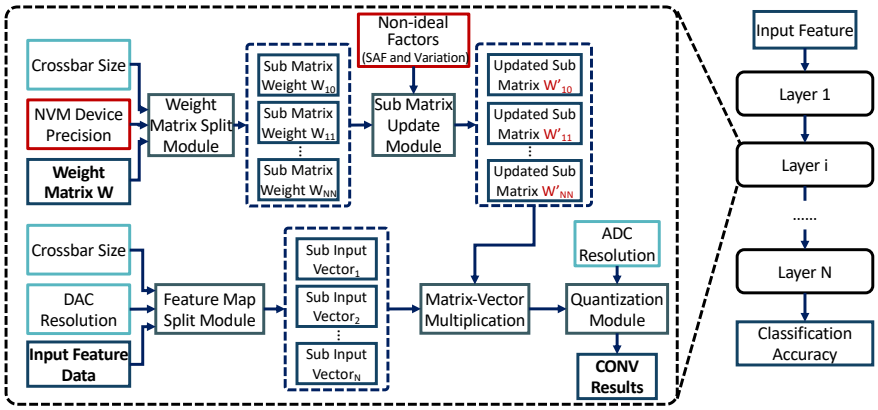

Figure 5. Accuracy evaluation of PIM-based CNN inference

5 FUTURE WORK AND UPDATE

There are still many incompleteness and imperfections in the current version of MNSIM_Python and we will continue to update and improve MNSIM_Python.

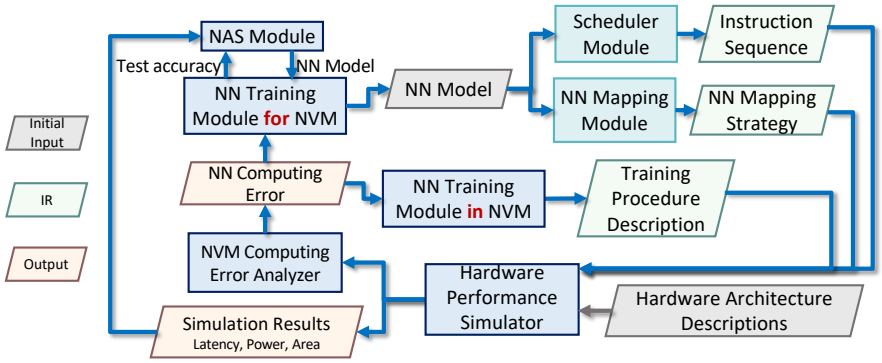

Figure 6. The completed version of MNSIM_Python

The completed version of MNSIM_Python we plan is shown in Figure 6. Compared with the current version, we will add a Network-Architecture-Search (NAS) module for PIM system to generate a "suitable" CNN structure for PIM and NN training module in PIM to model the on-line training architecture based on NVM [1].
Here is our update plan:
Recent updates:

1. Complement the missing digital module simulation data;
2. Update the buffer modeling and add more different buffer design options (e.g., NVM- based buffer design);
3. Design the network structure parameters automatic extraction module;
4. Optimize the modeling accuracy;

5. Support more kinds of non-ideal factors;
6. Add the NN training module for NVM-based PIM system.

Long term planning:

1. Add PIM-based on-line training module;
2. Add NAS module for PIM;
3. Design the interface between MNSIM_Python and other circuits-level simulators.

REFERENCES

[1] Cheng, M. et al. (2017). Time: A training-in-memory architecture for memristor-based deep neural networks. In DAC, 2017. ACM.

[2] Dong, X. et al. (2012). Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory. IEEE TCAD.

[3] Muralimanohar, N., Balasubramonian, R., and Jouppi, N. P. (2009). Cacti 6.0: A tool to model large caches. HP laboratories, 27:28.

[4] Wilton, S. J. E. and Jouppi, N. P. (1996). Cacti: an enhanced cache access and cycle time model. JSSC, 1996, 31(5):677–688.