A highly-parallel and energy-efficient 3D multi-layer CMOS-RRAM accelerator for tensorized neural network

Huang, Hantao; Ni, Leibin; Wang, Kanwen; Wang, Yuangang; Yu, Hao

2018

Huang, H., Ni, L., Wang, K., Wang, Y., & Yu, H. (2018). A highly-parallel and energy-efficient 3D multi-layer CMOS-RRAM accelerator for tensorized neural network. IEEE Transactions on Nanotechnology, 17(4), 645-656.

https://hdl.handle.net/10356/87049

https://doi.org/10.1109/TNANO.2017.2732698

A Highly-parallel and Energy-efficient 3D Multi-layer CMOS-RRAM Accelerator for Tensorized Neural Network

Hantao Huang, *Student Member, IEEE*, Leibin Ni *Student Member, IEEE*, Kanwen Wang, Yuangang Wang and Hao Yu, *Senior Member, IEEE*

***Abstract*—It is a grand challenge to develop highly-parallel yet energy-efficient machine learning hardware accelerator. This paper introduces a 3D multi-layer CMOS-RRAM accelerator for tensorized neural network (TNN). Highly parallel matrix-vector multiplication can be performed with low power in the proposed 3D multi-layer CMOS-RRAM accelerator. The adoption of tensorization can significantly compress the weight matrix of neural network using much fewer parameters. Simulation results using the benchmark MNIST show that the proposed accelerator has $1.283\times$ speed-up, $4.276\times$ energy-saving and $9.339\times$ area-saving compared to 3D CMOS-ASIC implementation; and $6.37\times$ speed-up and $2612\times$ energy-saving compared to 2D CPU implementation. In addition, $14.85\times$ model compression can be achieved by tensorization with acceptable accuracy loss.**

***Index Terms*—RRAM Computing, 3D Accelerator, Tensorized neural network.**

I. Introduction

Machine learning based big-data analytics has introduced great demand of highly-parallel yet energy-efficient hardware accelerators [1], [2], [3]. It is noticed that the main computation in a deep neural network involves intensive matrix-vector multiplications. The GPU-based acceleration can achieve the highest parallelism but with huge power overhead [4]. The low-power FPGA-based acceleration on the other hand cannot achieve high throughput due to limited computation resource (processing element and memory) [5]. The major recent attention is to develop 2D CMOS-ASIC accelerators [6]. However, these traditional accelerators are both in a 2D out-of-memory architecture with low bandwidth at I/O and high leakage power consumption from the CMOS SRAM memory [4].

From supporting hardware perspective, the recent in-memory resistive random access memory (RRAM) devices [4], [7], [8], [9] have shown great potential for an energy-efficient acceleration of multiplication on crossbar. It can be exploited as both storage and computational elements with minimized leakage power due to its non-volatility. Recent researches in [10], [11] show that the 3D heterogeneous integration can further support more parallelism with high I/O bandwidth in acceleration by stacking RRAM on CMOS using through-silicon-vias (TSVs).

From computing algorithm perspective, network compression is required to enable a successful mapping of a simplified neural network to the supporting hardware for machine learning. [12], [13] proposed to use low-precision numerical value to represent weights. [14], [15] used low-rank approximation directly to the weight matrix. Such over-simplified approximated computing can simply reduce complexity but cannot maintain the accuracy.

In this paper, we propose a tensorized neural network (TNN) obtained during training with significant compression. By representing dense data in high dimensional space with sparsity, significant network compression can be achieved. More importantly, we introduce an accordingly 3D multi-layer CMOS-RRAM accelerator to support such TNN-based machine learning with high parallelism yet low power. By buffering input data on the first RRAM layer, intensive matrix-vector multiplication are efficiently performed on the second RRAM layer. One more layer of CMOS is further utilized for the data control and synchronization. Experiment results using the benchmark of MNIST show that the proposed accelerator has $1.283\times$ speed-up, $4.276\times$ energy-saving and $9.339\times$ area-saving compared to 3D CMOS-ASIC implementation; and $6.37\times$ speed-up and $2612\times$ energy-saving compared to 2D CPU implementation. In addition, $14.85\times$ times model compression can be achieved with acceptable accuracy loss by the TNN.

The rest of this paper is organized as follows. The tensorized neural network and its training process are discussed in Section II. The 3D multilayer CMOS-RRAM accelerator architecture is discussed in Section III. Section IV shows the detailed accelerator mapping on the 3D RRAM-crossbar and CMOS respectively. Experiment results are presented in Section V with conclusion drawn in Section VI.

II. Tensorized Neural Network

Previous neural network compression is simply performed by either precision-bit truncation or low-rank approximation [12], [13], [14], [15], which cannot maintain good balance

Manuscript received Jan 13, 2017; revised Jul 03, 2017. The review of this paper was arranged and approved by Editor Fabrizio and Editor-in-Chief Fabrizio.

H. Huang, and L. Ni are with School of Electrical and Electronic Engineering, Nanyang Technological University (NTU), Singapore. H. Yu is with Southern University of Science and Technology, China (yuh3@sustc.edu.cn). K. Wang and Y. Wang are both with Data Center Technology Laboratory, 2012 Labs, Huawei Technologies Co., Ltd, China

Acknowledgement: this work is sponsored by grants from Singapore NRF-CRP (NRF-CRP9-2011-01, NRF-CRP16-2015-03) and MOE Tier-2 (MOE2015-T2-2-013).

TABLE I
SYMBOL NOTATIONS AND DETAILED DESCRIPTIONS.

Notations	Descriptions
$\boldsymbol{\mathcal{V}} \in \mathbb{R}^{n_1 \times n_2 \times ... \times n_d}$	d-dimensional tensor of size $n_1 \times n_2 ... n_d$
$\boldsymbol{G}_i \in \mathbb{R}^{r_{i-1} \times n_i \times r_i}$	Tensor cores of tensor-train data format
$\boldsymbol{W}_1, \boldsymbol{W}_2, ..., \boldsymbol{W}_d$	Neural network weights of d layers
$\boldsymbol{B}_1, \boldsymbol{B}_2, ..., \boldsymbol{B}_d$	Neural network bias of d layers
$\boldsymbol{H}$	Activation matrix
$\boldsymbol{T}, \boldsymbol{Y}$	Labels and neural network output
$\boldsymbol{U}, \boldsymbol{S}, \boldsymbol{V}$	SVD decomposition matrices
$\boldsymbol{X}$, $\boldsymbol{X_t}$	Input features, testing input features
$r_0, r_1, ..., r_d$	Rank of tensor cores
$L_1, L_2, ..., L_d$	Dimension of d-layer neural network weight
$l_{k,0}, l_{k,1}, ..., l_{k,m}$	Factorized L_k, $L_k = l_{k,0} \times l_{k,1}, ..., \times l_{k,m}$
$i_1, i_2, ..., n_d$	Index vectors referring to a tensor element
$n_1, n_2, ..., n_d$	Mode size of tensor $\boldsymbol{\mathcal{V}} \in \mathbb{R}^{n_1 \times n_2 \times ... \times n_d}$
n_m	Maximum mode size of $n_1, n_2, ..n_d$
N_t	Number of training samples
N, L_0	Number of input features
M	Number of classes
V_w, V_r, V_{th}	RRAM write, read and threshold voltage
$c_{i,j}$	Configurable conductance of RRAM-crossbar

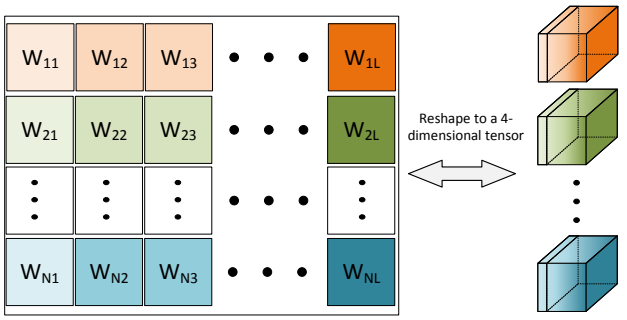

Fig. 1. Block matrices and their representation by a 4-dimensional tensor

between network compression and network accuracy. By representing dense data in high dimensional space with natural sparsity, tensorized data formatting can significantly compress the neural network complexity without much accuracy loss [16]. In this section, we discuss a tensor-train [17] formatted neural network during the training. The tensor-train based decomposition and compression will be first introduced. Then, a tensorized neural network (TNN) will be discussed based on the extension of general neural network. Finally, a layer-wise training of TNN using modified alternating least-squares method is further proposed.

A. Tensor-train Decomposition and Compression

Tensors are natural multi-dimensional generation of matrices. Here, we refer one-dimensional data as *vectors*, denoted as $\boldsymbol{v}$. Two-dimensional arrays are *matrices*, denoted as $\boldsymbol{V}$ and higher dimensional arrays are *tensors* denoted as $\boldsymbol{\mathcal{V}}$. To refer one specific element from a tensor, we use calligraphic upper letters $\boldsymbol{\mathcal{V}}(\boldsymbol{i}) = \boldsymbol{\mathcal{V}}(i_1, i_2, ...i_d)$, where d is the dimensionality of the tensor $\boldsymbol{\mathcal{V}}$. We can effectively reshape a 2-dimensional matrix into a 4-dimensional tensor as shown in Fig. 1.

A d-dimensional $n_1 \times n_2 \times ... \times n_d$ tensor $\boldsymbol{\mathcal{V}}$ is decomposed into the tensor-train data format if tensor core $\boldsymbol{G}_k$ is defined as $r_{k-1} \times n_k \times r_k$ and each element is defined [17] as

$$\boldsymbol{\mathcal{V}}(i_1, i_2, ...i_d) = \sum_{\alpha_0, \alpha_1 ... \alpha_d}^{r_0, r_1, ... r_d} \boldsymbol{G}_1(\alpha_0, i_1, \alpha_1) \boldsymbol{G}_2(\alpha_1, i_2, \alpha_2) ... \boldsymbol{G}_d(\alpha_{d-1}, i_d, \alpha_d) \tag{1}$$

where α_k is the index of summation, which starts from 1 and stops at rank r_k. $r_0 = r_d = 1$ is for the boundary condition and $n_1, n_2, ...n_d$ are known as mode size. Here, r_k is the core rank and $\boldsymbol{G}$ is the core for this tensor decomposition. By using the notation of $\boldsymbol{G}_k(i_k) \in \mathbb{R}^{r_{k-1} \times r_k}$, we can rewrite the above equation in a more compact way:

$$\boldsymbol{\mathcal{V}}(i_1, i_2, ...i_d) = \boldsymbol{G}_1(i_1) \boldsymbol{G}_2(i_2) ... \boldsymbol{G}_d(i_d) \tag{2}$$

where $\boldsymbol{G}_k(i_k)$ is an $r_{k-1} \times r_k$ matrix, a slice from the 3-dimensional matrix $\boldsymbol{G}_k$. The symbol notations and detailed description are shown in Table I.

Such a representation is memory-efficient to store high-dimensional data and hence with significant energy saving as well. For example, a d-dimensional tensor requires $N = n_1 \times n_2 \times ... \times n_d = n^d$ number of parameters. However, if it is represented using the tensor-train format, it takes only $\sum_{k=1}^{d} n_k r_{k-1} r_k$ parameters. Here, we define a tensorized neural network (TNN) if the weight of the neural network can be represented in the tensor-train data format. For example, a two-dimensional weight $\boldsymbol{W} \in \mathbb{R}^{L_0 \times L_1}$ can be reshaped to a $k_1 + k_2$ dimensional tensor $\boldsymbol{\mathcal{W}} \in \mathbb{R}^{l_{0,1} \times l_{0,2} \times ... l_{0,k_1} \times l_{1,1} \times l_{1,2} \times ... l_{1,k_2}}$ by factorizing $L_0 = \prod_{m=1}^{k_1} l_{0,m}$ and $L_1 = \prod_{m=1}^{k_2} l_{1,m}$ and such tensor can be decomposed into the tensor-train data format to save storages [1].

B. Tensor-train based Neural Network (TNN)

To make TNN clear, we first start with a general feed forward neural network and then extend it to the tensor-train based neural network. We use a single hidden layer neural network as an example and the same principle can be applied to the multi-layer neural network [19], [20], [21]. Generally, we can train a neural network based on data features $\boldsymbol{X}$ and labels $\boldsymbol{T}$ with N_t number of training samples, N dimensional input features and M classes. During the training, one needs to minimize the error function with determined weights: $\boldsymbol{W}_1$ (at input layer) and $\boldsymbol{W}_2$ (at output layer) for a single hidden layer neural network[2]:

$$E = ||\boldsymbol{T} - f(\boldsymbol{W}_1, \boldsymbol{W}_2, \boldsymbol{X})||_2 \tag{3}$$

where $f(\cdot)$ is the trained model to perform the predictions from input.

Here, we mainly discuss the inference (testing) process and leave the training process to the next section. The output of each layer is based on matrix multiplication and activation. For example, the first layer output $\boldsymbol{H}$ is

$$\boldsymbol{preH} = \boldsymbol{X_t} \boldsymbol{W}_1 + \boldsymbol{B}_1, \ \boldsymbol{H} = \frac{1}{1 + e^{-\boldsymbol{preH}}} \tag{4}$$

[1]Interested readers can also refer to [17], [18] for more details on the tensor-train data format.

[2]We ignore bias for a clear explanation.

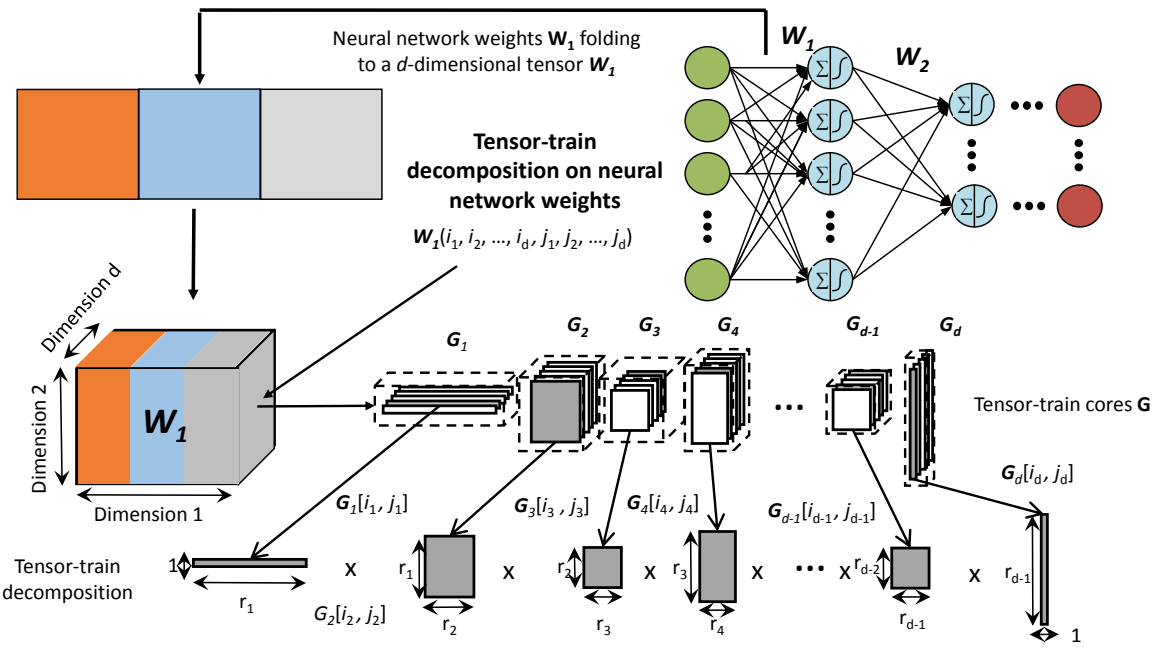

Fig. 2. Neural network weight tensorization and represented by tensor-train data format for parameter compression (from n^d to dnr^2)

where $\boldsymbol{X}_t$ is the testing data. $\boldsymbol{W}_1 \in \mathbb{R}^{N\times L_1}$ and $\boldsymbol{B}_1 \in \mathbb{R}^{N_t\times L_1}$ are the input weights and bias respectively. Then the neural network output for a single hidden layer neural network is

$$\begin{aligned} \boldsymbol{Y} &= f(\boldsymbol{W}_1, \boldsymbol{W}_2, \boldsymbol{X}_t) \\ p(i/y_i) &\approx y_i,\ \ y_i \in \boldsymbol{Y} \end{aligned} \tag{5}$$

where i represents class index $i \in [1,\ M]$. We approximate the prediction probability for each class by the output of neural network.

For the tensor-train based neural network, Fig. 2 shows the general idea. A two-dimensional weight is folded into a three-dimensional tensor and then decomposes into tensor cores $\boldsymbol{G}_1, \boldsymbol{G}_2, ...\boldsymbol{G}_d$. These tensor cores are relative small matrices due to the small value of rank r leading to a high neural network compression rate. Then the whole neural network will be trained in the tensor-train data format.

The TNN inference is a directly application of the tensor-train-matrix-by-vector operations [16], [17]. We will use $\boldsymbol{W} \in \mathbb{R}^{N\times L}$ to discuss the forward pass of neural network. Firstly, we rearrange $\boldsymbol{W}$ to a d-dimensional tensor $\boldsymbol{\mathcal{W}}$ whose k_{th} dimension is a vector of length $n_k l_k$. Here, we define n_k and l_k as $N = \prod_{k=1}^{d} n_k$ and $L = \prod_{k=1}^{d} l_k$. Without consideration of the bias $\boldsymbol{B}$ and activation function, the neural network forward pass $\boldsymbol{H} = \boldsymbol{XW}$ in the tensor-train data format is

$$\mathcal{H}(\boldsymbol{i}) = \sum_{\boldsymbol{j}=[j_1,j_2,...j_d]}^{l_1,l_2,l_3,...,l_d} \mathcal{X}(\boldsymbol{j})\boldsymbol{G}_1[i_1,j_1]\boldsymbol{G}_2[i_2,j_2]...\boldsymbol{G}_d[i_d,j_d] \tag{6}$$

where $\boldsymbol{i} = i_1, i_2, ..., i_d, i_k \in [1, n_k]$, $\boldsymbol{j} = j_1, j_2, ..., j_d, j_k \in [1, l_k]$ and $\boldsymbol{G}_k[i_d, j_d] \in \mathbb{R}^{r_{k-1}\times r_k}$ is a slice of cores. We use a pair $[i_k, j_k]$ to refer an index of vector $[1\ \ n_k l_k]$, where $\boldsymbol{G}_k \in \mathbb{R}^{r_{k-1}\times n_k l_k \times r_k}$. Since the fully-connected layer is a special case of convolutional layer with kernel size 1×1, such tensorized weights can also be applied to other convolutional layers.

This tensor-train-matrix-by-vector multiplication complexity is $O(dr^2 n_m\ max(N, L))$ [16], where r is the maximum rank of cores $\boldsymbol{G_i}$ and n_m is the maximum mode size $m_k n_k$ of tensor $\boldsymbol{\mathcal{W}}$. This can be very efficient if the rank r is very small compared to general matrix-vector multiplication. It is also favorable for distributed computation on RRAM devices since each core is small and matrix multiplication is associative.

C. Training on TNN

Tensor-train based neural network is first proposed by [16] but its training complexity significantly increases due to the backwards propagation under the tensor-train data format. A layer-wise training provides good performance with reduced epoch number of backward propagation leading to a significant training time reduction [20], [22], [21]. Moreover, to perform a successful mapping of TNN, recursively training of TNN is required for the trade-off of accuracy and compression rate. Thereby, a fast layer-wise training method is developed in this paper for TNN.

The training process of TNN is the same as general neural network layer-wise training but with the tensor-train data format. We first discuss the general training process following the training framework form [23] and then extend it to TNN. Given a single hidden layer with random generated input weight, the training process is to minimize:

$$min.\ ||\boldsymbol{HW}_2 - \boldsymbol{T}||_2 + \lambda||\boldsymbol{W}_2||_2 \tag{7}$$

where $\boldsymbol{H}$ is the hidden-layer output matrix generated from the Sigmoid function for activation; and λ is a user defined parameter that biases the training error and output weights

Algorithm 1 Layer-wise Training of Neural Network with Modified Least Squares Solver

Input: Input Set $(\boldsymbol{X}, \boldsymbol{T})$, $\boldsymbol{X}$ is the input data and $\boldsymbol{T}$ is the desired output depending on the layer architecture, activation function $G(a_i, b_i, x_j)$, number of hidden neuron node $L_0, L_1, ..., L_d$

Output: Neural Network Weight $\boldsymbol{W}_1, \boldsymbol{W}_2, ...\boldsymbol{W}_d$ for d-layer neural network

1: **for** $i = 1 : d$ **do** ▷ *d layer neural network*
2: **if** $i == 1$ **then** ▷ *Random generated weights*
3: Factorize L_0 and L_1 (e.g. $L_0 = l_{0,1} \times l_{0,2}$ and $L_1 = l_{1,1} \times l_{1,2}$)
4: Randomly generate tensor cores $\boldsymbol{G}_1 \in \mathbb{R}^{r_1 \times n_1 \times r_2}$, $\boldsymbol{G}_2 \in \mathbb{R}^{r_2 \times l_1 \times r_3}$ and other tensor cores to represent a tensor $\boldsymbol{\mathcal{W}}_1 \in \mathbb{R}^{n_1 \times n_2 \times l_1 \times l_2}$
5: **else** ▷ *Layer-wise training*
6: Random generated $\boldsymbol{\mathcal{W}}_i$ following Step 3, 4
7: Perform tensor-train-matrix-by-vector multiplication based on (6), which equivalents to $\boldsymbol{preH}_i = \boldsymbol{H}_{i-1}\boldsymbol{W}_{i-1}$
8: Perform acitvation function which equivalents to $\boldsymbol{H}_i = 1/(1 + e_i^{-\boldsymbol{preH}})$.
9: Compute $\boldsymbol{\mathcal{W}}_i$ using the modified alternating least-squares $||\boldsymbol{H}_i\boldsymbol{\mathcal{W}}_i - \boldsymbol{P}||_2$
10: Note: For auto-encoder layers, $\boldsymbol{P}$ is the activation matrix $\boldsymbol{H}_{i-1}$ ($\boldsymbol{H}_0 = \boldsymbol{X}$). For the decision layer, $\boldsymbol{P}$ is the label matrix $\boldsymbol{T}$.
11: **end if**
12: **end for**

[23]. The output weight $\boldsymbol{W}_2$ is computed based on least-squares problem:

$$\begin{aligned} &\boldsymbol{W}_2 = (\tilde{\boldsymbol{H}}^T\tilde{\boldsymbol{H}})^{-1}\tilde{\boldsymbol{H}}^T\tilde{\boldsymbol{T}},\ \tilde{\boldsymbol{H}} \in \mathbb{R}^{N_t \times L} \\ &where\ \tilde{\boldsymbol{H}} = \begin{pmatrix} \boldsymbol{H} \\ \sqrt{\lambda}\boldsymbol{I} \end{pmatrix}\ \ \tilde{\boldsymbol{T}} = \begin{pmatrix} \boldsymbol{T} \\ \boldsymbol{0} \end{pmatrix} \end{aligned} \tag{8}$$

where $\tilde{\boldsymbol{T}} \in \mathbb{R}^{(N_t+L_1)\times M}$ and M is the number of classes. $\boldsymbol{I} \in \mathbb{R}^{L_1 \times L_1}$ and $\tilde{\boldsymbol{H}} \in \mathbb{R}^{(N_t+L_1)\times L_1}$.

To build a multi-layer neural network, backwards propagations [19] or layer-wise training using the auto-encoder method [20], [21] can be applied. An auto-encoder layer is to set the single layer output $\mathbf{T}$ the same as input $\mathbf{X}$ and find an optimal weight to represent itself. By stacking auto-encoder layers on the finial decision layer, we can build the multi-layer neural network. Algorithm 1 summarizes the layer-wise training with modified alternating least-squares method.

As discussed in the general neural network, the training of TNN requires to solve a least-squares problem in the tensor-train data format. For the output weight $\boldsymbol{W}_2$ in (7), we propose a tensor-train based least-squares training method using modified alternating least squares algorithm (also known as density matrix renormalization group in quantum dynamics) [24], [25]. The modified alternating least squares (MALS) for minimization of $||\boldsymbol{H}\boldsymbol{\mathcal{W}}_2 - \boldsymbol{T}||_2$ is working as below.

1) **Initialization**: Randomly initialized cores $\boldsymbol{G}$ and set $\boldsymbol{\mathcal{W}}_2 = \boldsymbol{G}_1 \times \boldsymbol{G}_2 \times ... \times \boldsymbol{G}_d$. The process is the same as Step 3, 4 in Algorithm 1.
2) **Sweep of Cores**: core $\boldsymbol{G}_k$ is optimized with other cores fixed. Left-to-right sweep from $k = 1$ to $k = d$
3) **Supercore generated**: Create supercore $\boldsymbol{X}(k, k+1) = \boldsymbol{G}_k \times \boldsymbol{G}_{k+1}$ and find it by minimizing of least-squares problem $||\boldsymbol{H} \times \boldsymbol{Q}_{k-1} \times \boldsymbol{X}_{k,k+1} \times \boldsymbol{R}_{k+2} - \boldsymbol{T}||_2$, reshape $\boldsymbol{Q}_{k-1} = \prod_{i=1}^{k-1}\boldsymbol{G}_i$ and $\boldsymbol{R}_{k+2} = \prod_{i=k+2}^{d}\boldsymbol{G}_i$ to fit matrix-matrix multiplication
4) **Split supercore**: SVD $\boldsymbol{X}(k, k+1) = \boldsymbol{U}\boldsymbol{S}\boldsymbol{V}^T$, let $\boldsymbol{G}_k = \boldsymbol{U}$ and $\boldsymbol{G}_{k+1} = \boldsymbol{S}\boldsymbol{V}^T \times \boldsymbol{G}_{k+1}$. $\boldsymbol{G}_k$ is determined and $\boldsymbol{G}_{k+1}$ is updated. Truncated SVD can also be performed by removing smaller singular values to reduce ranks.
5) **Sweep Termination**: Terminate if maximum sweep times reached or error is smaller than required.

The low rank initialization is very important to have smaller rank r for each core. Each supercore generation is the process of solving least-squares problems. The complexity of least-squares for $\boldsymbol{X}$ are $O(n_m R r^3 + n_m^2 R^2 r^2)$ [25] and the SVD compression requires $O(n_m r^3)$, where R, r and n_m are the rank of activation matrix $\boldsymbol{H}_1$, the maximum rank of core $\boldsymbol{G}$ and maximum mode size of $\boldsymbol{\mathcal{W}}_2$ respectively. By using truncated SVD, we can adaptively reduce the rank of each core to reduce the computation complexity and save memory storage.

Such tensorization can benefit of implementing large neural networks. Firstly, by performing tensorization, the size of neural network can be compressed. Moreover, the computation load can also be reduced by adopting small tensor ranks. Secondly, a tensorzaiton of weight matrix can decompose the big matrix into many small tensor-core matrices, which can effectively reduce the configuration time of RRAM. Lastly, the multiplication of small matrix can be performed in a highly parallel fashion on RRAM to speed-up the large neural network processing time.

III. 3D Multi-layer CMOS-RRAM Accelerator

In this section, we introduce RRAM-crossbar devices, which can be used for both storage and computation. Furthermore, the 3D hardware platform is proposed based on the non-volatile RRAM-crossbar devices with the design flow for TNN mapping on the proposed architecture.

A. RRAM-Crossbar Device

Emerging resistive random access memory (RRAM) [26], [27] is a two-terminal device with 2 non-volatile states: high resistance state (HRS) and low resistance state (LRS). The state of RRAM is determined when a write voltage V_w is applied to its two terminals. It is most stable in bistate, where high resistance state (RHS) R_{off} and low resistance state (LHS) R_{on} are determined by the polarity of write voltage. As RRAM states are sensible to the input voltages, special care needs to be taken while reading, as such read voltage V_r is less than half of write voltage V_w, given as (9). V_w and V_r are related as follows

$$V_w > V_{th} > V_w/2 > V_r, \tag{9}$$

where V_{th} is the threshold voltage of RRAM.

In one RRAM-crossbar, given the input probing voltage, the current on each bit-line (BL) is the multiplication-accumulation of current through each RRAM device on the BL. Therefore, the RRAM-crossbar array can intrinsically perform the analog matrix-vector multiplication [28]. Given an

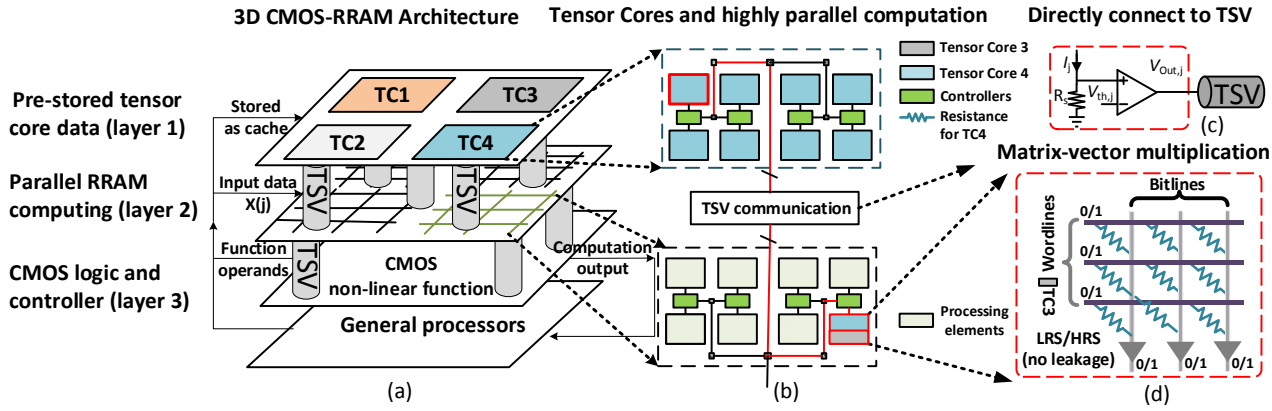

Fig. 3. (a) Proposed 3D multi-layer CMOS-RRAM accelerator (b) RRAM memory and highly parallel RRAM based computation engine (c) TSV communication (d) Memristor Crossbar

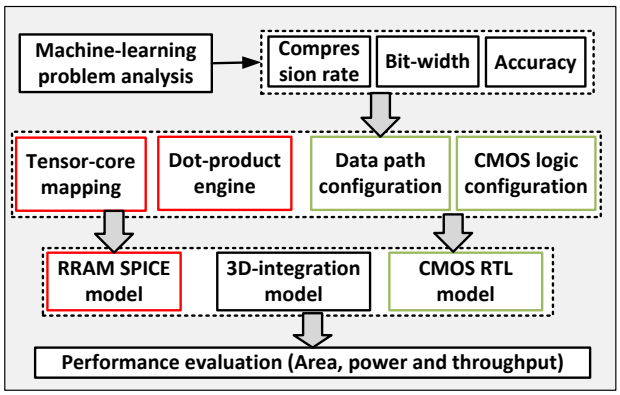

Fig. 4. Mapping flow for tensor-train based neural network (TNN) on the proposed architecture

input voltage vector $V_{WL} \in \mathbb{R}^{N\times 1}$, the output voltage vector $V_{BL} \in \mathbb{R}^{N\times 1}$ can be expressed as

$$\begin{bmatrix} V_{BL,1} \\ \vdots \\ V_{BL,M} \end{bmatrix} = \begin{bmatrix} c_{1,1} & \cdots & c_{1,M} \\ \vdots & \ddots & \vdots \\ c_{N,1} & \cdots & c_{N,M} \end{bmatrix} \begin{bmatrix} V_{WL,1} \\ \vdots \\ V_{WL,N} \end{bmatrix} \quad (10)$$

where $c_{i,j}$ is configurable conductance of the RRAM resistance $R_{i,j}$, which can represent a real number of weight. Compared to traditional CMOS implementation, RRAM-crossbar achieves better parallelism and consumes less power. However, note that analog implementation of matrix-vector multiplication is strongly affected by non-uniform resistance values [4]. As such, one needs to develop a digital fashioned multiplication based on the RRAM-crossbar instead. Therefore, a digital-fashioned multiplication on RRAM-crossbar is preferred to minimize the device non-uniform impact from process variation [29].

B. 3D Multi-layer CMOS-RRAM Architecture

3D-integration: Recent works [30], [31] show that the 3D integration supports heterogeneous stacking because different types of components can be fabricated separately with different technologies and then layers can be stacked into 3D structure. Therefore, stacking non-volatile memories on top of micro-processors enables cost-effective heterogeneous integration. Furthermore, works in [32], [33], [34] also show the feasibility to stack RRAM on CMOS to achieve smaller area and lower energy consumption.

3D-stacked Modeling: In this proposed accelerator, we adopt the face-to-back bonding with TSV connections. TSVs can be placed vertically on the whole layer as shown in Fig. 3. The granularity at which TSV can be placed is modeled based on CACTI-3DD using the fine-grained strategy [35], which will automatically partition the memory array to utilize TSV bandwidth. Although this strategy requires a large number of TSV, it provides higher bandwidth and better access latency, which are greatly needed to perform highly-parallel tensor based computation. We use this model to evaluate our proposed architecture and will show the bandwidth improvement in Section V-B.

Architecture: In this paper, we propose a 3D multi-layer CMOS-RRAM accelerator with three layers as shown in Fig. 3. This accelerator is composed of a two-layer RRAM-crossbar and a one-layer CMOS circuit. More specifically, they are designed as follows.

- Layer 1 of RRAM-crossbar is implemented as a buffer to store neural network model weights as Fig. 3(a) shows. The tensor cores are 3-dimensional matrices and each slice is a 2-dimensional matrix stored distributively in a H-tree like fashion on the Layer 1 as described in Fig. 3(b). They can be accessed through TSV as the input of the RRAM-crossbar or used to configure the RRAM-crossbar resistance in Layer 2.
- Layer 2 of RRAM-crossbar performs logic operations such as matrix-vector multiplication and also vector addition. As shown in Fig. 3(b), Layer 2 collects tensor cores from Layer 1 through TSV communication to perform parallel matrix-vector multiplication. The RRAM data is directly sent through TSV. The wordline takes the input (in this case, tensor core 3) and the multiplicand (in this case, tensor core 4) is stored as the conductance of RRAM. The output will be collected from the bitlines as shown in Fig. 3(d).

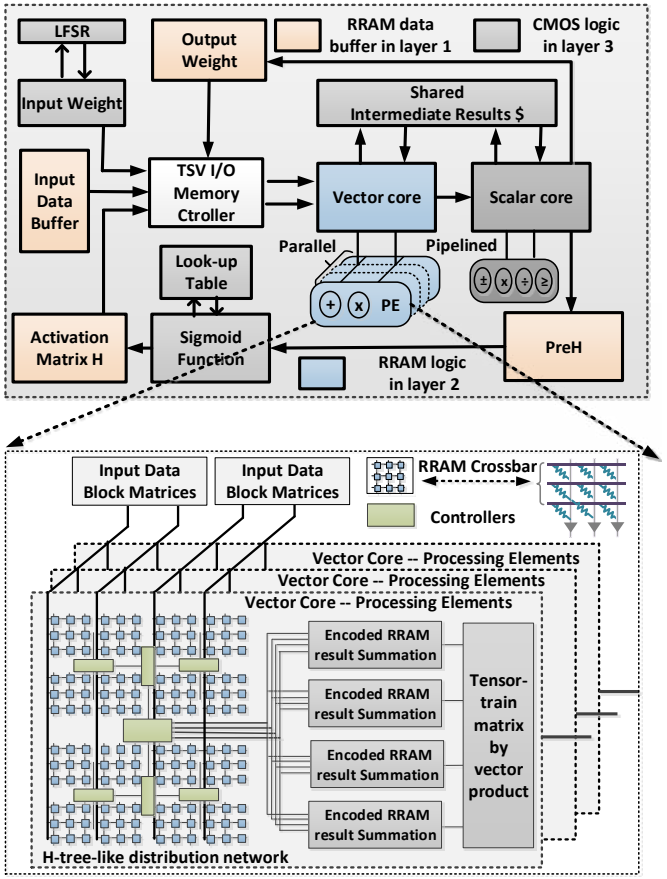

Fig. 5. Data control and synchronization on layer of CMOS with highly-parallel RRAM based processing elements

- Layer 3 is designed to perform the overall synchronization of the tensorized neural network. It will generate the correct tensor core index as described in (6) to initiate tensor-train matrix multiplication. In addition, the CMOS layer will also perform the non-linear mapping.

Note that buffers are designed to separate resistive networks between Layer 1 and Layer 2. The last layer of CMOS contains read-out circuits for RRAM-crossbar and performs logic control for neural network synchronization.

Mapping Flow: Fig. 4 shows the working flow for the tensor-train based neural network mapping on the proposed architecture. Firstly, the algorithm optimization targeting to the specific application is performed. The neural network compression is performed through layer-wise training process. Then, the design space between compression rate, bit-width and accuracy is explored to determine the optimal neural network configuration (such as number of layers and activation function). Secondly, the architecture level optimization is performed. The RRAM buffer on Layer 1 and the computing elements on Layer 2 are designed to minimize the read access latency and power consumption. Furthermore, the CMOS logic is designed based on finite state machine for neural network synchronization. Finally, the whole system is evaluated based on the RRAM SPICE model, CMOS RTL Verilog model and 3D-integration model to determine the system performance.

IV. TNN Accelerator Design on 3D CMOS-RRAM Architecture

In this section, we further discuss how to utilize the proposed 3D multi-layer CMOS-RRAM architecture to design the TNN accelerator. We first discuss the CMOS layer design, which performs the high level control of TNN computation. Then a highly-parallel RRAM based accelerator is introduced with the TNN accelerator and dot-product engine.

A. CMOS Layer Accelerator

To fully map TNN on the proposed 3D multi-layer CMOS-RRAM accelerator, the CMOS logic is designed mainly for logic control and synchronization using top-level state machine. It prepares the input data for computing cores, monitors the states of RRAM logic computation and determines the computation layer of neural network. Fig. 5 shows the detailed mapping of the tensorized neural network (TNN) on the proposed 3D multi-layer CMOS-RRAM accelerator. This is a folded architecture by utilizing the sequential operation of each layer on the neural network. The testing data will be collected from RRAM memory through TSV and then sent into vector core to perform matrix-vector multiplication through highly parallel processing elements in the RRAM layer. The RRAM layer has many distributed RRAM-crossbar structures to perform multiplication in parallel. Then the computed output from RRAM will be transfered to scalar score to perform accumulations. The scaler core can perform addition, subtraction and comparisons. Then the output from the scalar core will be sent to the sigmoid function model for activation in a pipelined fashion, which performs the computation of (4). The activation matrix $\boldsymbol{H}$ will be used for the next layer computation. As a result, the whole TNN inference process can be mapped to the proposed 3D multi-layer CMOS-RRAM accelerator.

In addition, to support TNN on RRAM computation, a dedicated index look-up table is formed. Since the weight matrix is actually folded into a high dimensional tensor as shown in Fig. 1, a correct index selection function called bijective function is designed. The bijective function for weight matrix index is also performed by the CMOS layer. Based on the top state diagram, it will choose the correct slice of tensor core $\boldsymbol{G}_i[i, j]$

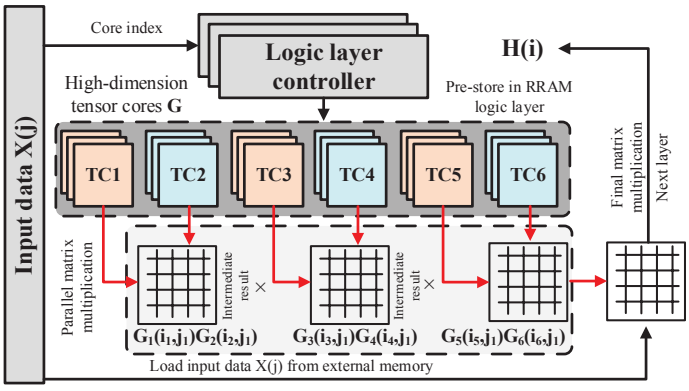

Fig. 6. RRAM based TNN accelerator for highly parallel computation on tensor cores

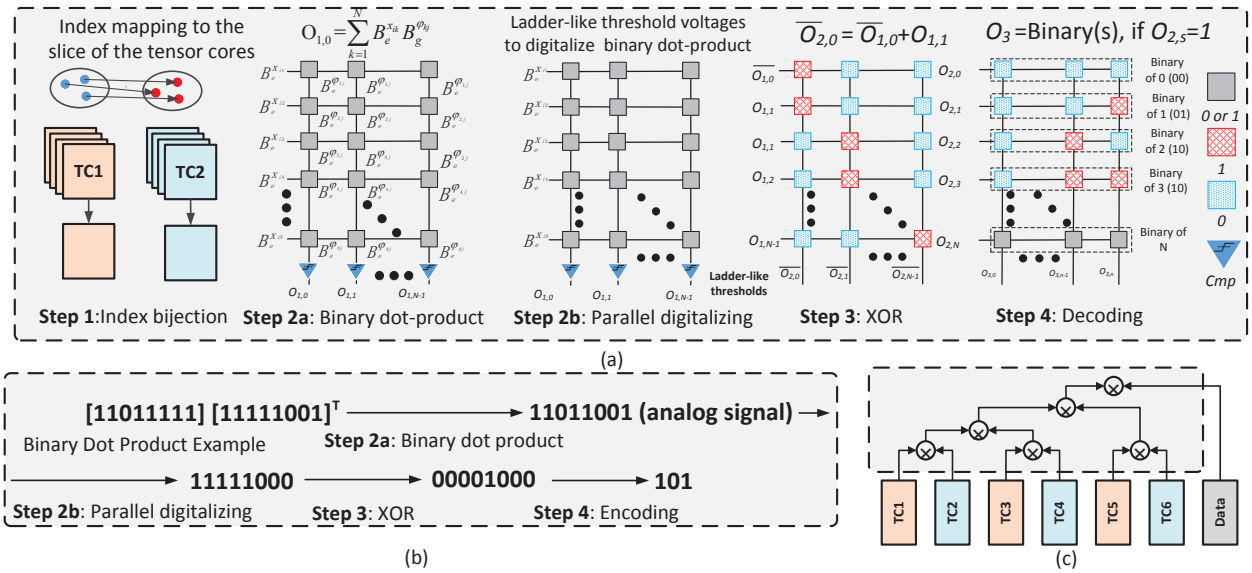

Fig. 7. (a) RRAM based dot-product engine for TNN (b) an example of dot-product overall flow (c) tree-based parallel tensor cores multiplication accelerator

by determining the i, j index. Then the RRAM-crossbar will be configured to perform matrix-vector multiplication.

B. RRAM Layer Accelerator

In the RRAM layer, we design the RRAM layer accelerator for highly-parallel computation using single instruction multiple data (SIMD) method to support data parallelism.

1) Highly-parallel TNN Accelerator on the RRAM Layer: The TNN accelerator is designed to support highly parallel tensor-train-matrix-by-vector multiplication by utilizing the associative principle of matrix product. According to (6), $\mathcal{X}(\mathbf{i})$ needs to be multiplied by d matrices unlike the general neural network. As a result, if traditional matrix-vector multiplication in serial is applied, data needs to be stored in the RRAM array for d times, which is time-consuming. Since the size of tensor cores in the TNN is much smaller than the weights in the general neural network, multiple matrix-vector multiplication engines can be placed in the RRAM logic layer. When then input data is loaded, the index of $\boldsymbol{G}_i$ can be known. For example, we need compute $\boldsymbol{X}(\boldsymbol{j})\boldsymbol{G}_1[i_1, j_1]\boldsymbol{G}_2[i_2, j_1]\boldsymbol{G}_3[i_3, j_1]\boldsymbol{G}_i[i_4, j_1]$ given $d = 4$ for the summation in (6). $\boldsymbol{G}_1[i_1, j_1]\boldsymbol{G}_2[i_2, j_1]$ and $\boldsymbol{G}_3[i_3, j_1]\boldsymbol{G}_i[i_4, j_1]$ in (6) can be pre-computed in a parallel fashion before the input data $\mathcal{X}(\mathbf{i})$ is loaded.

Fig. 6 gives an example of parallel tensor core multiplications. The tensor cores (TC1-6) are firstly stored in the RRAM layer. When the input data $\mathcal{X}(\mathbf{j})$ comes, the index of each tensor core is loaded by the logic layer controllers first. The controller will configure the RRAM conductance to write the according data from the tensor cores to RRAM cells. As shown in the Fig. 6, tensor cores (TC2 TC4 and TC6) are used to configure the RRAM to write the data and tensor cores (TC1 TC3 and TC5) are selected for the RRAM input for the multiplication. Please note that the matrix-vector multiplication of $\boldsymbol{G}_i$ can be performed in parallel to calculate the intermediate matrices while $\mathcal{X}(\mathbf{i})$ is in the loading process. After all the intermediate results are ready, they can be multiplied by $\mathcal{X}(\mathbf{i})$ so that the operation will be efficient and not affected by the input data $\mathcal{X}(\mathbf{i})$ loading process.

2) Highly-parallel Dot-product Engine on the RRAM Layer: We further develop the digitalized RRAM based dot-product engine on the RRAM layer. The tensor-train-matrix-by-vector operation can be efficiently accelerated by the fast matrix-vector multiplication engine on the RRAM layer. Each matrix-vector multiplication can be further divided into a vector-vector dot-product operation for parallel computation. Here, we design the digitalized dot-product engine based on [29]. We use the output matrix $\boldsymbol{Y}$, input matrices $\boldsymbol{X}$ and $\boldsymbol{\Phi}$ for better explanation. The overall equation is $\boldsymbol{Y} = \boldsymbol{X}\boldsymbol{\Phi}$ with $\boldsymbol{Y} \in \mathbb{R}^{M\times m}$, $\boldsymbol{X} \in \mathbb{R}^{M\times N}$ and $\boldsymbol{\Phi} \in \mathbb{R}^{N\times m}$. For every element in $\mathbf{Y}$, it follows

$$y_{ij} = \sum_{k=1}^{N} x_{ik}\varphi_{kj}, \tag{11}$$

where x and φ are the elements in $\mathbf{X}$ and $\mathbf{\Phi}$ respectively. The basic idea of implementation is to split the matrix-vector multiplication to multiple dot-product operations of two vectors x_i and φ_j. Furthermore, such multiplication can be computed in the binary data format on RRAM with the adoption of fixed point representation of x_{ik} and φ_{kj}. The multiplication process can be reformulated as

$$\begin{aligned} y_{ij} &= \sum_{k=1}^{N}\Big(\sum_{e=0}^{E-1} B_e^{x_{ik}} 2^e\Big)\Big(\sum_{g=0}^{G-1} B_g^{\varphi_{kj}} 2^g\Big), \\ &= \sum_{e=0}^{E-1}\sum_{g=0}^{G-1}\Big(\sum_{k=1}^{N} B_e^{x_{ik}} B_g^{\varphi_{kj}} 2^{e+g}\Big) = \sum_{e=0}^{E-1}\sum_{g=0}^{G-1} s_{eg} 2^{e+g} \end{aligned} \tag{12}$$

where s_{eg} is the accelerated result from RRAM-crossbar. $B^{x_{ik}}$ is the binary bit of x_{ik} with E bit-width and $B^{\varphi_{kj}}$ is the binary bit of φ_{kj} with G bit-width. As mentioned above, bit-width E and G are decided during the algorithm level optimization.

The dot-product operation for (12) can be summarized in four steps on the second RRAM layer.

Step 1: Index Bijection: Select the correct slice of tensor cores $\boldsymbol{G}_d[i_d, j_d] \in \mathbb{R}^{r_d \times r_{d+1}}$, where a pair of $[i_d, j_d]$ determines a slice from $\boldsymbol{G}_d \in \mathbb{R}^{r_d \times n_d \times r_{d+1}}$. In our current example, we use $\boldsymbol{X} \in \mathbb{R}^{M \times N}$ and $\boldsymbol{\Phi} \in \mathbb{R}^{N \times m}$ to represent two selected slices from cores $\mathbf{G}_1$ and $\mathbf{G}_2$.

Step 2: Parallel Digitizing: The matrix multiplication $\boldsymbol{X} \times \boldsymbol{\Phi}$ requires Mm times N-length vector dot-product multiplication. Therefore, an $N \times N$ RRAM-crossbar is required. For clarity, we explain this step as two sub-steps but they are implemented on the same RRAM-crossbar.

(a) **Binary Dot-Product Process**: The inner-product is computed on RRAM-crossbar as shown in Fig. 7(a). $B_e^{x_{ik}}$ is set at the RRAM-crossbar input and $B_g^{\varphi_{kj}}$ is written in RRAM cells. The multiplication process on RRAM following (10), which produces the analog output.

(b) **Digitalizing**: In the $N \times N$ RRAM-crossbar, resistance of RRAMs in each column is the same, but V_{th} among columns are different. As a result, the output of each column is calculated based on ladder-like threshold voltages $V_{th,c}$ for parallel digitalizing, where c represents RRAM-crossbar column index.

If the inner-product result is s, the output of Step 2 is like $(1...1, 0...0)$, where $O_{1,s} = 1$ and $O_{1,s+1} = 0$.

Step 3: XOR: It is to identify the index of 1 in the binary data s with the operation $O_{1,s} \oplus O_{1,s+1}$. Note that $O_{1,s} \oplus O_{1,s+1} = 1$ only when $O_{1,j} = 1$ and $O_{1,j+1} = 0$ from Step 1. The mapping of RRAM-crossbar input and resistance are also shown in Fig. 7(a), and threshold voltage configuration for each column is $V_{th,c} = \frac{V_r R_s}{2R_{on}}$. Therefore, the index of 1 in the binary data is identified by XOR operation.

Step 4: Encoding: this step is to encode the output from the third step into binary representation $binary(s)$. As shown in Fig. 7(b), the third step output produces $(0...1, 0...0)$ like result where only the s^{th} digit is 1. The implementation of this step is illustrated in Fig. 7(a). Suppose the maximum dot-product result is N, each binary format ($binary(0)$ to $binary(N)$) is stored in the according row. We directly use the third step output as the encoding crossbar input, so that only the s^{th} row is read out in each column. For example, if $s = 5$, the encoding input is (00001000) and the output is (101). Such binary format can be directly stored in memory or used by other computing systems. In the encoding step, it needs an $N \times n$ RRAM-crossbar, where $n = \lceil \log_2 N \rceil$ is the number of bits in order to represent 1 to N in binary format.

Fig. 7(b) gives an example of digitalized dot-product operation on RRAM following these four steps. For the simplicity and clarity of the explanation, we show the dot-product operation of two binary vectors. The Step 2a is bit-wise multiplication operation on the RRAM-crossbar and then by designing ladder-like thresholds on each column, the multiplication result is converted to digital signal in parallel as shown in Step 2b. Then the XOR operation will convert the multiplication result to a $(0...1, 0...0)$ like result. Such result is further encoded to the binary representation in Step 4 encoding process. By applying these four steps, we can map different tensor cores on RRAM-crossbars to perform the matrix-vector multiplication in parallel as shown in Fig. 7(c). Compared to the state-of-arts realizations, this approach can perform the matrix-vector multiplication faster and more energy-efficient, which will be shown in Section V.

V. Experimental result

A. Experiment Settings

In the experiment, we have implemented different baselines for performance comparisons. The detail of each baseline is listed below:

Baseline 1: General CPU processor. The general process implementation is based on Matlab with optimized C-program. The computer server is with 6 cores of $3.46GHz$ and $64.0GB$ RAM.

Baseline 2: General GPU processor. The general-purpose GPU implementation is based on the optimized C-program and Matlab parallel computing toolbox with CUDA-enabled Quadro 5000 GPU [36].

Baseline 3: 3D CMOS-ASIC. The 3D CMOS-ASIC implementation with proposed architecture is done by Verilog with $1GHz$ working frequency based on CMOS 65nm low power PDK. Power, area and frequency are evaluated through Synopsys DC compiler (D-2010.03-SP2). Through-silicon via (TSV) area, power and delay are evaluated based on Simulator DESTINY [34] and fine-grained TSV model CACTI-3DD [35]. The buffer size of the top layer is set $128MB$ to store tensor cores with 256 bits data width. The TSV area is estimated to be $25.0\ \mu m^2$ with capacitance of $21fF$.

Proposed 3D CMOS-RRAM: The settings of CMOS evaluation and TSV model are the same as baseline 2. For the RRAM-crossbar design evaluation, the resistance of RRAM is set as $500K$ and $5M$ as on-state and off-state resistance and 2V SET/RESET voltage according to [37] with working frequency of $200MHz$. The CMOS and RRAM integration is evaluated based on [38].

To evaluate the proposed architecture, we apply UCI [39] and MNIST [40] dataset to analyze the accelerator scalability, model configuration analysis and performance analysis. The model configuration is performed on Matlab first using Tensor-train toolbox [25] before mapping on the 3D CMOS-RRAM architecture. To evaluate the model compression, we compare our method with SVD based node pruned method [41] and general neural network [23]. The energy consumption and speed-up are also evaluated. Note that the code for performance comparisons is based on optimized C-Program and deployed as the mex-file in the Matlab environment.

B. 3D Multi-layer CMOS-RRAM Accelerator Scalability Analysis

Since neural network process requires frequent network weights reading, memory read latency optimization configuration is set to generate RRAM memory architecture. By adopting 3D implementation, Simulation results on Table II show that memory read and write bandwidth can be significantly improved by 51.53% and 6.51% respectively comparing to 2D implementation. For smaller number of hidden nodes,

TABLE II
BANDWIDTH IMPROVEMENT UNDER DIFFERENT NUMBER OF HIDDEN NODES FOR MNIST DATASET

Hidden node†(L)	256	512	1024	2048	4096
Memory required (MB)	1.025	2.55	7.10	22.20	76.41
Memory set (MB)	2M	4M	8M	32M	128M
Write Bandwidth Imp.	1.14%	0.35%	0.60%	3.12%	6.51%
Read Bandwidth Imp.	5.02%	6.07%	9.34%	20.65%	51.53%

†4-layer neural network with 3 full-connected layer $784 \times L$, $L \times L$ and $L \times 10$.

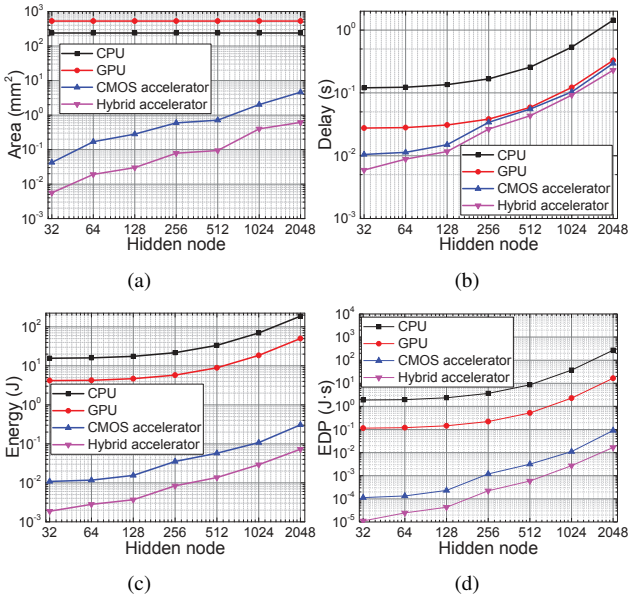

Fig. 8. Scalability study of hardware performance with different hidden node numbers for: (a) area; (b) delay; (c) energy and (d) energy-delay-product

read/write bandwidth is still improved but the bottleneck shifts to the latency of memory logic control.

To evaluate the proposed 3D multi-layer CMOS-RRAM architecture, we perform the scalability analysis of energy, delay and area on MNIST dataset [40]. This dataset is applied to multi-layer neural network and the number of hidden nodes may change depending on the accuracy requirement. As a result, the improvement of proposed accelerator with different L from 32 to 2048 is evaluated as shown in Fig. 8. With the increasing L, more computing units are designed in 3D CMOS-ASIC and RRAM-crossbar to evaluate the performance. The neural network is defined as a 4-layer network with weights $784 \times L$, $L \times L$ and $L \times 10$. For computation delay, GPU, 3D CMOS-ASIC and 3D CMOS-RRAM are close when $L = 2048$ according to Fig. 8(b). When L reaches 256, 3D CMOS-RRAM can achieve $7.56\times$ area-saving and $3.21\times$ energy-saving compared to 3D CMOS-ASIC. Although the computational complexity is not linearly related to the number of hidden node numbers, both energy consumption and energy-delay-product (EDP) of RRAM-crossbar increase with the rising number of hidden node. According to Fig. 8(d), the advantage of the hybrid accelerator becomes smaller when the hidden node increases, but it can still have a $5.49\times$ better EDP compared to the 3D CMOS-ASIC when the hidden node number is 2048.

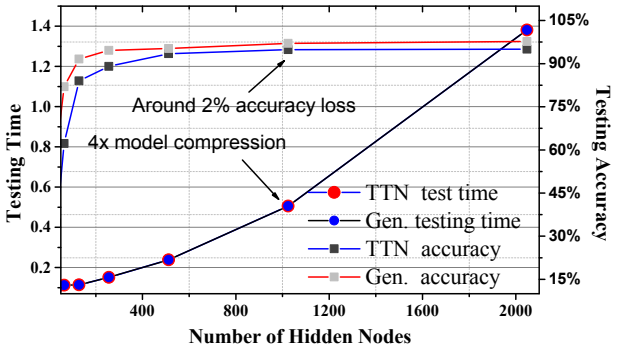

Fig. 9. Testing time and accuracy comparison between tensorized neural network (TTN) and general neural network (Gen.) with varying number of hidden nodes

C. 3D Multi-layer CMOS-RRAM Accelerator Model Configuration Analysis

As discussed in Section II, tensor-train based neural network shows a fast testing process with model compressed when the tensor rank is small. To evaluate this, we apply the proposed learning method comparing to general neural network [23] for speed-up and compression on UCI dataset and MNIST dataset. Please note that the memory required for the tensor-train based weight is $\sum_{k=1}^{d} n_k r_{k-1} r_k$ comparing to $N = n_1 \times n_2 \times ... \times n_d$ and the computation process can be speed-up from $O(NL)$ to $O(dr^2 n\ max(N, L))$, where n is the maximum mode size of the tensor train. Table III shows detailed comparison of speed-up, compressed-model and accuracy between TNN, general neural network and SVD pruned neural network. It clearly shows that proposed method can accelerate the testing process comparing to general neural network. In addition, our proposed method only suffers around 2% accuracy loss but SVD based method has varied loss (up to 18.1 %). Furthermore, by tuning the tensor rank we can achieve 3.13x compression for diabetes UCI dataset. Since we apply 10% node prune by removing the smallest singular values, the model compression remains almost the same for different benchmarks.

Fig. 9 shows the testing accuracy and running time comparisons for MNIST dataset. It shows a clear trend of accuracy improvement with increasing number of hidden nodes. The running time between TNN and general NN is almost the same. This is due to the relative large rank $r = 50$ and computation cost of $O(dr^2 n\ max(N, L))$. Such tensor-train based neural network achieve $4\times$ and $8.18\times$ model compression within 2% accuracy loss under 1024 and 2048 number of hidden nodes respectively. Details on model compression are shown in Table V. From Table V, we can observe that the compression rate is directly connected with the rank r, where the memory storage can be simplified as dnr^2 from $\sum_{k=1}^{d} n_k r_{k-1} r_k$ but not directly link to the number of hidden nodes. We also observe that by setting tensor core rank to 35, $14.85\times$ model compression can be achieved with acceptable accuracy loss. Therefore, initialization of a low rank core and the SVD split of supercore in MALS algorithm (Section II-C) are important steps to reduce the core rank and increase compression rate.

TABLE III
PERFORMANCE COMPARISON BETWEEN TENSORIZED NEURAL NETWORK (TNN), GENERAL NEURAL NETWORK (NN) AND SVD PRUNED NEURAL NETOWRK (SVD) ON UCI DATASET

Dataset§	Model†	Test-time TNN‡ (s)	Test-acc TNN‡	Test-time NN (s)	Test-Acc NN	Test-time SVD (s)	Test-Acc SVD	Mode Cmp		Acc. loss		Speed up	
								TNN	SVD	TNN	SVD	TNN	SVD
iris	128x4x3	7.57E-04	0.968	1.14E-03	0.991	8.73E-04	0.935	1.1200	1.113	0.023	0.056	1.506	1.306
adult	128x14x2	8.70E-03	0.788	1.04E-02	0.784	9.36E-03	0.783	1.8686	1.113	-0.004	0.001	1.195	1.111
credict	128x14x2	9.26E-04	0.778	2.20E-03	0.798	2.02E-03	0.743	1.7655	1.113	0.02	0.055	2.376	1.087
diabetes	128x8x2	8.07E-04	0.71	1.80E-03	0.684	1.70E-03	0.496	3.1373	1.113	-0.026	0.188	2.230	1.061
Glass	64x10x7	6.44E-04	0.886	1.50E-03	0.909	1.12E-03	0.801	1.3196	1.103	0.023	0.108	2.329	1.343
leukemia	256x38x2	4.97E-04	0.889	1.50E-03	0.889	1.49E-03	0.778	1.8156	1.113	0.000	0.111	3.018	1.008
liver	128x16x2	6.15E-04	0.714	2.00E-03	0.685	1.63E-03	0.7000	1.5802	1.113	-0.029	-0.015	3.252	1.228
segment	128x19x12	5.72E-03	0.873	1.84E-02	0.886	1.52E-02	0.847	2.6821	1.113	0.013	0.039	3.207	1.211
shuttle	1024x9x7	5.29E-02	0.995	5.41E-02	0.989	3.81E-02	0.986	1.5981	1.111	-0.006	0.003	1.023	1.419

† $L \times n \times m$, where L is number of hidden nodes, n is the number of features and m is the number of classes. All the datasets are applied to single hidden layer neural network with L sweep from 64 to 1024.
§ Detailed information on dataset can be found from [39]. We randomly choose 80% of total data for training and 20% for testing.
‡ Rank is initialized to be 2 for all tensor cores.

TABLE IV
PERFORMANCE COMPARISON UNDER DIFFERENT HARDWARE IMPLEMENTATIONS ON MNIST DATASET WITH 10,000 TESTING IMAGES

Implementation	General CPU processor [42]				General GPU processor [36]				3D CMOS-ASIC Architecture				3D CMOS-RRAM Architecture			
Freq. Power (W)	3.46GHz, 130W				513MHz, 152W				1GHz, 1.037W				100MHz, 0.317W			
Area (mm^2)	240 (Intel Xeon X5690)				529 (Nvidia Quadro 5000)				9.582 (65nm Global Foundary)				1.026 (65nm CMOS and RRAM)			
Throughput	74.64 GOPS				328.41 GOPS				367.43 GOPS				475.45 GOPS			
Efficiency	0.574 GOPS/W				2.160 GOPS/W				347.29 GOPS/W				1499.83 GOPS/W			
Layer †	L1	L2	L3	Overall	L1	L2	L3	Overall	L1	L2	L3	Overall	L1	L2	L3	Overall
Time (s)	0.44	0.97	0.045	1.45	0.040	0.289	0.0024	0.33	0.032	0.26	2.4E-3	0.295	0.025	0.20	1.7E-3	0.23
Energy (J)	57.23	125.74	5.82	188.8	6.05	43.78	0.3648	50.19	0.0333	0.276	2.6E-3	0.312	7.9E-3	64E-2	5E-4	7.2E-2
Speed-up	–	–	–	–	11.06	3.36	18.67	4.40	13.9	3.71	18.4	4.92	17.77	4.80	25.86	6.37
Energy-saving	–	–	–	–	9.46	2.87	15.96	3.76	1711.8	455.4	2262	604.9	7286	1969	1.1E4	2612

†4-layer neural network with weights 784×2048, 2048×2048 and 2048×10.

TABLE V
MODEL COMPRESSION UNDER DIFFERENT NUMBER OF HIDDEN NODES AND TENSOR RANKS ON MNIST DATASET

No Hid†	32	64	128	256	512	1024	2048
Compression	5.50	4.76	3.74	3.23	3.01	4.00	8.18
Rank‡	15	20	25	30	35	40	45
Compression	25.19	22.51	20.34	17.59	14.85	12.86	8.63
Accuracy (%)	90.42	90.56	91.41	91.67	93.47	93.32	93.86

‡ Number of hidden nodes are all fixed to 2048 with 4 fully connected layers.
† All tensor Rank is initialized to 50.

TABLE VI
TESTING ACCURACY OF ML TECHNIQUES UNDER DIFFERENT DATASET AND BIT-WIDTH CONFIGURATION

Datasets	32-bit Acc. (%) & Compr.		4 bit Acc. (%) & Compr.		5 bit Acc. (%) & Compr.		6 bit Acc. (%) & Compr.	
Glass	88.6	1.32	89.22	10.56	83.18	8.45	88.44	7.04
Iris	96.8	1.12	95.29	8.96	96.8	7.17	96.8	5.97
diabets	71	3.14	69.55	25.12	69.4	20.10	71.00	16.75
adult	78.8	1.87	75.46	14.96	78.15	11.97	78.20	9.97
leuke.	88.9	1.82	85.57	14.56	87.38	11.65	88.50	9.71
MNIST	94.38	4.18	91.28	33.44	92.79	26.75	94.08	22.29

D. 3D Multi-layer CMOS-RRAM Accelerator Bit-width Configuration Analysis

To implement the whole neural network on the proposed 3D multi-layer CMOS-RRAM accelerator, the precision of real values requires a careful evaluation. Compared to the software double precision floating point format (64-bit), values are truncated into finite precision. By using the greedy search method, an optimal point for hardware resource (small bit-width) and testing accuracy can be achieved.

Our tensor-train based neural network compression techniques can work with low-precision value techniques to further reduce the data storage. Table VI shows the testing accuracy by adopting different bit-width on UCI datasets [39] and MNIST [40]. It shows that accuracy of classification is not very sensitive to the RRAM configuration bits for UCI dataset. For example, the accuracy of Iris dataset is working well with negligible accuracy at 5 RRAM bit-width. When the RRAM bit-width increased to 6, it performs the same as 32 bit-width configurations. Please note that the best configuration of quantized model weights varies for different datasets and requires careful evaluation.

E. 3D Multi-layer CMOS-RRAM Accelerator Performance Analysis

In Table IV, performance comparisons among C-Program Optimized CPU performance, GPU performance , 3D CMOS-ASIC and 3D multi-layer CMOS-RRAM accelerator are presented for $10,000$ testing images. The acceleration of each layer is also presented for 3 layers (784×2048, 2048×2048 and 2048×10). Please note that the dimension of weight matrices are decomposed into $[4\ 4\ 7\ 7]$ and $[4\ 4\ 8\ 8]$ with 6 bit-width and maximum rank 6. The compression rate is $22.29\times$ and $4.18\times$ with and without bit-truncation. Among the four implementation, 3D multi-layer CMOS-RRAM accelerator performs the best in area, energy and speed. Compared to CPU, it achieves $6.37\times$ speed-up, $2612\times$ energy-saving and $233.92\times$ area-saving. For GPU based implementation, our proposed 3D CMOS-RRAM architecture achieves $1.43\times$ speed-up and $694.68\times$ energy-saving. We also design a 3D CMOS-

ASIC implementation with similar structure as 3D multi-layer CMOS-RRAM accelerator with better performance compared to CPU and GPU based implementations. The proposed 3D multi-layer CMOS-RRAM 3D accelerator is $1.283\times$ speed-up, $4.276\times$ energy-saving and $9.339\times$ area-saving compared to 3D CMOS-ASIC.

The throughput and energy efficiency for these four cases are also summarized in Table III. For energy efficiency, our proposed accelerator can achieve 1499.83 GOPS/W, which has $4.30\times$ better energy efficiency comparing to 3D CMOS-ASIC result (347.29 GOPS/W). In comparison to our GPU baseline, it has $694.37\times$ better energy efficiency comparing to NVIDIA Quadro 5000. For a newer GPU device (NVIDIA Tesla K40), which can achieve 1092 GFLOPS and consume 235W [36], our proposed accelerator has $347.49\times$ energy efficiency improvement.

VI. Conclusion

In this paper, we propose a 3D multi-layer CMOS-RRAM accelerator for highly-parallel yet energy-efficient machine learning. A tensor-train based tensorization is developed to represent dense weight matrix with significant compression. The neural network processing is mapped to a 3D architecture with high-bandwidth TSVs, where the first RRAM layer is to buffer input data; the second RRAM layer is to perform intensive matrix-vector multiplication using digitized RRAM; and the third CMOS layer is to coordinate the reaming control and computation. Simulation results using the benchmark MNIST show that the proposed accelerator has $1.283\times$ speed-up, $4.276\times$ energy-saving and $9.339\times$ area-saving compared to 3D CMOS-ASIC implementation; and $6.37\times$ speed-up and $2612\times$ energy-saving compared to 2D CPU implementation. In addition, $14.85\times$ model compression can be achieved by tensorization with acceptable accuracy loss.

References

[1] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural computation*, vol. 18, no. 7, pp. 1527–1554, 2006.

[2] K. R. Müller, M. Tangermann, G. Dornhege, M. Krauledat, G. Curio, and B. Blankertz, "Machine learning for real-time single-trial EEG-analysis: from brain–computer interfacing to mental state monitoring," *Journal of neuroscience methods*, vol. 167, no. 1, pp. 82–90, 2008.

[3] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *International conference on artificial intelligence and statistics*, 2010, pp. 249–256.

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

[5] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2015, pp. 161–170.

[6] Y.-H. Chen, T. Krishna, J. Emer, and V. Sze, "14.5 eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," in *2016 IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE, 2016, pp. 262–263.

[7] Y. Wang, H. Yu, L. Ni, G.-B. Huang, M. Yan, C. Weng, W. Yang, and J. Zhao, "An energy-efficient nonvolatile in-memory computing architecture for extreme learning machine by domain-wall nanowire devices," *IEEE Transactions on Nanotechnology*, vol. 14, no. 6, pp. 998–1012, 2015.

[8] Y. Wang, C. Zhang, R. Nadipalli, H. Yu, and R. Weerasekera, "Design exploration of 3d stacked non-volatile memory by conductive bridge based crossbar," in *IEEE 3DIC*, 2012.

[9] Y. Wang, X. Li, K. Xu, F. Ren, and H. Yu, "Data-driven sampling matrix boolean optimization for energy-efficient biomedical signal acquisition by compressive sensing," *IEEE Transactions on Biomedical Circuits and Systems*, 2016.

[10] I. Micron Technology, "Breakthrough nonvolatile memory technology." [Online]. Available: http://www.micron.com/about/emerging-technologies/3d-xpoint-technology/

[11] S. Yu and et al., "3D vertical RRAM-scaling limit analysis and demonstration of 3D array operation," in *IEEE VLSIT*, 2013.

[12] D. C. Plaut *et al.*, "Experiments on learning by back propagation." 1986.

[13] I. Hubara, D. Soudry, and R. E. Yaniv, "Binarized neural networks," *arXiv preprint arXiv:1602.02505*, 2016.

[14] A. Davis and I. Arel, "Low-rank approximations for conditional feedforward computation in deep neural networks," *arXiv preprint arXiv:1312.4461*, 2013.

[15] P. Nakkiran, R. Alvarez, R. Prabhavalkar, and C. Parada, "Compressing deep neural networks using a rank-constrained topology," 2015.

[16] A. Novikov, D. Podoprikhin, A. Osokin, and D. Vetrov, "Tensorizing Neural Networks," *ArXiv e-prints*, Sep. 2015.

[17] I. V. Oseledets, "Tensor-train decomposition," *SIAM Journal on Scientific Computing*, vol. 33, no. 5, pp. 2295–2317, 2011.

[18] A. Cichocki, "Era of big data processing: A new approach via tensor networks and tensor decompositions," *arXiv preprint arXiv:1403.2048*, 2014.

[19] M. T. Hagan, H. B. Demuth, M. H. Beale, and O. De Jesús, *Neural network design*. PWS publishing company Boston, 1996, vol. 20.

[20] Y. Bengio, P. Lamblin, D. Popovici, H. Larochelle *et al.*, "Greedy layer-wise training of deep networks," *Advances in neural information processing systems*, vol. 19, p. 153, 2007.

[21] J. Tang, C. Deng, and G.-B. Huang, "Extreme learning machine for multilayer perceptron," *IEEE transactions on neural networks and learning systems*, vol. 27, no. 4, pp. 809–821, 2016.

[22] D. Erhan, P.-A. Manzagol, Y. Bengio, S. Bengio, and P. Vincent, "The difficulty of training deep architectures and the effect of unsupervised pre-training." in *AISTATS*, vol. 5, 2009, pp. 153–160.

[23] G. B. Huang, Q. Y. Zhu, and C.-K. Siew, "Extreme learning machine: theory and applications," *Neurocomputing*, vol. 70, no. 1, pp. 489–501, 2006.

[24] S. Holtz, T. Rohwedder, and R. Schneider, "The alternating linear scheme for tensor optimization in the tensor train format," *SIAM Journal on Scientific Computing*, vol. 34, no. 2, pp. A683–A713, 2012.

[25] I. V. Oseledets and S. Dolgov, "Solution of linear systems and matrix inversion in the tt-format," *SIAM Journal on Scientific Computing*, vol. 34, no. 5, pp. A2718–A2739, 2012.

[26] K. H. Kim, S. Gaba, D. Wheeler, J. M. Cruz Albrecht, T. Hussain, N. Srinivasa, and W. Lu, "A functional hybrid memristor crossbar-array/CMOS system for data storage and neuromorphic applications," *Nano letters*, vol. 12, no. 1, pp. 389–395, 2011.

[27] H. Akinaga and H. Shima, "Resistive random access memory (ReRAM) based on metal oxides," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2237–2251, Dec 2010.

[28] P. Gu, B. Li, T. Tang, S. Yu, Y. Cao, Y. Wang, and H. Yang, "Technological exploration of RRAM crossbar array for matrix-vector multiplication," in *IEEE ASP-DAC*, 2015.

[29] L. Ni, H. Huang, Z. Liu, R. V. Joshi, and H. Yu, "Distributed in-memory computing on binary rram crossbar," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 13, no. 3, p. 36, 2017.

[30] R. O. Topaloglu, *More than moore technologies for next generation computer design*. Springer, 2015.

[31] Y. Wang, H. Yu, and W. Zhang, "Nonvolatile cbram-crossbar-based 3-d-integrated hybrid memory for data retention," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 5, pp. 957–970, 2014.

[32] Y. Y. Liauw, Z. Zhang, W. Kim, A. El Gamal, and S. S. Wong, "Non-volatile 3D-FPGA with monolithically stacked rram-based configuration memory," in *IEEE ISSCC*, 2012.

[33] Y.-C. Chen, W. Wang, H. Li, and W. Zhang, "Non-volatile 3D stacking rram-based fpga," in *IEEE FPL*, 2012.

[34] M. Poremba, S. Mittal, D. Li, J. S. Vetter, and Y. Xie, "Destiny: A tool for modeling emerging 3d nvm and edram caches," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*. IEEE, 2015, pp. 1543–1546.

[35] K. Chen, S. Li, N. Muralimanohar, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "Cacti-3dd: Architecture-level modeling for 3d die-stacked dram main memory," in *Proceedings of the Conference on Design, Automation and Test in Europe*. IEEE, 2012, pp. 33–38.
[36] "GPU specs," http://www.nvidia.com/object/workstation-solutions.html, accessed: 2017-03-30.
[37] B. Govoreanu, G. Kar, Y. Chen, V. Paraschiv, S. Kubicek, A. Fantini, I. Radu, L. Goux, S. Clima, R. Degraeve *et al.*, "10× 10nm 2 hf/hfo x crossbar resistive ram with excellent performance, reliability and low-energy operation," in *Electron Devices Meeting (IEDM), 2011 IEEE International*. IEEE, 2011, pp. 31–6.
[38] W. Fei, H. Yu, W. Zhang, and K. S. Yeo, "Design exploration of hybrid CMOS and memristor circuit by new modified nodal analysis," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 20, no. 6, pp. 1012–1025, 2012.
[39] M. Lichman, "UCI machine learning repository," 2013. [Online]. Available: http://archive.ics.uci.edu/ml
[40] Y. LeCun, C. Cortes, and C. J. Burges, "The mnist database of handwritten digits," 1998.
[41] J. Xue, J. Li, and Y. Gong, "Restructuring of deep neural network acoustic models with singular value decomposition." in *INTERSPEECH*, 2013, pp. 2365–2369.
[42] "CPU xeon-x5690specs," https://ark.intel.com/products/52576/Intel-Xeon-Processor-X5690-12M-Cache-3_46-GHz-6_40-GTs-Intel-QPI, accessed: 2017-03-30.

Hantao Huang (S'14) received the BS degree from Nanyang Technological University (NTU), Singapore in 2013. Since 2014, he is working towards the PhD degree from the School of Electrical and Electronic Engineering in Nanyang Technological University. His PhD is sponsored by MediaTek. His research interests are data analytics, machine-learning algorithms, and low-power system design. He is a student member of the IEEE.

Leibin Ni (S'17) received the B.S. degree in microelec- tronic from Shanghai Jiao Tong University, Shang- hai, China, 2014. He is currently pursuing the Ph.D. degree from the School of Electrical and Elec- tronic Engineering, Nanyang Technological Univer- sity, Singapore. His current research interests include emerging nonvolatile memory platform and big-data inmemory computing.

Kanwen Wang received the B.S. and Ph.D degree in microelectronics from Fudan University, Shanghai, China, in 2006 and 2012, respectively. From 2012 to 2014, he has been a research fellow at the School of Electrical and Electronic Engineering, Nanyang Technological University, Singapore. Since November 2014, he joined Data Center Technology Laboratory, 2012 Labs, Huawei Technologies Co., Ltd as a senior research engineer. His primary research interests include 2.5D/3D system architectures for exa-scale computing, future memory systems, emerging device technologies for future computing paradigms.

Yuangang Wang received the M.S degree in computer science from National University of Defence Technology, Changsha, China, in 2008. Since March 2008, he joined Data Center Technology Laboratory, 2012 Labs, Huawei Technologies Co., Ltd as a senior research engineer. As a project manager, he has led several research programs regarding new technology innovations. His primary research interests include SCM storage, new memory architecture, PIM technology, system architectures for exa-scale computing, AI system architecture, emerging device technologies for future computing paradigms.

Hao Yu (M'06-SM'14) received the B.S. degree from Fudan University, Shanghai, China, and the Ph.D. degree from the Department of Electrical Engineering, University of California, CA, USA. He was with the School of Electrical and Electronic Engineering, Nanyang Technological University, Singapore and now is with Southern University of Science and Technology, China. His primary research interest is CMOS emerging technology for data sensor, link and accelerator. He received the Best Paper Award from the ACM TODAES10, the Best Paper Award nominations in DAC06, ICCAD06, ASP-DAC12, the Best Student Paper (advisor) Finalist in SiRF13, RFIC13, IMS'15, and SRC Inventor Award09. He is the Distinguished Lecturer of IEEE CAS since 2017. He is the Associate Editor and technical program committee member for a number of journals (Nature Scientific Reports, IEEE Trans. on BioCAS, ACM Trans. on Embedded Computing System, Elsevier Microelectronics Journal, Integration, the VLSI Journal) and conferences (DAC, ASSCC, DATE, ISLPED etc.).