ReRAM-based Accelerator for Deep Learning

Bing Li*, Linghao Song*, Fan Chen*, Xuehai Qian†, Yiran Chen* and Hai (Helen) Li*
Email: *{bing.li.ece, linghao.song, fan.chen, yiran.chen, hai.li}@duke.edu, †xuehai.qian@usc.edu,
*Department of Electrical and Computer Engineering, Duke University, Durham, NC, United States
†Department of Computer Science, University of Southern California, Los Angeles, CA, United States

***Abstract*—Big data computing applications such as deep learning and graph analytic usually incur a large amount of data movements. Deploying such applications on conventional von Neumann architecture that separates the processing units and memory components likely leads to performance bottleneck due to the limited memory bandwidth. A common approach is to develop architecture and memory co-design methodologies to overcome the challenge. Our research follows the same strategy by leveraging resistive memory (ReRAM) to further enhance the performance and energy efficiency. Specifically, we employ the general principles behind processing-in-memory to design efficient ReRAM based accelerators that support both testing and training operations. Related circuit and architecture optimization will be discussed too.**

I. Introduction

Deep neural networks (DNN) have made remarkable successes in the *artificial intelligence* (AI) domain, such as computation vision, natural language process, and image processing. Increasing the scale of DNNs is a simple approach to elevate the performance of applications, while larger and deeper network models demand more computation resources and data storage. For example, AlexNet [1] in 2012 executed about 100 million operations for one image processing, while GoogleNet [2] in 2014 required 3.9 billion ones. Due to the large amount of data transition between the on-chip logic units and the off-chip memory, deploying DNNs on the computing platforms based on traditional computing paradigms faces unprecedented challenges [3]. With the end of Moore's law on the horizons, advance in architecture and memory co-design is urgent to accelerate deep learning.

Processing-near-memory and *processing-in-memory* (PIM) that attempt to integrated more local memory and even use it for computation become an attractive approach for deep learning acceleration. Recently 3D-stacked DRAM memory technology is utilized to satisfy the high bandwidth requirement of the memory-intensive applications [4], [5]. 3D memory based PIM stacks the logic chip with the 3D memory chip using massive number of *through-silicon-vias* (TSV) [6], where computation and memory are close but decoupled with each other.

Emerging *resistive random access memory* (ReRAM) [7] in simple crossbar array has a property of bitline current summation. It achieves computation and storage simultaneously as a low-cost matrix-vector multiplication. Example ReRAM-based PIM architectures include Prime [8] and ISAAC [9]. However, deploying the complete execution of DNN on ReRAM-based structures remains difficult due to the lacking of support for sophisticated training procedure. Compared to the testing procedure, the training involves iterative weight updates and complicate data dependencies. The latest *generative adversarial networks* (GAN) [10], [11] that comprises two co-trained networks further aggravates the hardware requirement and therefore deserves specific optimizations. Our work primarily focuses on the ReRAM-based PIM design and optimization to accelerate both inference and training processes of DNNs [12]–[14].

This paper is organized as follows: Section II introduces the background of neural networks and ReRAM-based PIM design. Section III elaborates ReRAM-based PIM architectures along with circuit implementations for DNN and GAN. At the end, we conclude this paper in Section IV.

II. Background

A. Basics of Deep Neural Network

1) CNN Structure: Here, we take the common deployed *convolutional neural network* (CNN) as an example to explain the DNN model and the associated computation requirement. A CNN is composed of three types of layers: *convolutional layer* (CONV), *pooling layer* (POOL) and *fully-connected layer* (FC). Fig. 1 shows the basic structures in CNN models.

In a convolution layer, a set of kernels are convoluted with data of channels from the previous layer (layer l) to generate data for channels of next layer (layer $l+1$). d_l is a cube of data in a layer. $d_l[x,y,c]$ is the value at a point in the three dimensional data cube. We also denote the size of d in layer l as $(X_l \times Y_l \times C_l)$, so $0 \leq x \leq X_d - 1, 0 \leq y \leq Y_d - 1, 0 \leq c \leq C_d - 1$ and C_d is the number of channels. (x_l, y_l, c_l) indicates a point in layer l's data cube. K is the kernel composed of a set of weights. K_l is the kernel used in the computation to generate data in layer l. A kernel represents four dimensional data: the size of each dimension is K_x, K_y, C_l and C_{l+1}, where K_x and K_y are determined by algorithm. d_{l+1} is computed as:

$$d_{l+1}[x,y,c] = \sum_{c_l=0}^{C_l-1} \sum_{k_x=0}^{K_x-1} \sum_{k_y=0}^{K_y-1} K_l[k_x,k_y,c_l,c] \times d_l[x+k_x, y+k_y, c_l] \tag{1}$$

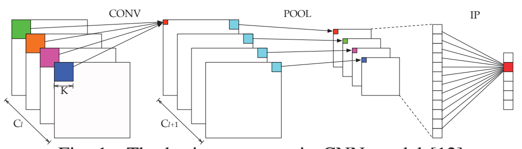

Fig. 1. The basic structures in CNN model [12].

A convolutional layer is always followed by an element-wise non-linearity activation function. The common used function is *rectified linear unit* (ReLU).

A pooling layer performs the down-sampling along the spatial dimensions (width and height). A max POOL passes the maximum element in a pooling window while an average POOL takes the mean of all the elements in a pooling window.

In fully connected layers, a.k.a. *inner product* layer (IP), the values in data tube of l and $l+1$ are considered as a vector (denoted as $\vec{d_l}$ and $\vec{d_{l+1}}$). If the previous layer is convolution or pooling, the size of $\vec{d_l}$ is $X_l \times Y_l \times C_l$. If the previous layer is also inner product, then the size of $\vec{d_l}$ is the size of the output vector from l. $\vec{d_{l+1}}$ is a $n \times 1$ vector, n is determined by the algorithm. W_{l+1-l} is a weight matrix of size $(n \times m)$, m is the size of $\vec{d_l}$. $\vec{b}$ is a vector of bias. The vector of $l+1$ is computed as:

$$\vec{d_{l+1}} = W_{l+1-l}\vec{d_l} + \vec{b} \tag{2}$$

Among the three types of layers, convolutional layers contribute the most to computation as matrix-vector computation and accumulation are the prominent operations in DNNs [2].

2) Data Forward and Backward: The complete execution of a neural network includes both training (learning) and testing (inference). In the testing phase, a DNN performs some specific tasks on input data, such as classification and recognition. The input data flow through the layers consecutively in forward direction. The weight values of the DNN are determined by the training process, which involves both data forwarding and back propagation.

During the training, a training example first goes through the network from the first layer to the last one, generating an output. The error between the network output and its corresponding expected output will be quantitatively calculated. The error will then be propagated backwards to the previous layer and used to calculated the change of weights. The back-propagation continues till the input layer as such the weights of every layers are updated. In practice, training examples are placed in batches. The error is averaged at the end of each batch, which is then used to update the weights.

The forward computation that are executed in both testing and training phases can be simply realized through matrix multiplication of input feature maps and kernels. The training phase also involves data back propagation and the computation becomes more complex. For example the weight updates depend on the previous layer's errors and the input data of the earlier forward phase.

3) Generative Adversarial Networks (GANs): GANs and its variations have recently been extensively deployed in various applications, for example, recovering corrupted images based on the surrounding values [15] or generating novel art with established styles [16]. A very important feature of GAN is that can be used for unsupervised and semi-supervised learning to reduce large volume of annotations and labels commonly required by supervised deep leaning algorithms. In the context of image processing, most of the GAN variations are (or at least partially) based on the *deep convolutional generative adversarial networks* (DCGAN) [3], [10], [17]–[19]. So we use DCGAN as an example to introduce the frame of GAN.

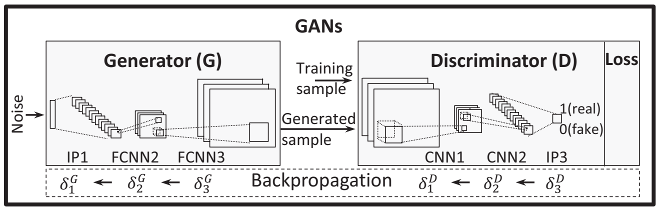

Fig. 2. A GAN system [13].

As illustrated in Fig. 2, a GAN contains two sub-networks, namely, *Generator* (G) and *Discriminator* (D). Both of them are typically modeled as DNNs. In GAN-based applications, G is optimized to produce fake samples to fool D while D is a binary classifier which is trained to distinguish real samples from those generated by G. More specific, D acts as the general CNN which "down-samples" the input to produce classification [3]. In contrast, G takes a uniform noise distribution as input, which is then projected to a small spatial extent convolutional representation with many feature maps and used as the start of a series of *fractional-strided convolution* layers (FCNN). Different from the traditional convolution operations, a FCNN inserts zeros into its input feature map and then performs up-sampling convolution. So the generated output feature map has a much larger dimension than the input one. In addition, the training process of GANs usually operates the batch normalization before the activation layer to improve its stability.

B. ReRAM Used for PIM

Resistive random access memory (ReRAM) is a type of non-volatile memory that stores information as device resistance states. Recent studies demonstrated that ReRAM is a promising candidate to realize area efficient matrix-vector multiplication in a crossbar architecture [8], [9]. Fig. 3(a,b) shows an example of mapping a matrix-vector multiplication to a ReRAM crossbar. The vector is represented by the input signals on the *wordlines* (WLs). Each element of the matrix is programmed into the cell conductance in the crossbar array. Thus, the current flowing to the end of each *bitline* (BL) is

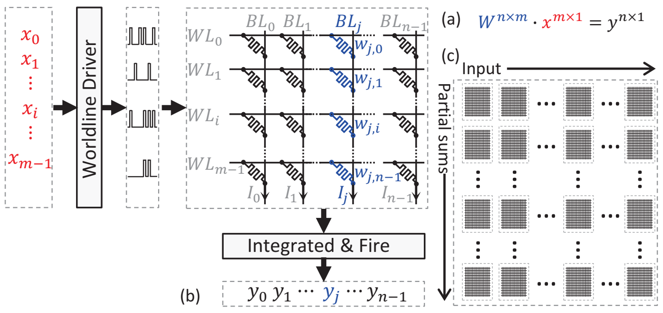
x_0
x_1
x_i
x_{m-1}
Worldline Driver
BL_0 BL_1 BL_j BL_{n-1}
WL_0
WL_1
WL_i
WL_{m-1}
$w_{j,0}$
$w_{j,1}$
$w_{j,i}$
$w_{j,n-1}$
I_0 I_1 I_j I_{n-1}
Integrated & Fire
(b)
$y_0\ y_1 \cdots y_j \cdots y_{n-1}$
(a) $W^{n\times m} \cdot x^{m\times 1} = y^{n\times 1}$
(c)
Input
Partial sums

Fig. 3. Mapping a matrix-vector multiplication (a) to a ReRAM crossbar array (b). (c) Mapping large matrix to multiple ReRAM arrays [13].

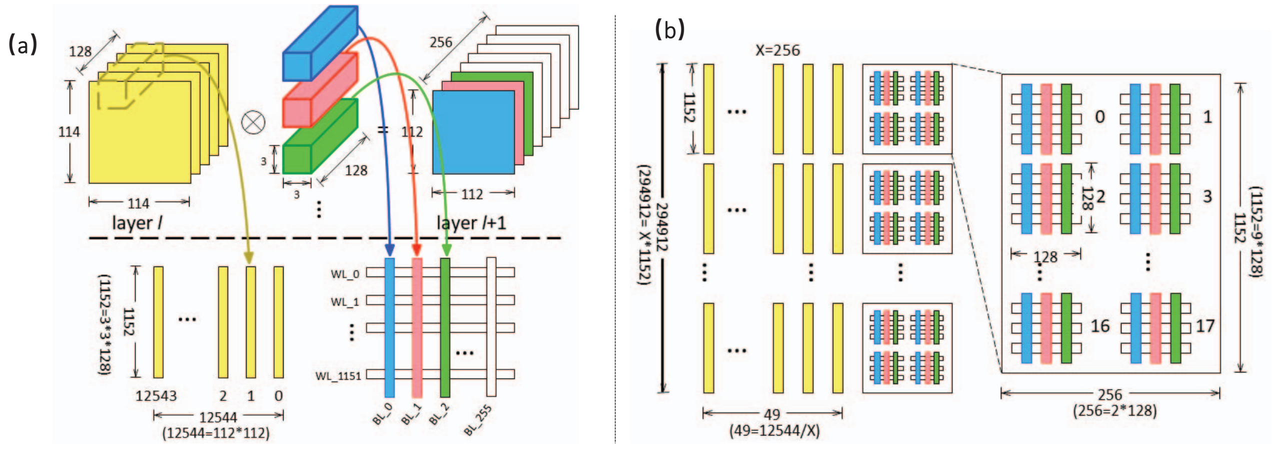
Fig. 4. (a) A naïve scheme and (b) a balanced scheme for data input and kernel mapping [12].

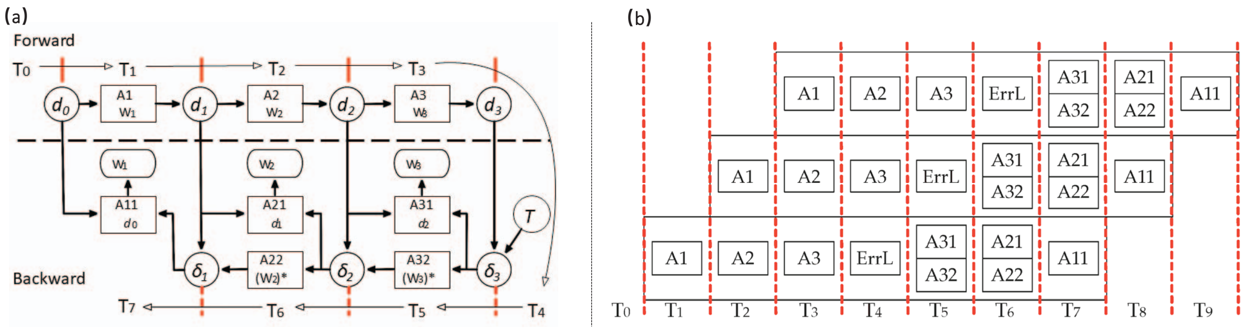
Fig. 5. (a) The data dependencies of a four-layer neural network. (b) The inter-layer pipeline design. [12]

viewed as the result of the matrix-vector multiplication. For a large matrix that can not fit in a single array, the input and the output shall be partitioned and grouped into multiple arrays as shown in Fig. 3(c). The output of each array is a partial sum, which is collected horizontally and summed vertically to generate the final calculation results.

III. ReRAM based PIM Accelerator

We investigated the ReRAM-based PIM for DNN and GAN acceleration. The design and optimization to speedup both the training and testing processes were explored.

A. Accelerator for General Neural Networks

PipeLayer [12] is a ReRAM-based PIM accelerator that supports complete deep learning applications of general neural networks. We investigated the data mapping and the inter-layer pipeline to improve the efficiency of testing and training.

1) Data Mapping: Fig. 4(a) illustrates a naïve scheme of data input and kernel mapping to a ReRAM array. In this example, the data of layer l, kernels and data of layer $l+1$ have a size of $114\times114\times128$, $3\times3\times128\times256$ and $112\times112\times256$, respectively. Each kernel in layer $l+1$ has a size of $3\times3\times128$. The size of an input vector (yellow bar) to the ReRAM array at one cycle is 1152×1 (the bias is neglected for express clarity). As the input data enter ReRAM array sequentially, the given example will take 12544 cycles to get all outputs of layer $l+1$. One kernel is mapped to the bit line, for example, the blue cuboid is mapped to the blue bar in the array and it is the same case for the red, green and the rest cuboids, resulting in 256 bitlines and 1152 wordlines for the ReRAM array.

Fig. 4(b) illustrates a data mapping scheme for performance and execution efficiency improvement. The 1152×256 matrix is divided into a group of 18 ($= 9 \times 2$) matrices and each of subgroup maps to a 128×128 ReRAM array. The array outputs are horizontally collected and then vertically added to get the results. In addition, the weights are duplicated into X copies and stored in multiple ReRAM subarrays to achieve higher intra parallelism. If $X = 1$, the design is equivalent to the naive scheme. If $X = 12544$, the results of a layer could be generated in just one cycle but the hardware cost is excessive. Essentially, a good trade-off between hardware resource of ReRAM array and performance requires a carefully chosen X. Fig. 4 is an example with $X = 256$.

2) Inter Layer Parallelism: For simplicity, Fig. 5(a) shows the data dependency during the training of a 3-layer CNN. The rectangles are ReRAM subarrays that perform the computation for one layer. The circles are memory subarrays to store intermediate results transferred between ReRAM subarrays for different layers. Red dashed lines denote the system states between two consecutive cycles. Data dependencies exist in forward and backward computations. Training phase is more complicated than testing phase because it involves data dependencies between outputs from previous layers and the compute of partial derivatives and errors for weight updates.

In training, the input data are normally processed in batch. The inputs in the same batch are all processed based on the same weights at the start of the batch. The weight updates due to each input are stored and only applied to at the end of a batch. Therefore, no dependency exists among data inputs inside a batch. The *Pipelayer* architecture in Fig. 5(b) was proposed to support the training pipeline.

Assuming that a neural network has L layers and the batch size is B. The computation of a batch requires $(2L+1)B+1$ cycles, in which the forward process takes $L \times B$ cycles, the backward computation takes $(L+1) \times B$ cycles, and each weight update needs one cycle. For a total number of N inputs, the total number of training cycles is $(2L+1)N+N/B$. For the pipelined execution, a new input could enter every cycle within a batch. A new input belonging to the next batch, however, cannot enter the pipeline until all inputs in the previous batch are processed and weights are updated. The first weight update is generated after $(2L+1)$ cycles. Then there will be $(B-1)$ cycles until the end of batch. Finally, one cycle is needed to update all weights within the batch. The total number of cycles to process N inputs with L layers is $(N/B)(2L+B+1)$.

3) Implementation of PipeLayer: Fig. 6 shows the implementation of PipeLayer, in which a memory bank is divided into three regions—morphable subarrays, memory subarrays, and bank buffer subarrays. The ReRAM-based morphable subarray can alter its function between memory and computing modes. A morphable unit behaves the same as a regular ReRAM subarray in the memory mode and performs matrix-vector multiplications in the computing mode. Extra peripheral circuitry are extended to supports activation function. Memory subarrays are used as buffers to store intermediate results, which greatly reduces the data movements. Each memory bank contains a bank control unit, which decodes the incoming instructions and determines the operation mode of morphable subarrays.

Here are the details of some key circuit components used in the hardware implementation. *(a) Spike driver* converts the input to a sequence of spikes. In weight update, it serves as write driver to tune weights stored in the ReRAM array. PipeLayer uses a weighted spike coding scheme [9] to further reduce the area and energy overhead. *(b) Integration and fire circuit (I&F)* is a required component of a spike-based scheme. It integrates input current and generates output spikes. The output spikes are connected to a counter. Essentially the analog currents are converted into digital values. *(c) Activation function* defined in CNN algorithms is implemented in our design. When morphable subarrays are configured as memory, it is bypassed. A register is used to keep the maximum value of a sequence, realizing the max pooling function. *(d) Connection component* is used to connect morphable subarrays and memory subarrays. When a morphable subarray is in computing mode, it needs to store the produced output into memory subarray(s) so the output can be used as inputs of the morphable subarrays for other layers in the following cycle. *(e) Control unit* offloads the computation from the host CPU and orchestrates the data transfers between memory subarrays and morphable subarrays in training and testing based on the algorithm configurations (e.g., batch size).

B. Accelerator for Generated Adversarial Networks

ReGAN [13] is a ReRAM-based PIM accelerator to improve the computational efficiency for GAN training. The details of the architecture design shall be discussed in this subsection.

1) Mapping GAN to ReRAM Crossbar Arrays: As aforementioned, a GAN contains two different subnetworks, making the hardware implementation more challenging. Compared to the general neural networks discussed in Section III-A, the generator (G) employs a different convolution, namely, *fractional-strided convolution* layers (FCNN), which is used to project the input feature maps to a higher-dimensional space. However, as illustrated in Fig. 7(a), the computation of a FCNN during data forwarding process can be taken the same way as a traditional convolution [20] by first adding zeros between each input in the feature maps with zero padding and then computing the convolution between the extended input feature maps and the kernel. Fig. 7(b) describes the error propagation backwards in FCNN, which indeed is a typical convolution with strides.

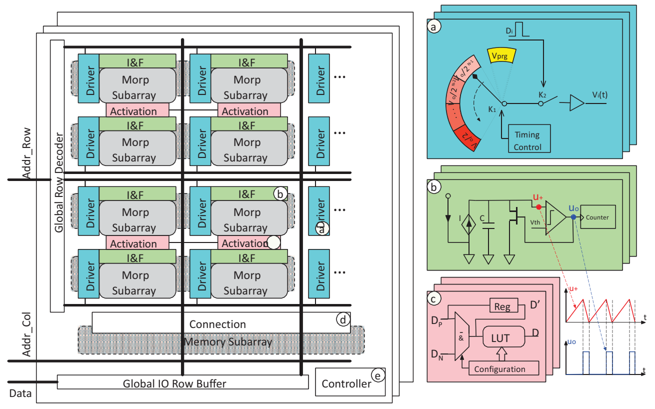

Fig. 6. The *PipeLayer* implementation.

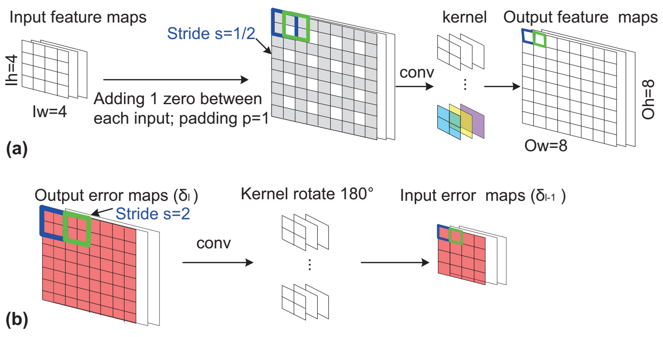

Fig. 7. Visualization of a single fractional-strided convolution layer during (a) data forwarding and (b) error backpropagation processes [13].

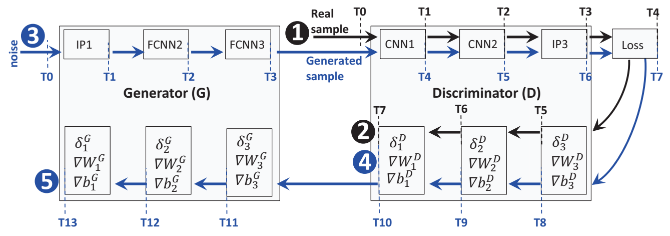

Fig. 8. The training pipeline in GAN [13].

2) *Training Pipeline for Efficiency:* The training pipeline in Fig. 8 is formed in ReGAN architecture to increase the system throughput of the training process.

Training Discriminator (D). D is trained on the training samples as well as the faked samples generated by G. (a) When training on the training samples, the dataflow is illustrated by ❶~❷ in Fig. 8. A real training sample is fed into D and flows through layers of D consecutively in forward direction. A loss function is then computed at `T4` based on accurate labels ('1' for training sample). Finally, the error and partial derivatives propagate all the way back to the first layer of D and are stored. (b) ❸~❹ of Fig. 8 depicts the dataflow of training D on generated samples. In this case, G is concatenated with D to form a large network. G maps a random vector to a sample which has the same dimension with real samples. This sample follows the layers in D and a loss function is performed with the label ('0' for generated sample). Similarly, the partial derivatives propagate back to the first layer of D at `T10`. Therefore, in `T11`, the derivatives stored at the end of stages (a) and(b) are summed accordingly and then used to update the weights of D. During this process, G is used but not updated.

Training Generator (G). The data flow of training G follows ❸~❺ in Fig. 8. The procedure is very similar to training D with generated samples depicted by ❸~❹, except: (a) The error is computed with inaccurate labels ('1' for generated sample) in `T7`; (b) The error propagates all the way back to the first layer of G; and (c) The weights of G are updated in `T14` while D is fixed.

Assuming D has L_D layers and G has L_G layers. To update D, training D on real samples takes $2L_D + 1 + B - 1$ cycles as a new batch has to wait $B - 1$ cycles for the previous batch to drain from the pipeline; then $L_G + 2L_D + 1 + B - 1$ cycles to train D on generated samples; finally, it takes one cycle to update D. Similarly, it takes $2L_G + 2L_D + B + 1$ cycles to train G. Without the training pipeline, the D and G training processes for a batch of data consume $(4L_D + L_G + 2)B$ cycles and $(2L_D + 2L_D + 1)B$ cycles, respectively.

3) *Pipeline Optimization: Spatial parallelism* (SP) and *computation sharing* (CS) were proposed in ReGAN to further improve the pipeline performance. As D remains unchanged in ❶~❷ and ❸~❹, we proposed to duplicate D into two copies. So the training phases ❶~❷ and ❸~❹ work in parallel, realizing the *spatial parallelism*. The latency of ❶~❷ is hidden so the effective latency is reduced to the one of ❸~❹.

Fig. 9 highlights the difference between training phases ❸~❹ and ❸~❺. They share the same forward path but have different loss functions, and hence different back-propagation

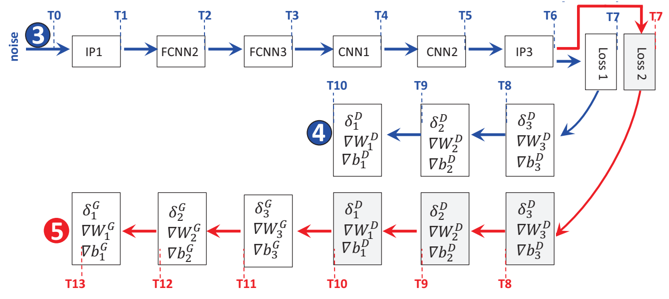

Fig. 9. Improved pipeline by computation sharing [13].

data paths. *Computation sharing* attempts to co-train D and G by doubling the memory storage for intermediate computation (i.e., the error and partial derivatives). As such, the training of D and G share the forward path `T0-T6`. At cycle `T7`, two backward branches are executed in parallel. The weights of G will be updated at `T14` and D can be updated at `T11`.

4) *Hardware Implementation:* Fig. 10 shows the overview of ReGAN implementation. The design partitions the ReRAM main memory into three regions: *memory* (Mem) subarrays, *full function* (FF) subarrays, and Buffer subarrays. Mem subarrays are the same as conventional memory subarrays for data storage. FF subarrays can be configured in both computation and storage modes. In computation mode, FF subarrays execute matrix-vector multiplications; in memory mode, they are used as Mem subarrays for data storage. Buffer subarrays are used to store the intermediate results between layers (e.g., generated images, data required for compute partial derivatives, etc.). They are connected to FF subarrays through private data ports, so that buffer accesses do not consume the bandwidth of Mem subarrays.

FF subarrays are implemented to support the *batch normalization* (BN), activation function, and *fully connected* (FC) layers in ReRAM arrays. First, additional components are included in wordline drivers to support *virtual batch normalization* (VBN) (marked as light blue in Fig. 10Ⓐ), in which each example is normalized based on the statistics collected on a reference batch [19]. The reference batch is chosen once and fixed at the start of training. *Sub* and *shift* are used to do the minus and division in BN layers, and the divisor is 2^n.

To realize the activation function, the results from both the

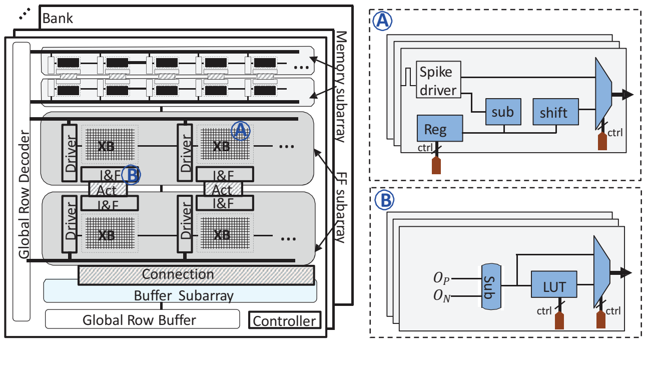

Fig. 10. The ReGAN implementation [13].

TABLE I
COMPARISON OF ReRAM-BASED ACCELERATORS

Accelerator	Architecture Optimization	Implementation	Speedup	Energy Saving
PipeLayer [12]	Data mapping; Inter-layer parallelism	Convolution; activation; pooling	$42.45\times$	$7.17\times$
ReGAN [13]	FCNN computation; D&G co-training; SP&CS	Convolution; activation; pooling; BN	$240\times$	$94\times$

positive subarray and negative subarray are first merged by the subtractor, and then sent to the configurable *look-up table* (LUT), as depicted in Fig. 10Ⓑ.

FC layer in DCGAN is the last layer in D and is the first layer in G (see in Fig. 2). The last layer of D is the flattened version of previous CNN layer and does not require extra computation. ReGAN maps the first layer of G to ReRAM arrays to implement the matrix multiplication.

C. Evaluation of ReRAM based Accelerators

Table I summarizes the architecture design and optimizations in PipeLayer and ReGAN. Both evaluations were compared to the implementation on the state-of-art GPU platform, GTX 1080. PipeLayer used the databases MNIST [21] and ImageNet [22] as the benchmarks. ReGAN selected the following datasets to obtain enough usable data: MNIST [21], cifar-10 [23], celebA [24] and LSUN [25]. Compared to the GPU platform, on average, PipeLayer achieves $42.45\times$ speedup and $7.17\times$ energy saving. Due to the high complexity of GAN system, ReGAN obtains even higher benefit—$240\times$ improvement in performance and $94\times$ energy reduction.

IV. CONCLUSION

Emerging non-volatile *resistive random access memory* (ReRAM) is capable of computation and storage simultaneously, demonstrating great potential in *processing-in-memory* (PIM). In this paper, we discuss novel ReRAM-based accelerator architecture for deep learning. The underlying circuit implementation and architectural optimization are extensively explored to improve the computing efficiency in training and inference procedures. Our previous exploration shows that the in-situ computation property of ReRAM crossbar arrays enables a broader design space for deep learning acceleration and can be further leveraged to construct the future computing platforms.

ACKNOWLEDGMENT

This work was supported in part by NSF 1725456, 1744082, 1253424 and DOE DE-SC0018064. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of grant agencies or their contractors.

REFERENCES

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *NIPS*, 2012, pp. 1097–1105.

[2] V. Sze, Y.-H. Chen, T.-J. Yang, and J. Emer, “Efficient processing of deep neural networks: A tutorial and survey,” *arXiv preprint arXiv:1703.09039*, 2017.

[3] J. Schmidhuber, “Deep learning in neural networks: An overview,” *Neural networks*, vol. 61, pp. 85–117, 2015.

[4] D. H. Woo, N. H. Seong, D. L. Lewis, and H.-H. S. Lee, “An optimized 3d-stacked memory architecture by exploiting excessive, high-density tsv bandwidth,” in *HPCA*. IEEE, 2010, pp. 1–12.

[5] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, “A scalable processing-in-memory accelerator for parallel graph processing,” in *ISCA*, 2015, pp. 105–117.

[6] G. H. Loh, “3d-stacked memory architectures for multi-core processors,” in *ISCA*. IEEE, 2008, pp. 453–464.

[7] H.-S. P. Wong, H.-Y. Lee, S. Yu, Y.-S. Chen, Y. Wu, P.-S. Chen, B. Lee, F. T. Chen, and M.-J. Tsai, “Metal–oxide rram,” *Proc. of the IEEE*, vol. 100, no. 6, pp. 1951–1970, 2012.

[8] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, “Prime: a novel processing-in-memory architecture for neural network computation in reram-based main memory,” in *HPCA*. IEEE Press, 2016, pp. 27–39.

[9] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, “Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars,” in *ISCA*, 2016, pp. 14–26.

[10] A. Radford, L. Metz, and S. Chintala, “Unsupervised representation learning with deep convolutional generative adversarial networks,” *arXiv preprint arXiv:1511.06434*, 2015.

[11] I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin, and A. Courville, “Improved training of wasserstein gans,” *arXiv preprint arXiv:1704.00028*, 2017.

[12] L. Song, X. Qian, H. Li, and Y. Chen, “Pipelayer: A pipelined reram-based accelerator for deep learning,” in *HPCA*. IEEE, 2017, pp. 541–552.

[13] F. Chen, L. Song, and Y. Chen, “Regan: A pipelined reram-based accelerator for generative adversarial network,” in *ASPDAC*, 2018.

[14] L. Song *et al.*, “Graphr: Accelerating graph processing using reram,” in *HPCA*, 2018.

[15] l. Raymond Yeh, *et al.*, “Semantic image inpainting with perceptual and contextual losses,” *arXiv preprint arXiv:1607.07539*, 2016.

[16] l. Ahmed Elgammal, *et al.*, “Can: Creative adversarial networks, generating “art” by learning about styles and deviating from style and deviating from style norms,” *arXiv preprint arXiv:1706.07068*, 2017.

[17] R. Yeh, C. Chen, T. Y. Lim, M. Hasegawa-Johnson, and M. N. Do, “Semantic image inpainting with perceptual and contextual losses,” *arXiv preprint arXiv:1607.07539*, 2016.

[18] A. Elgammal, B. Liu, M. Elhoseiny, and M. Mazzone, “Can: Creative adversarial networks, generating” art” by learning about styles and deviating from style norms,” *arXiv preprint arXiv:1706.07068*, 2017.

[19] T. Salimans, I. Goodfellow, W. Zaremba, V. Cheung, A. Radford, and X. Chen, “Improved techniques for training gans,” in *Advances in Neural Information Processing Systems*, 2016, pp. 2234–2242.

[20] V. Dumoulin and F. Visin, “A guide to convolution arithmetic for deep learning,” *arXiv preprint arXiv:1603.07285*, 2016.

[21] Y. LeCun, “The mnist database of handwritten digits,” *http://yann. lecun.com/exdb/mnist/*, 1998.

[22] O. Russakovsky *et al.*, “ImageNet Large Scale Visual Recognition Challenge,” *IJCV*, vol. 115, no. 3, pp. 211–252, 2015.

[23] A. Krizhevsky and G. Hinton, “Learning multiple layers of features from tiny images,” 2009.

[24] Z. Liu, P. Luo, X. Wang, and X. Tang, “Deep learning face attributes in the wild,” in *Proceedings of the IEEE International Conference on Computer Vision*, 2015, pp. 3730–3738.

[25] F. Yu, A. Seff, Y. Zhang, S. Song, T. Funkhouser, and J. Xiao, “Lsun: Construction of a large-scale image dataset using deep learning with humans in the loop,” *arXiv preprint arXiv:1506.03365*, 2015.