LrGAN: A Compact and Energy Efficient PIM-Based Architecture for GAN Training

Haiyu Mao, *Member, IEEE*, Jiwu Shu, *Fellow, IEEE*,
Mingcong Song, *Member, IEEE*, and Tao Li, *Fellow, IEEE*

Abstract—As a powerful unsupervised learning method, Generative Adversarial Network (GAN) plays an essential role in many domains. However, training a GAN imposes four more challenges: (1) intensive communication caused by complex train phases of GAN; (2) much more ineffectual computations caused by peculiar convolutions; (3) more frequent off-chip memory accesses for exchanging intermediate data between the generator and the discriminator; and (4) high energy consumption of unnecessary fine-grained MLC programming. In this article, we propose LrGAN, a PIM-based GAN accelerator, to address the challenges of training GAN. We first propose a zero-free data reshaping scheme for ReRAM-based PIM, which removes the zero-related computations. We then propose a 3D-connected PIM, which can reconfigure connections inside PIM dynamically according to dataflows of propagation and updating. After that, we propose an approximate weight update algorithm to avoid unnecessary fine-grain MLC programming. Finally, we propose LrGAN based on these three techniques, providing different levels of accelerating GAN for programmers. Experiments show that LrGAN achieves 47.2×, 21.42×, and 7.46× speedup over FPGA-based GAN accelerator, GPU platform, and ReRAM-based neural network accelerator respectively. Besides, LrGAN achieves 13.65×, 10.75×, and 1.34× energy saving on average over GPU platform, PRIME, and FPGA-based GAN accelerator, respectively.

Index Terms—Processing in memory, generative adversarial network, approximate computing, non-volatile memory

1 Introduction

Tremendous success has been fueled by supervised deep learning in image classification, speech recognition, and so on [23], [26], [32], [34], [52], [61], [63]. However, non-trivial amount of training datasets with millions of lables prevents high-accuracy supervised deep learning from being employed in many domains where massive labels are either unavailable or costly to collect through human effort.

By automatically generating richer synthetic datasets without labeling data sets, semi-supervised learning [9], [28] and unsupervised learning [18], [21], [25] are promising to extend the intelligence of deep learning. On the frontier, GAN is the most popular unsupervised learning method, effectively working in many domains, such as video prediction [21], autonomous driving [22] and photo resolution upgrading [35].

Though GAN is powerful to generate items without labeling training sets by human, its network structure is more complex than traditional NN's to efficiently execute on hardware. The generator model and discriminator model of GAN collaboratively work in a minimax manner, to achieve stronger GAN with higher accuracy. To uphold the interaction between the two models, massive amount of intermediate data is required to be communicated between the two models frequently. Since there are quite limited on-chip memory space to store intermediate data, GAN training will introduce additional pressure on off-chip memory accesses, which consume nearly two orders of magnitude more energy than a floating point operation [20]. Thus, these huge data movements become a bottleneck of the system design for GAN.

To solve the memory wall problem in GAN training, researchers proposed ReRAM-based Processing In Memory (PIM) [7], [15], [40], [59], [62], which exhibits energy efficiency in reducing memory access cost compared with CPUs and GPUs. Besides, it can complete a Matrix-Multiply-Vector (MMV) operation in almost only one read cycle with low energy consumption. Since MMV operations dominate the computation patterns in GAN training, ReRAM-based PIM technologies have the potential to reduce memory access cost and accelerate GAN training efficiently.

However, GAN has two main features which are different from traditional neural networks: (1) zero-insertion during training phase; (2) intricate dataflow patterns between the two models. These two features degrade the efficiency of the PIM-based accelerator for GAN. First, zero-insertion adds a heavy burden on storage. Also, I/O traffic becomes the system bottleneck because (1) the interaction between generator and discriminator requires more communication via I/Os in PIM. (2) complex dataflow of GAN exists irregular data dependencies. Therefore, limited I/O bandwidth stalls GAN training. Moreover, ReRAM-based PIM employs Multi-Level Cell (MLC), which consumes a large amount of energy during programming, hindering the low-power ReRAM-based GAN training.

- *Haiyu Mao and Jiwu Shu are with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China. E-mail: mhy15@mails.tsinghua.edu.cn, shujw@tsinghua.edu.cn.*
- *Mingcong Song and Tao Li are with the Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL 32603 USA. E-mail: songmingcong@ufl.edu, taoli@ece.ufl.edu.*

Manuscript received 26 Sept. 2019; revised 14 July 2020; accepted 19 July 2020. Date of publication 22 July 2020; date of current version 8 Sept. 2021. (Corresponding author: Jiwu Shu.)
Recommended for acceptance by A. Karanth.
Digital Object Identifier no. 10.1109/TC.2020.3011122

To address these challenges in PIM-based GAN architecture, we first propose a novel, software-managed Zero Free Data Reshaping (ZFDR) scheme to remove all the zero-related operations produced by GAN. Then, we propose a reconfigurable 3D connection architecture, which not only efficiently fits complex dataflows of GAN, but also supports efficient ReRAM reads and writes and hides the I/O overhead to a great extent. What's more, we propose an approximate weight update algorithm to eliminate the high energy-consuming portion of programming an MLC. Finally, we propose LrGAN[1] (based on LerGAN [48]), a ReRAM-based 3D connection GAN accelerator with low energy consumption, which carefully maps the data processed by ZFDR to the 3D-connected PIM. By doing so, it not only achieves higher I/O performance but also enables I/O connection configuration flexibly for the complex dataflows in GAN training. Experiments show that LrGAN achieves 47.2×, 21.42×, and 7.46× speedup over FPGA-based GAN accelerator, GPU platform, and ReRAM-based neural network accelerator respectively. Moreover, LrGAN achieves 13.65×, 10.75×, and 1.34× energy saving on average over GPU platform, PRIME, and FPGA-based GAN accelerator, respectively.

The main contributions of this paper are as follows:

1) We elaborate three steps of zero-inserting that enable transposed convolution operations in GAN and further analyze the amount of zeros in GAN training. To address problems caused by massive zeros in ReRAM-based PIM, we propose Zero-Free Data Reshaping to remove zero-related operations. ZFDR is flexible to support different paddings, strides and kernel sizes, capable of handling both existing GANs and future GANs with larger stride (e.g., stride of 3).
2) We present the dataflows of training GAN in detail and propose a novel reconfigurable 3D-connected PIM to handle the complicated dataflows. Our 3D connection supports efficient data transferring of both propagation and updating. It is worth mentioning that, to the best of our knowledge, we are the first to study efficient connections in ReRAM-based PIM.
3) We propose an approximate weight update algorithm to avoid energy-inefficient operations in the fine-grain MLC programming. The hardware which supports the approximate update scheme has a negligible modification on the circuit of data-comparison-write.
4) We propose LrGAN based on ZFDR, approximate weight update and 3D-connected PIM. We make slight modifications on the software (via providing interfaces for ZFDR) and memory controller (via creating a finite-state machine for data mapping and configuration of switches) to enable LrGAN to combine ZFDR and 3D-connected PIM well. Also, we enable programmers to use heterogeneous levels of acceleration according to demands.

The rest of this paper is organized as follows. We first introduce ReRAM-based PIM and GAN in Section 2. Then we analyze the challenges of using PIM to accelerate GAN training in Section 3. We present our ZFDR, approximate weight update algorithm and 3D-connected PIM in Section 4. The design of LrGAN is in Section 5. Section 6 evaluates the proposed algorithms, 3D-connected PIM and LrGAN system. Finally, we present related works and conclusions in Sections 7 and 8 respectively.

1. "Lr" comes from removing "o" from "zero" which represents removing 0, changing "z" to "l" to represent shortening wire connection, and deleting "e" to represent energy-saving.

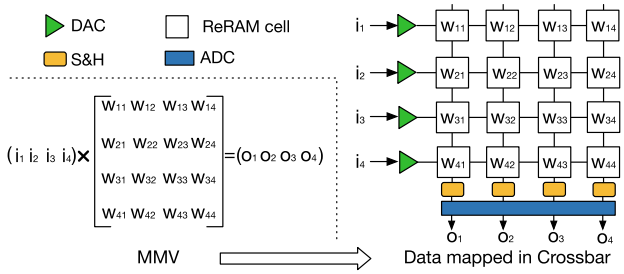


Fig. 1. Mapping MMV to ReRAM crossbar.

2 Background

This section first introduces ReRAM-based PIM and how it can be utilized to implement NNs efficiently, then presents GAN and its features.

2.1 ReRAM-Based PIM

ReRAM stands out from other non-volatile memories (NVMs) since it has high density, relatively low write latency [68], and low write energy [49]. Moreover, it has high endurance ($> 10^{10}$ [36], [37], up to 10^{12} [27], [37], much higher than that of PCM , which is $10^7 \sim 10^8$ [56]). If a network needs to be trained for 10^5 times [43], ReRAM-based PIM can train $10^5 \sim 10^7$ such networks. Due to these benefits of ReRAM, recent studies [14], [15], [59], [62] modify it as the hardware of PIM to accelerate the inference and training of NNs.

ReRAM-based PIM consists of ReRAM arrays and peripheral circuits. Note that, ReRAM arrays can be configured to either support MMVs (called CArrays in this paper), or be used as traditional storage (called SArrays in this paper). When ReRAM arrays are configured as CArrays, they store weights of NNs and conduct MMVs by feeding corresponding inputs (briefly shown in Fig. 1). ReRAM-based PIM also has buffer which is composed of ReRAM cells and connected to CArrays directly. Such buffer is called BArray and enables CArray to access it randomly, hiding the memory access time when performing computation [15]. Equipped with CArrays, BArrays and peripheral circuits to support various basic computations, ReRAM-based PIM can be used to accelerate NNs efficiently.

2.2 Generative Adversarial Network

The Generative Adversarial Network (GAN) consists of two components: a discriminator and a generator. The discriminator learns to decide whether a sample is from the real data set or the generator. The generator aims to generate a sample close to the real data to confuse the discriminator. Therefore, in GAN, the two components play a minimax game to compete with each other iteratively. A minibatch stochastic gradient descent method can be used to train this

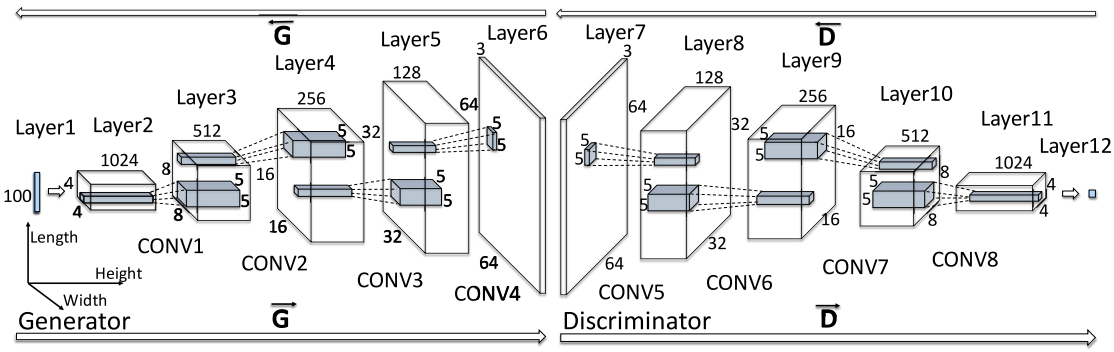


Fig. 2. DCGAN outline.

model, where in each training iteration, a minibatch of m noise samples $\{n_1, n_2, \ldots, n_m\}$ and m true examples $\{x_1, x_2, \ldots, x_m\}$ are sampled from a prior noise distribution $p_e(n)$ and real data distribution $p_d(x)$, respectively. We use $G(n;\theta_g)$ to denote the generative model that generates samples from noises with parameters θ_g and $D(x)$ to denote the discriminative model that represents the probability that x comes from the real data distribution $p_d(x)$. In order to optimize the discriminator, it needs to be updated by ascending its stochastic gradient using Equation (1), which means that the discriminator can assign correct labels to both training examples from D and samples from G. In order to maximize the generator, GAN uses Equation (2) to update it by descending its gradient, which tries to confuse the discriminator to predict the samples as data from the real data distribution. In conclusion, GAN will converge eventually so that the generator can generate an example which is similar to a real one.

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} [\log D(x_i) + \log(1 - D(G(n_i)))] \tag{1}$$

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log(1 - D(G(n_i))). \tag{2}$$

We take the most popular Deep Convolutional Generative Adversarial Network (DCGAN) [57] as an example to further introduce GAN. The framework of DCGAN is shown in Fig. 2. There are some differences between traditional Convolutional Neural Network (CNN) and DCGAN in training phase. In forward propagation phase of discriminator, DCGAN employs strided convolution (S-CONV) instead of pooling. As shown in Fig. 2, the generator has an inverse structure of discriminator, and it employs transposed convolution (T-CONV) in forward propagation phase.

Fig. 3 shows dataflows of training DCGAN and Table 1 shows notations for explanation of training DCGAN. Overall, training DCGAN involves two major parts: one is forward propagation and the other is backward propagation. The backward propagation has two main sub-tasks: error transferring and ∇ weight calculation. When training the discriminator, the generator produces m fake samples using m noises (m is the batch size and a noise (input) is denoted as a vector with 100 elements shown in Layer1 of Fig. 2). This step is denoted by $\overrightarrow{G}$, where DCGAN conducts T-CONV. Then, one batch of real samples and one batch of fake samples are fed into the discriminator. This step is denoted by $\overrightarrow{D}$, where DCGAN conducts S-CONV. Next, DCGAN computes the error of output layer ∇z^L using the loss function Equation (1), where L is the last layer of the discriminator. After that, DCGAN feeds ∇z^L back to the network and begins the backward propagation, which consists of two stages $\overleftarrow{D}$ and $\overleftarrow{D_w}$. First, ∇z^L is fed back layer by layer in $\overleftarrow{D}$ using Equation (3) ($*$ denotes an element-wise multiplication). Therefore, in $\overleftarrow{D}$, the T-CONV takes ∇z^{l+1} and z^l cached by $\overrightarrow{D}$ as inputs then outputs ∇z^l.

$$\nabla z^l = (W^{l+1})^T \nabla z^{l+1} * g'(z^l). \tag{3}$$

Conducting $\overleftarrow{D_w}$ needs ∇z^l transfered by $\overleftarrow{D}$ and the intermediate a^{l-1} cached by $\overrightarrow{D}$. Equation (4) shows the computation in $\overleftarrow{D_w}$, denoted as W-CONV of discriminator since it is different from both S-CONV and T-CONV.

$$\nabla W^l = a^{l-1} \nabla z^l. \tag{4}$$

After $\overleftarrow{D_w}$, the discriminator is updated with ∇W^l. When training the generator, the generator generates m samples and feeds them into the discriminator. After conducting $\overrightarrow{D}$, according to the Equation (2), the error of the output layer in discriminator is sent to $\overleftarrow{D}$. With the intermediate z^l cached by $\overrightarrow{D}$, $\overleftarrow{D}$ can calculate errors and send them to error propagation of generator (denoted as $\overleftarrow{G}$). With ∇z^l sent by $\overleftarrow{G}$ and the intermediate a^{l-1} cached by $\overrightarrow{G}$, $\overleftarrow{G_w}$ can

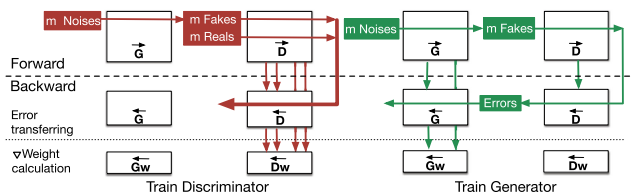


Fig. 3. Dataflows of training discriminator and generator of DCGAN.

TABLE 1
Notations Used for Explanation of Training DCGAN

Symbol	Description
W^l	Kernel weights for lth layer
∇W^l	Derivative of kernel weights for lth layer
z^l	Value of $(W^l)^T x + b$
∇z^l	Derivative of z for lth layer
g	Active function
a^l	Value of $g(z^l)$

TABLE 2
Notations Used for Explanation of Convolution Operations

Symbol	Description
I_w, I_l, I_h	Width, length, height of input
O_w, O_l, O_h	Width, length, height of output
W_w, W_l, W_h	Width, length, height of kernel weight
N_w	Number of kernel weights
S	Stride size of convolution
$S^{'}$	Stride size of converse convolution
P_w, P_l	Padding on width, length
$P_w^{'}$, $P_l^{'}$	Padding on width, length of converse convolution
N_{iz_w}	Number of insert zeros on width
N_{iz_l}	Number of insert zeros on length
N_{zero}	Number of zeros

calculate ∇W^l of generator. After that, the generator is updated with ∇W^l.

3 Challenges

Although GAN has two networks, each of which resembles CNN, it manifests some differences from traditional CNN. In this section, we discuss challenges for PIM-based NN accelerator to execute GAN.

3.1 Redundant Zero-Related Operations

Since DCGAN employs S-CONV, its training introduces considerable zero-insertion, increasing burden on both storage and bandwidth. In order to explain how redundant zeros are introduced and restrain the efficiency, we first introduce some notations used in this paper in Table 2 and take CONV1 of the generator in Fig. 2 as an example of T-CONV. As shown in Fig. 4, $I_w = I_l = 4$ and $O_w = O_l = 8$. The converse convolution of CONV1 is the same as CONV8 in Discriminator, so $S^{'} = 2$, $S = 1$, $P_w^{'} = P_l^{'} = 2$, $P_w = P_l = 2$. Also, CONV1 and CONV8 have the same size of kernel weight. To conduct CONV1, we first insert one zero between every two adjacent input numbers horizontally and vertically (Step 1), then add one zero at the end of input (Step 2) and finally use zero padding of 2 (Step 3). After that, we convolute it with 512 kernels, whose $W_w = W_l = 5$ and W_h is 1,024. Eventually, we obtain an output whose size is $8 \times 8 \times 512$. In this example, we store and transfer 147,456 input values while only 16,384 of them are useful. Moreover, we conduct 1,638,400 multiplications while 295,936 of them are useful, whose efficiency is only 18.06 percent.

In general, $I_w = I_l$, $O_w = O_l$, $P_w = P_l$ and $P_w^{'} = P_l^{'}$. So we denote them as I, O, P and $P^{'}$, respectively, and their

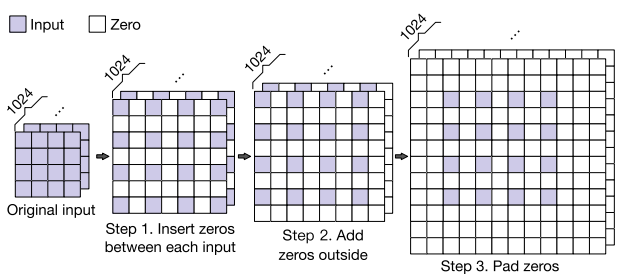


Fig. 4. Steps of adding zeros in inputs of CONV1.

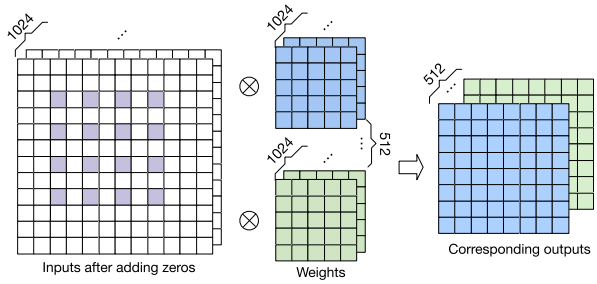


Fig. 5. Convolution on inserted zeros inputs with stride of 1.

relationship is described in Equation (5).

$$\frac{O + 2P^{'} - W}{S^{'}} = (I - 1) \cdots R \quad (R \text{ is the remainder}). \tag{5}$$

Generally, to conduct a convolution in the generator, we first insert $S^{'} - 1$ zeros between every two input numbers, then we add R zeros at the end and finally we use zero padding of P (where $P = W - P^{'} - 1$). Based on the operations above, we can calculate N_{iz_w} and N_{zero}.

$$N_{iz\_w} = N_{iz\_l} = (S^{'} - 1) \times (I - 1) + R \tag{6}$$

$$N_{zero} = (N_{iz\_w} + I_w + P_w) \times (N_{iz\_l} + I_l + P_l) - I_w \times I_l. \tag{7}$$

From Equations (6) and (7) we can observe that with the increase of $S^{'}$ and P, the issue of redundant zeros in T-CONV becomes more severe.

Similar to T-CONV, W-CONV of a generator needs to insert zeros into inputs. However, W-CONV of a discriminator needs to insert zeros to both inputs and kernels. We take a W-CONV connecting Layer11 and Layer10 in Fig. 2 as an example. For simplicity, we take one input feature map to illustrate the difference of zero-insertion between W-CONV and T-CONV in the example.

As shown in Fig. 6, in the forward propagation, given a 8×8 input, we first pad it with 2, then convolve it with a 5×5 kernel, and finally obtain a 4×4 output. In the backward propagation, we denote ∇ Output as dz in Equation (3), whose shape is the same as the output. We first insert zeros to ∇ Output and regard ∇ Output as a kernel weight. Then, we convolute the given 8×8 input with the kernel weight to obtain ∇ Weight.

For W-CONV of the discriminator, the relationship between input and output can be described as Equation (8).

$$\frac{I + 2P - W}{S} = (O - 1) \cdots R \quad (R \text{ is remainder}). \tag{8}$$

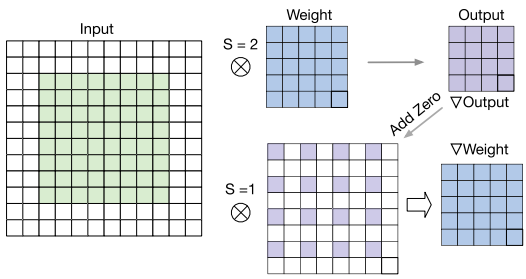


Fig. 6. Example of W-CONV of discriminator.

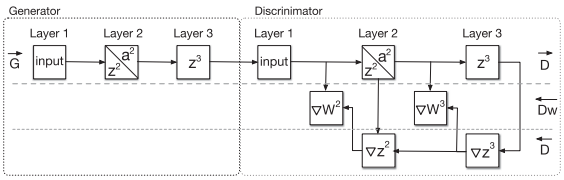


Fig. 7. Dataflow of training discriminator.

Furthermore, the relationship between N_{iz_w} and N_{iz_l} of the kernel weight can be described as Equation (9).

$$N_{iz\_w} = N_{iz\_l} = (S-1) \times (O-1) + R. \tag{9}$$

According to Fig. 6, N_{zero} in W-CONV of the discriminator equals to the sum of the number of zeros used for input padding and the number of zeros used for ∇ insertion. It can be described using Equation (10).

$$\begin{aligned} N_{zero} = &[(N_{iz\_w} + O_w) \times (N_{iz\_l} + O_l) - O_w \times O_l] \\ &+ [(I_w + P_w) \times (I_l + P_l) - I_w \times I_l]. \end{aligned} \tag{10}$$

For W-CONV of the discriminator, N_{zero} also increases either S or P increases according to Equations (9) and (10).

3.2 Inefficient I/O Connection

For training where massive memory reads/writes are required to update kernel weights, PipeLayer [62] employs efficient H-tree wire routing. However, the dataflows of GAN training are more complicated than that of traditional NNs. We take a simple GAN (3-layer generator and 3-layer discriminator) as an example to show details of dataflows (training discriminator in Fig. 7 and training generator in Fig. 8). Thus, if we train a GAN by mapping phases to H-tree connection architecture, it will experience a large number of long routings.

Fig. 9 shows two GAN examples N_1, N_2 training on the H-tree routing banks. Each bank has 16 tiles and each tile is composed of several CArrays, BArrays and SArrays. There are two kinds of routing nodes: (1) multiplexing node, connecting data wires of the same width; (2) merging node, through which the width of data wire is divided into two halves. In the examples shown in Fig. 9, N_1 is a relatively small GAN, while N_2 may be a bigger GAN or a small GAN with high parallelism (i.e., duplicating kernel weights for several times). In other words, the space utilized by training a GAN is decided by the size of GAN itself and the number of kernel weight duplications. When we map a GAN, we can separately training discriminator and generator as N_1 shows. This introduces more space while reduces total data movements compared with the map without duplication like the mapping pattern of N_2. However, all of these mapping patterns suffer from long routings, as examples

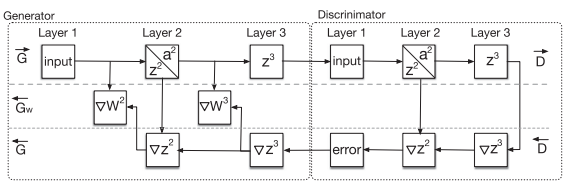


Fig. 8. Dataflow of training generator.

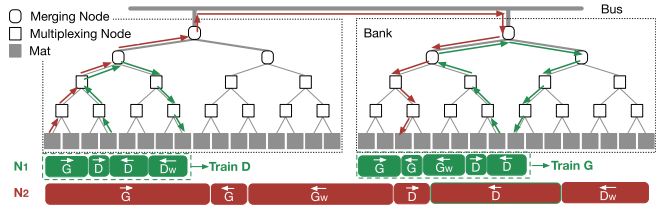


Fig. 9. Networks mapped to H-tree connected tiles.

marked in green and red arrows shown in Fig. 9. With network size and number of duplications increasing, this problem becomes more severe. We can relieve this problem by adding some connections between the routing nodes whose parent nodes are different, as the connection pattern used in by MAERI [33]. Since the dataflow of GAN training is much more complicated, simply doing so will not achieve desirable performance of speedup.

3.3 High Energy-Consuming Write Scheme of MLC

ReRAM-based PIM [14], [15], [59], [62] employs MLC instead of SLC for the following three reasons. (1) Using SLC-based NVM array to conduct a vector-matrix multiplication requires more time for intermediate data processing. (2) The SLC-based array not only has latency overhead for processing intermediate data but also introduces larger peripheral circuit to process intermediate data. (3) MLC-based NVM has higher memory density than MLC-based NVM.

Although MLC has these three advantages over SLC, its programming is more complex than that of SLC, which only conducts a single SET or RESET operation. Fig. 10 depicts the procedure for programming an MLC, which is also known as "Program-and-Verify" (P&V). First, a SET-sweep pulse is applied to program the cell to its lowest resistance state. This is followed by a RESET to initialize the cell to a total RESET state. Next, P&V applies an SET pulse and then verifies that a specified resistance has been achieved, iteratively. The protracted programming procedure mentioned above accounts for MLC's long write latency and high energy consumption. Based on this programming scheme, we can also figure out that altering values varies time and energy consumption.

Table 3 is retrieved from [53]. It records the worst case of latency and the average energy consumption for programming 3-bit MLC (eight resistance states). An MLC costs more time and energy to program a cell to middle resistance states (e.g., state 3, 4 in 3-bit MLC) since it requires additional tuning iterations. Moreover, time and energy consumption spike with 4-bit MLC due to the finer-grained tuning procedure. Unfortunately, training GAN requires cells with more resistance levels, which improves

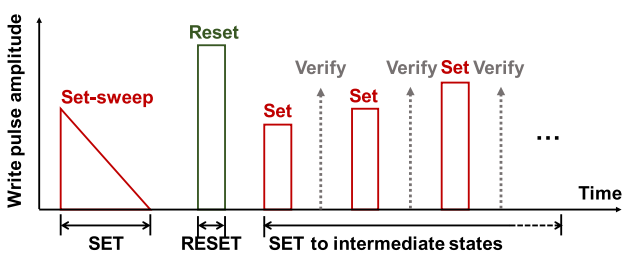


Fig. 10. Programming scheme of MLC.

TABLE 3
Programming Latency and Energy of MLC ReRAM Cell

Target	0	1	2	3	4	5	6	7
$T_{WC}(ns)$	15.2	46.8	98.3	143	150	101	52.7	12.1
$E_{ave}(pJ)$	2.0	6.7	19.3	35.1	35.6	19.6	8.5	1.5

performance and memory density. Since NVM-based PIM lacks flexibility when compared with FPGA and GPU, it should ensure both high performance and low energy consumption. Thus, MLC's energy inefficient write scheme has become a challenge for GAN training with ReRAM-based PIM.

4 Our Proposed Solutions

In this section, we propose our solutions to address the three challenges analyzed in Section 3.

4.1 PIM-Based Zero-Free Scheme

In order to address the problem mentioned in Section 3.1, we propose a novel software managed, memory controller supported scheme called Zero-Free Data Reshaping to remove zero operations. This scheme consists of two components: (1) *T-CONV ZFDR* for T-CONVs; (2) *W-CONV-S ZFDR* for W-CONV of stride convolution.

We first take CONV1 (Fig. 5) as an example to explain our T-CONV ZFDR scheme. We usually convert convolutions into MMVs in PIM-based computation, so we first reshape kernel weights into vectors. The reshape operation is different from the general one since we only extract kernel weights that multiply non-zero inputs, as shown in Fig. 11. After reshaping all the 512 weight kernels into a 512×4096 matrix, we map this weight matrix into the Carray and feed the corresponding 4,096 inputs, then we obtain 512 results. All of above operations correspond to one convolution operation with 512 kernel weights. After the first convolution operation shown in Fig. 11, we slide kernel weights with stride of 1. When sliding, the useful kernel weights change. Fig. 12 gives an example of how useful kernel weights change when sliding. Thus, in Step 3, the weight matrix can be reused since it is the same with that in Step 1. We find that some reshaped weight matrices are reused when kernels slide on the edge of input map and more reshaped weight matrices are reused when kernels slide inside the input map.

In summary, we store 25 kinds of reshaped weight matrix in this case (also the same in CONV2, CONV3 and CONV4). Notwithstanding this ZFDR scheme introduces more space to store weights, it improves parallelism greatly.

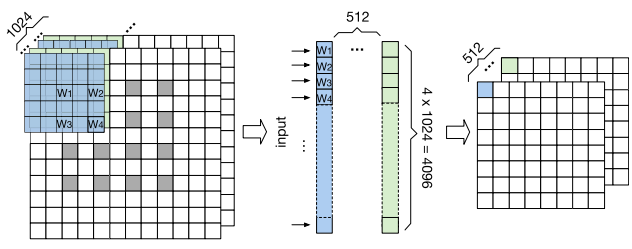


Fig. 11. Example of zero free data reshaping.

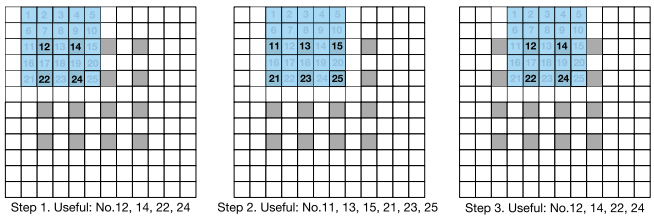


Fig. 12. Example of how useful weights change when sliding.

For example, it only needs 9 cycles (one MMV uses one cycle) to complete CONV1. While without ZFDR, it will take 64 cycles. Moreover, if we duplicate kernel weights directly (without ZFDR), and we want to conduct CONV1 in 9 cycles, we need to store at least 179,200 weights. It means that in order to achieve the same performance as ZFDR, duplicating weights directly not only consumes 75 percent more storage, but also transfers $9\times$ inputs.

In order to extend our ZFDR scheme to a general case, we first define the Loop Length (LL) using the following equation.

$$LL = \begin{cases} I \times S^{'} + (S^{'} - 1) & P \geq S^{'} - 1 \\ I \times S^{'} & P < S^{'} - 1, P + R \geq S^{'} - 1 \\ I \times S^{'} - (S^{'} - 1) & P < S^{'} - 1, P + R < S^{'} - 1 \end{cases}. \tag{11}$$

Then we divide the T-CONV ZFDR scheme into three cases as follows. *Case 1: Reshape kernel weights that conduct convolution on the corner of input map.* This case has $((I-1) \times S^{'} + 1 + R + 2P - LL)^2$ sets of reshaped weights, and each kind of weights is non-reusable. *Case 2: Reshape kernel weights that conduct convolution on the edge of input map.* We define R_1, R_2 using Equations (12) and (13)

$$R_1 = \begin{cases} P & P < S^{'} - 1 \\ P - (S^{'} - 1) & else \end{cases} \tag{12}$$

$$R_2 = \begin{cases} (P + R) - (S^{'} - 1) & P + R \geq S^{'} - 1 \\ P + R & else. \end{cases} \tag{13}$$

Then number of reshaped kernel weights in this case is $R_1 \times S^{'} \times 2 + R_1 \times S^{'} \times 2$, and each reshaped kernel weight can be reused by t times ($t \in \{\left\lfloor \frac{LL-W+1}{S^{'}} \right\rfloor, (\left\lfloor \frac{LL-W+1}{S^{'}} \right\rfloor + 1)\}$). *Case 3: Reshape kernel weights that conduct convolution inside the input map.* This case has $S^{'} \times S^{'}$ reshaped weights, and each reshaped weight can be reused by t times ($t \in \{\left\lfloor \frac{LL-W+1}{S^{'}} \right\rfloor^2, (\left\lfloor \frac{LL-W+1}{S^{'}} \right\rfloor + 1)^2, \left\lfloor \frac{LL-W+1}{S^{'}} \right\rfloor \times (\left\lfloor \frac{LL-W+1}{S^{'}} \right\rfloor + 1)\}$).

The pattern of W-CONV-S ZFDR is similar to that of T-CONV ZFDR. The difference is, for W-CONV of stride convolution, we remove zeros from ∇ output, reshape it as weight, then conduct convolution on input map to receive ∇ weight. W-CONV-S ZFDR has three cases as follows. *Case 1: Reshape zero-insertion ∇output that conducts convolution at the corner of input map.* This case has $\lceil \frac{P}{S} \rceil^2 + \lceil \frac{P-R}{S} \rceil^2 + 2\lceil \frac{P}{S} \rceil \lceil \frac{P-R}{S} \rceil$ number of reshaped ∇ outputs and each of them is non-reusable. *Case 2: Reshape zero-insertion ∇ output that conducts convolution on the edge of input map.* This case has $2\lceil \frac{P}{S} \rceil + 2\lceil \frac{P-R}{S} \rceil$ number of reshaped ∇ outputs, and each of them can be reused by $I - (O-1)S$ times. *Case 3: Reshape*

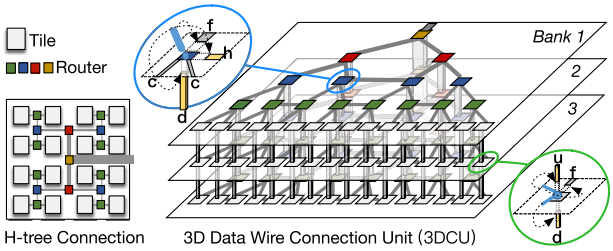


Fig. 13. 3D connection based on original H-tree connection.

zero-insertion ∇ output that conduct convolution inside the input map. This case has only one zero-insertion ∇ output whose size is equal to ∇ output, and it can be reused by $[I-(O-1)S]^2$ times.

Since both T-CONV ZFDR and W-CONV-S ZFDR have three similar types, we name them as $CornerReshape$, $EdgeReshape$ and $InsideReshape$ respectively. Note that $CornerReshape$ has no reuse of reshaped weights while $InsideReshape$ tends to have more reuses than $EdgeReshape$ does. This involves an unbalance in runtime because $InsideReshape$ takes a long time to execute while $CornerReshape$ is idle in most of the time. Such unbalance not only exists in the executing stage, but also in the I/O transmission, because I/O connected to $InsideReshape$ is busy while that connected to $CornerReshape$ is slack. In order to address this problem, we duplicate $EdgeReshape$ and $InsideReshape$ for R_e times and R_i times respectively.

4.2 3D-Connected PIM for GAN Training

In order to solve the problem elaborated in Section 3.2, we propose a 3D-connected PIM, aiming to efficiently fit dataflows of GAN training.

Fig. 13 shows an original H-tree data wire connection in a bank with 16 tiles (light grey squares). Green and blue squares are multiplexing nodes, while red and yellow squares are merging nodes. To better illustrate our 3D connection architecture, we draw the connections as a binary tree and mark different connection layers with different colors. First, we add wires between two nodes whose parent nodes are different in one layer, such as the wire between the middle two blue nodes shown in Fig. 13. Then we pile up three banks and add vertical wires between two corresponding nodes. For each two vertical connected nodes, the width of wire between them is the same as the width of wire connected to their parent nodes. Due to the pin bandwidth limitation, we modify the routers by adding switches. We take two nodes as examples shown in Fig. 13 (original wires are colored grey and added wires are in yellow). For the node circled in blue, it has one switch, which can connect $wire\ h$, $wire\ d$ or $wire\ f$, and two wires connected to child nodes are fixed as original. For the light gray node circled in green, it has two switches, which can connect $wire\ u$, $wire\ d$ or $wire\ f$. Note that, only nodes in $Bank\ 2$ have two switches, which enable the nodes to connect both upper/down nodes at the same time. We create a state set s_set for each switch, and we have $s\_set \subseteq \{parent,\ horizontal,\ upper, down\}$. Moreover, we add an adder into the each node, which can be also bypassed. Thus, we build a 3D data wire connection unit (3DCU), which can be configured into two modes: $Smode$ for normal memory read/write and $Cmode$ for computing. In $Smode$, the connections are static and configured as H-tree pattern. While in $Cmode$, the connections are dynamically reconfigured according to dataflows.

With 3DCU, we can build our 3D-connected PIM for training GANs. Fig. 14 ellaborates how to use 3DCUs to train a GAN. First, we connect two 3DCUs ($\{B_1, B_2, B_3\}$, $\{B_4, B_5, B_6\}$) together. Banks in these two 3DCUs are all connected to the bus in traditional way. Moreover, B_1 and B_4, B_3 and B_6 can be connected to each other directly, bypassing the bus and CPU.

After connecting two 3DCUs, we first present the way of training discriminator in Fig. 14. Note that we only present the critical concept paths, omitting other paths like data transferring of ∇ weight calculation inside the bank. When training discriminator, B_2 and B_3 are not used and stay in $Smode$, working as traditional memory. We first map $\overrightarrow{G}$ to B_1 and $\overrightarrow{D}$ to B_4. After that, we configure $\{B_1, B_4, B_5, B_6\}$ into $Cmode$. We show the dataflows of training discriminator with P_x (P reperesents the point marked on dataflows in Fig. 14, x is the number of the point). $P_1 \rightarrow P_2$ is the dataflow of $\overrightarrow{G}$, and the zigzag line represents that during $\overrightarrow{G}$, we may transfer data from one tile to another tile through horizontal connections. $P_2 \rightarrow P_3$ transfers outputs of generator to discriminator through the bypass bus connection. $P_3 \rightarrow P_4$ shows the dataflow of $\overrightarrow{D}$. During $P_3 \rightarrow P_4$, when we complete the computation of one layer, we map the corresponding part of $\overleftarrow{D_w}$ and $\overrightarrow{D}$ to B_5 and B_6 respectively. Note that we continue forward propagation of the discriminator when we map $\overleftarrow{D_w}$ and $\overrightarrow{D}$. For example, we conduct $P_{11} \rightarrow P_{12}$ and $P_9 \rightarrow P_8$ simultaneously. We start the backward propagation by transferring error from P_4 to P_5. During the backward propagation, we

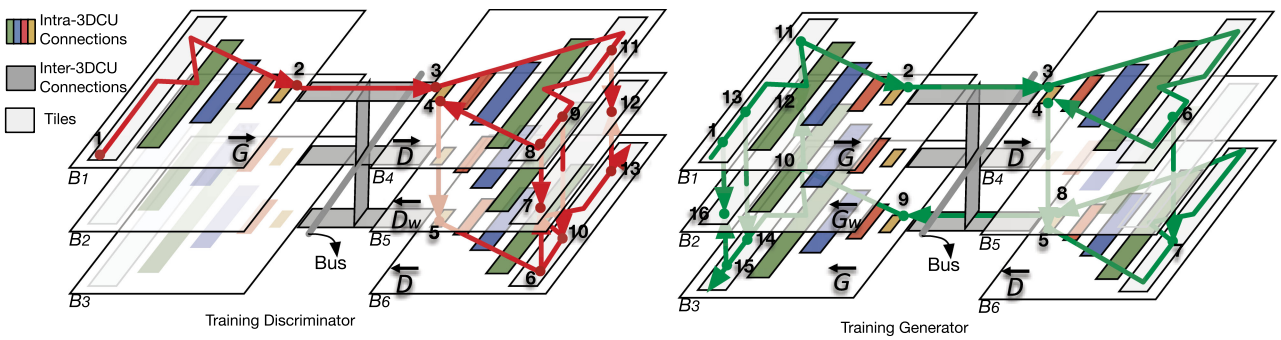


Fig. 14. Dataflows of GAN training using 3DCUs.

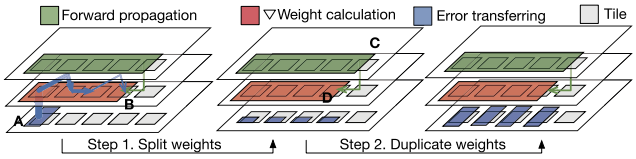


Fig. 15. Data mappings on 3D-connected PIM.

need the results from both $\overrightarrow{D}$ ($P_8 \rightarrow P_7$, $P_{11} \rightarrow P_{12}$) and $\overleftarrow{D}$ ($P_6 \rightarrow P_7$, $P_{13} \rightarrow P_{12}$) to conduct $\overleftarrow{D_w}$. Also, we need the result from $\overrightarrow{D}$ ($P_9 \rightarrow P_{10}$) to conduct $\overleftarrow{D}$. After backward propagation, we configure $\{B_4, B_5, B_6\}$ into $Smode$. Through reading B_5 and some calculations in CPU, we update discriminator by writing new kernel weights to B_4.

The right part of Fig. 14 illustrates the dataflows of training generator. Note that, after training discriminator, B_1 is in $Cmode$, while others are in $Smode$. Thus, we first switch others to $Cmode$. At the same time, we can conduct $\overrightarrow{G}$ shown as $P_1 \rightarrow P_2$, and map $\overleftarrow{G_w}$, $\overleftarrow{G}$ to B_2, B_3 simultaneously. Then we output results of $\overrightarrow{G}$ to $\overrightarrow{D}$ marked as $P_2 \rightarrow P_3$ and start $\overrightarrow{D}$ through $P_3 \rightarrow P_4$. Simultaneously, we map $\overleftarrow{D}$ to B_6. After that, we start backward propagation by transferring error from P_4 to P_5. The error is transferred to generator through $P_5 \rightarrow P_8 \rightarrow P_9$, and during this period, the result in $\overrightarrow{D}$ is used for $\overleftarrow{D_t}$ such as $P_6 \rightarrow P_7$. After transferring error to $\overleftarrow{G}$, we start $\overleftarrow{G}$ and $\overleftarrow{G_w}$ in an interleaving way. Similar as dataflows in backward propagation of discriminator, we need $P_{11} \rightarrow P_{12}$ and $P_{10} \rightarrow P_{12}$ to conduct $\overleftarrow{G_w}$ first and then we need $P_{13} \rightarrow P_{14}$ for $\overleftarrow{G}$. Afterwards, we use $P_1 \rightarrow P_{16}$ and $P_{15} \rightarrow P_{16}$ to complete $\overleftarrow{G_w}$. Finally, in the same way of updating discriminator, we switch $\{B_1, B_2, B_3\}$ to $Smode$ and update generator.

In general, we map generator to one or several 3DCUs and map discriminator to corresponding 3DCUs connected to generator. The top layer is usually for forward propagation and the second, third layers are usually for ∇ weight calculation, error transfer respectively. We locate ∇ weight calculation in the second layer since it needs data transferred from either phases, while error transfer only needs data from forward propagation. What's more, in order to reduce data movement, we should make sure each part of phase is vertical alignment. Take computation between $Layer1$ and $Layer2$ in Fig. 8 as an example. The left figure shown in Fig. 15 is an original way of data mapping. The green and red parts are bigger than the blue one, because we apply ZFDR scheme on them, duplicating kernel weights for several times. For the blue one, it applies the normal kernel weight mapping pattern. This naive data mapping introduces non-negligible data movements, like blue lines marked in the left figure. We can solve this problem by splitting kernel weights and enable each part to handle corresponding vertical partial results (shown in the middle figure of Fig. 15). Thus, we only need small-step data movements like $C \rightarrow D$. It's worth mentioning that green parts, red parts and blue parts are not vertical alignment perfectly. They may have small-step data movements horizontally, but it's much better than original data mapping shown in the left figure. The method in Step 1 is space-saving but less parallelism. Also, we can duplicate weights after splitting, like Step 2 shows. This improves the parallelism but turns out to be space consuming. The detailed design will be introduced in Section 5.

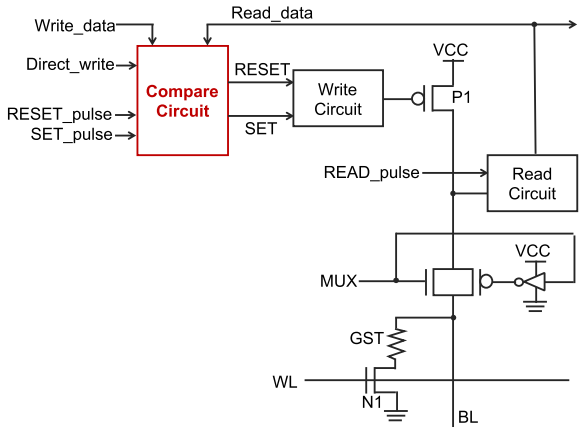


Fig. 16. Circuit of approximate update.

4.3 Approximate Weight Update of GAN Training

In this sub-section, we propose an approximate weight update algorithm to mitigate the problem of high overhead caused by MLC programming (Section 3.3).

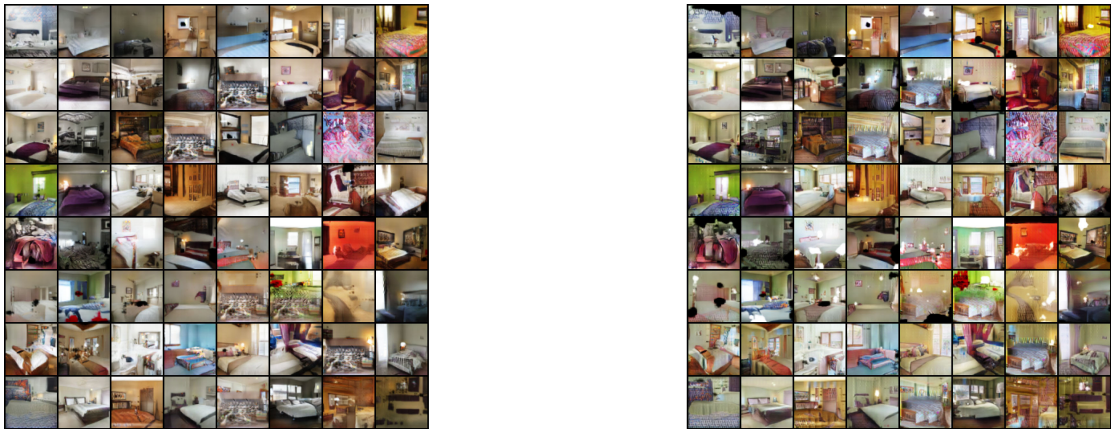
First, we are glad to observe that GANs can adapt to approximate computing. When we modify the low-significance bits of some weight values during GAN training, the quality of the samples generated by the trained generator remains stable. For example, we first train DCGAN in the standard procedure with the dataset *bedroom* in Lsun [70]. We use a 16-bit precision value to train the discriminator and generator iteratively. We train them 25 times with 50,000 iterations in each training round. Then we modify 20 percent weight values by changing their four low-significant bits randomly during the DCGAN training procedure. The pictures of bedrooms generated by these two trained generators are shown in Fig. 17. The pictures generated from the modified training (Fig. 17b) and pictures generated from the standard training (Fig. 17a) look similar. To quantitatively compare these two trained GANs, we introduce *FID* in [46], a score to quantify the quality of the pictures generated. Experiments show that the difference between the *FID* of the original DCGAN and the *FID* of the modified-trained DCGAN is only 0.3. This means that the modified-trained DCGAN is virtually as good as the original one.

Algorithm 1. Approximate Weight Update

Input: permission of approximate weight update to the bitline: $permission$, set of values to be approximated: $[t_{down}, t_{up}]$, current value: c_value, value to write: w_value

if w_value == c_value **then**
  Skip the write operation;
else
  if $permission$ == *FALSE or* $w\_value \notin [t_{down}, t_{up}]$ **then**
    Conduct normal MLC-write operation;
  else
    Conduct SET-sweep and RESET operations;
    while $c\_value \notin [t_{down}, t_{up}]$ **do**
      Conduct SET and VERIFY operations;

With training modified, we propose *Approximate Update*, an approximate weight update algorithm for GAN training. The key idea of *Approximate Update* is to avoid the energy-



(a) Pictures Generated under Normal Training. (b) Pictures Generated under Modified Training.

Fig. 17. Comparison of pictures generated by the normal training and the training with modifying 4 low significant bits of 20 percent weight values.

consuming SET operations in MLC writings without diminishing GANs' accuracy. Algorithm 1 shows the pseudo-code of the approximate weight update.

The *Approximate Update* works with the $permission$ indicating whether the bitline allows approximate computing. This is decided by the significance of the values. The $[t_{down}, t_{up}]$ is set according to the pre-trained results to ensure the GAN's accuracy. In Algorithm 1, lines 1-2 show the data-comparison-write module, which skips the write when the writing value equals the value of the cell. Then, lines 4-5 describe the conditions in which *Approximate Update* cannot be performed. Finally, lines 7-8 illustrate the procedure of *Approximate Update*, in which $[t_{down}, t_{up}]$ includes values that require more fine-grained SET operations. The *Approximate Update* eliminates this time- and energy-intensive procedure by introducing crose-grained SET operations to enable c_value in $[t_{down}, t_{up}]$ instead of forcing $c\_value = w\_value$.

Fig. 16 depicts the circuit supporting *Approximate Update*, which is built on the data comparison write circuit. The red part of Fig. 16 is the only part that differs from the data comparison write, because, in addition to determining whether two values are equal, it also judges whether the current value is in the given interval. Furthermore, the *Direct_write* signal is controlled by line 4 in Algorithm 1.

5 LrGAN Design

In this section, we present how the Zero-Free Scheme in Section 4.1 and 3D Connected PIM in Section 4.2 work together in LrGAN. Fig. 18 elaborates the outline of LrGAN design in five parts.

Program. In the program stage, we program a network, describing it layer by layer. For example, in the lth layer, we use the size of input ($input\_size\_l$), size of kernel weight ($weight\_size\_l$) and size of output ($output\_size\_l$) to describe it. Moreover, $stride$ includes the stride of generator and stride of discriminator and so does $padding$. Structure $replica\_degree$ describes the degree of duplication in each phase of training GAN. It has three degrees, low, $middle$ and $high$. Programmers can easily use these three parameters which represent low to high parallelisms, without knowing how to duplicate kernel weights to increase parallelism, which will be performed by the compiler.

Interface. We realize ZFDR by providing two interfaces. One is $ZFDR\_T$ for T-CONV ZFDR and the other is $ZFDR\_WS$ for W-CONV-S ZFDR. These two functions do not reshape data directly but create place holders and dataflows for further removing zeros, just like the way of traditional NN frameworks. Their parameters are passed from

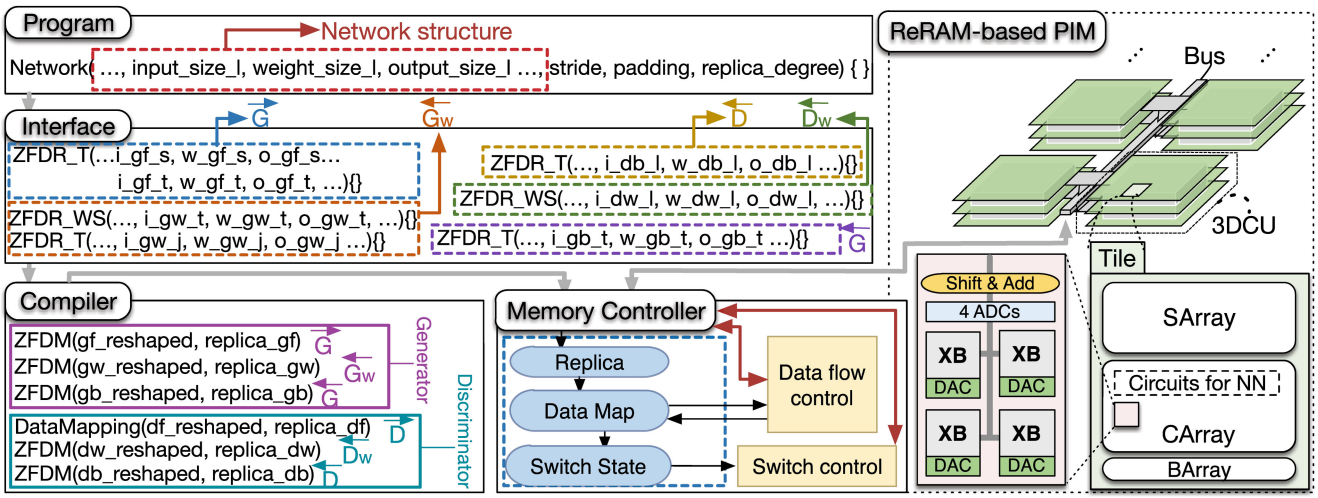


Fig. 18. Outline of LrGAN (an architecture combined techniques of $ZFDR$ and 3DCUs).

TABLE 4
Value of $replica\_gf$

Value \ Part / Level	$replica\_c$	$replica\_e$	$replica\_i$
low	1	1	$replica\_e_{max}$
$middle$	1	$replica\_e_{max}$	$replica\_e_{max}$
$high$	1	$replica\_e_{max}$	$replica\_i_{max}$

programming a network. These two functions also process the network layer by layer. The interface component in Fig. 18 shows the most complex situation: the generator of this GAN has both T-CONV and S-CONV, and the discriminator has T-CONV. The generator needs $ZFDR\_T$ for $\overrightarrow{G}$ (marked in blue), both $ZFDR\_T$ and $ZFDR\_WS$ for $\overleftarrow{G_w}$ (marked in orange), and $ZFDR\_T$ for $\overleftarrow{G}$ (marked in purple). The discriminator needs $ZFDR\_T$ for $\overleftarrow{D}$ and $\overleftarrow{D_w}$ (marked in yellow and green). Under normal situation where the generator has T-CONV and the discriminator has S-CONV, $ZFDR\_T$ is needed for $\overrightarrow{G}$, $\overleftarrow{G_w}$ and $\overleftarrow{D}$, and $ZFDR\_WS$ is for $\overleftarrow{D_w}$.

Compiler. After reshaping data, we start to map them through a compiler. Mapping data has two parts. One is mapping generator and the other is mapping discriminator. In the case of the generator with both T-CONV and S-CONV, we map $\overrightarrow{G}$, $\overleftarrow{G_w}$, $\overleftarrow{G}$, $\overleftarrow{D_w}$ and $\overleftarrow{D}$ by using Zero Free Data Mapping scheme (ZFDM), while we use normal data mapping scheme (DataMapping) to map $\overrightarrow{D}$ (shown in the compiler component of Fig. 18). In the case of the generator with only T-CONV, we use $DataMapping$ for $\overleftarrow{G}$ and $\overrightarrow{D}$, and ZFDM for the remaining phases. ZFDM has two main parameters: data reshaped by ZFDR and the number of replicas transferred from programming.

We take $\overrightarrow{G}$ to further elaborate ZFDM scheme. $gf\_reshaped$ is data reshaped by ZFDR during generator forward propagation. $replica\_gf$ is a vector which records the number of replicas in $CornerReshape$, $EdgeReshape$ and $InsideReshape$. We name items in $replica\_gf$ as $replica\_c$, $replica\_e$ and $replica\_i$. Also, we calculate the average reuse time of each case and name them as $reuse\_c$, $reuse\_e$ and $reuse\_i$. We do this because the reusing time of weights inside each case shows little difference. Assume that the time MMV consumed in CArray is t_m, then the total time of computation t_{c_total} in a layer is $t_m \times \frac{reuse\_i}{replica\_i}$ (the execution time of parallel tasks is decided by the longest task). We assume the time of transferring data from one tile to its neighbor is t_t, then transferring results of a layer to its next layer consumes at least $(\left\lceil \frac{layer\_size}{CArray\_size} \right\rceil - 1) \times t_t$, named t_{t_total} ($layer\_size$ is decided by $replica\_c$, $replica\_e$ and $replica\_i$). We fix $replica\_c$ as 1 since $reuse\_c$ is 1, and define the maximum value $replica\_e_{max}$, $replica\_i_{max} = LL \times replica\_e_{max}$ to let $t_{t\_total} \leq t_{c\_total}$ (LL is the loop length defined in Section 4.1). Based on parameters defined above, we can define $replica\_gf$ as Table 4 shows.

To summarize, we duplicate kernel weights considering three factors: *(1) Programmers' demand (space demands).* When the free space is small or programmers would like to use small memory space to train a GAN, they can set $replica\_degree$ as low, and vise versa. *(2) Improving the performance.* More replicas indicate higher parallelism, which means higher performance. *(3) Avoiding I/O to become a bottleneck.* More replicas may incur more communications among tiles, so we must avoid heavy communications from hindering performance. For other phases in ZFDM, parameters can be obtained in the same way of $\overrightarrow{G}$ does.

Then we take $\overrightarrow{D}$ to further introduce $DataMapping$ scheme. $df\_reshaped$ is data reshaped by normal reshaping scheme during forward propagation phase of discriminator. For $replica\_df$, we define it as Equation (14) shows, where s_{zf} is size of kernel weights after duplication in $\overrightarrow{D}$ and s_n is size of kernel weights before duplication in $\overrightarrow{D}$.

$$replica\_df = \begin{cases} 1 & replica\_degree = low \\ \left\lfloor \frac{s_{zf}}{2 \times s_n} \right\rfloor & replica\_degree = middle \\ \left\lfloor \frac{s_{zf}}{s_n} \right\rfloor & replica\_degree = high \end{cases}. \tag{14}$$

Memory Controller. Memory controller records the information transferred from the compiler, such as number of replicas and data mappings. What's more, it records states of switches, which are deduced by data mappings. These records come into a finite state machine, marked in blue rectangle in Memory Controller (Fig. 18). The finite state machine offers states for dataflow controller and switch controller to control 3DCUs. Also, these two controllers receive signals from 3DCUs and update the finite state machine. Thus, the memory controller can manage the data mapping and configure switches according to the dataflows dynamically.

ReRAM-Based PIM. The part communicating with memory controller is ReRAM-based PIM. It is also the main hardware that supports our LrGAN. It is configured with several 3DCU pairs introduced in Fig. 14 in Section 4.2. Each tile in 3DCU contains $SArray$, $CArray$ and $BArray$, using the design in PRIME [15], which has been already introduced in Section 2.1. The ReRAM crossbars in a $CArray$ (marked in light pink in Fig. 18) employ the design of that in ISAAC [59], since they can support 16-bit precision data while PRIME can not. Based on the tile equipped with basic NN computation and storage ability, our proposed 3DCU pairs can work well.

6 Evaluation

In this section, we first introduce our experimental setup and benchmarks used to evaluate the proposed designs. We then present our evaluation results in terms of performance, energy, and overhead.

6.1 Experimental Setup

We compare LrGAN with (1) GANs running on GPU platform; (2) FPGA-based GAN accelerator [50]; and (3) GANs running on modified ReRAM-based NN accelerator: PRIME [15]. We use the NVIDIA Titan X as our GPU platform and choose the Xilinx VCU118 board for implementing FPGA-based GAN accelerator. The hardware configurations we used for PRIME and LrGAN are listed in Table 5. The configurations of ReRAM are from [16], [53], [69].

For LrGAN configuration, we use 4-bit for each ReRAM cell, and 16-bit for input, weight and output (i.e., same as [62]). The size of ReRAM array is 128×128 cells. We configure half of a tile for CArray (64 MB), 1/64 of the tile for BArray (2 MB) and the remaining 62 MB for SArray. We use

TABLE 5
Hardware Configurations

Host Processor		Intel Xeon CPU E5520, 2.27GHz, 4 cores
L1 I/D cache		32KB/32KB; 4-way; 2 cycles access
L2 cache		256KB; 8-way; 10 cycles access
ReRAM-based Main Memory	**Overview**	TaO_x/TiO_2-based ReRAM 16GB; 2GB per bank, 128MB per tile;
	Bank	SET/RESET/ latency: 100/34 ns; read latency: 16 ns SET/RESET energy: 9/10 pJ read energy: 2 pJ
	H-Tree	29.9ns latency, 386pJ energy
I/O Frequency		1.6GHz

CACTI-6.5 [51], CACTO-IO [29] to model our interconnects and off-chip connects respectively.

6.2 Benchmarks

We employ 8 state-of-the-art GAN networks as our benchmarks, shown in Table 6. To describe the topologies of GANs, we use f, c and t to denote fully-connected, convolution and transposed convolution layers respectively. For example, the $512c5k2s$ denotes a convolution layer with 512 input feature maps, using $5 \times 5 \times 512$ kernels with a stride of 2, while $2s$ in $512t5k2s$ denotes a transposed convolution layer with a stride of 1/2. The $100f$ denotes a fully-connected layer with 100-unit input and $f1$ denotes a fully-connected layer with 1-unit output. The $t3$ represents that after T-CONV, there are 3 output feature maps. For simplicity, if several layers share the same size of kernel or stride, we consolidate those common factors at the end, for example $100f$-($1024t$-$512t$-$256t$-$128t$)($5k2s$)-$t3$, where layers $1024t$, $512t$, $256t$, and $128t$ share the common kernel size of 5 and stride size of 2.

6.3 Results

We fully train the networks in Table 6 with the batch size of 64, and the results are shown as follows.

We first examine the effectiveness of our proposed $ZFDR$ and 3D connection mechanisms. We then compare the performance and energy between LrGAN and alternative PIM design such as PRIME. Moreover, we compare LrGAN with FPGA-based GAN accelerator and GAN running on GPU platform. Note that we use 2D and 3D to represent H-tree and 3D connection design, respectively, and investigate configurations with different degrees of duplication (i.e., low, middle and high).

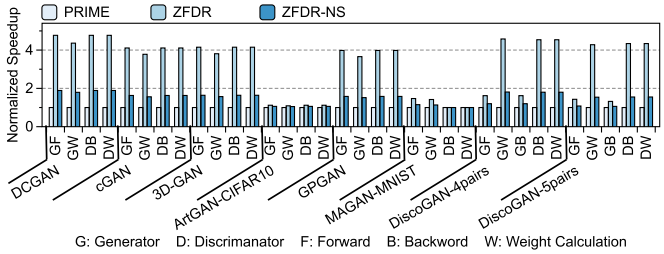


Fig. 19. Performance comparison on phases used ZFDR with PRIME.

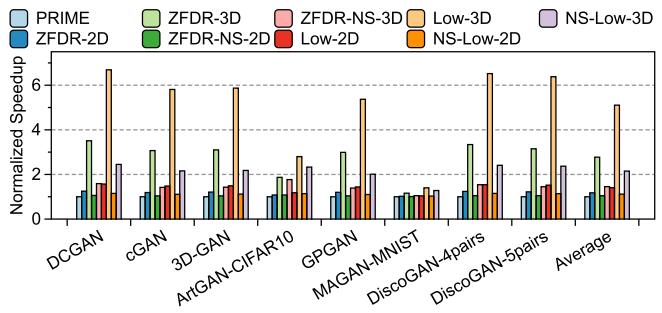


Fig. 20. Performance comparison between 3D-connection and H-tree connection with ZFDR.

Fig. 19 shows the performance of $ZFDR$ in different GAN phases. We use NS to represent normalized space, which means that PRIME uses the same CArray space as our design. $ZFDR$ achieves distinct speedup on DCGAN, cGAN, 3D-GAN, GPGAN and DiscoGAN, which reflects that there are large portions of zeros in these GANs. What's more, $ZFDR$ saves up to 5.2× SArray space for storing inputs (in the case of DCGAN), and saves 3.86× SArray space on average. Note that DiscoGAN-4pairs has 5 phases using $ZFDR$ because its generator has both S-CONV and T-CONV. Moreover, there is no speedup on discriminator of MAGAN-MNIST, because its layers are fully-connected.

When we evaluate the entire process of training GANs with H-tree connection, the speedup of $ZFDR$ almost disappears. This is resulted from the overhead of data transfers. Fig. 20 shows the performance of our 3D connection design compared with H-tree connection. We observe that with our 3D connection design, the speedup of $ZFDR$ is much more visible. Moreover, with 3D connection, duplication (low degree) achieves much higher performance speedup than $ZFDR$ with no duplication, while duplication achieves little speedup with H-tree connection.

TABLE 6
Topologies of GAN Benchmarks

Name	Generator	Item Size	Discriminator
DCGAN [57]	100f-(1024t-512t-256t-128t)(5k2s)-t3	64 × 64	(3c-128c-256c-512c-1024c)(5k2s)-f1
cGAN [55]	100f-(256t-128t-64t)(4k2s)-t3	64 × 64	(3c-64c-128c-256c)(4k2s)-f1
3D-GAN [67]	100f-(512t-256t-128t)(4k2s)-t3	64 × 64 × 64	(1c-64c-128c-256c-512c)(4k2s)-f1
ArtGAN-CIFAR-10 [64]	100f-1024t4k1s-512t4k2s-256t4k2s-128t4k2s-128t3k1s-t3	32 × 32	3c4k2s-128c3k1s-(128c-256c-512c-1024c)(4k2s)-f11
GPGAN [66]	100f-(512t-256t-128t-64t)(4k2s)-t3	64 × 64	(3c-64c-128c-256c-512c)(4k2s)-f1
MAGAN-MNIST [65]	50f-128t7k1s-64t4k2s-t1	28 × 28	784f-256f-256f-784f-f11
DiscoGAN-4pairs [31]	(3c-64c-128c-256c-512t-256t-128t-64t)(4k2s)-t3	64 × 64	(3c-64c-128c-256c-512c)(4k2s)-f1
DiscoGAN-5pairs [31]	(3c-64c-128c-256c-512c)(4k2s)-100f-(512t-256t-128t-64t)(4k2s)-t3	64 × 64	(3c-64c-128c-256c-512c)(4k2s)-f1

(*f:fully-connected c: convolution t:transposed convolution k:kernel s:stride*).

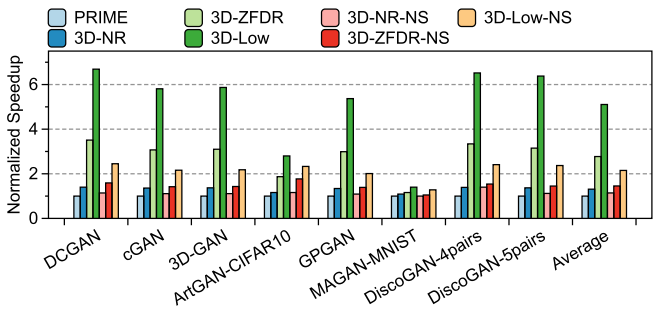


Fig. 21. Performance comparison between ZFDR and normal reshape with 3D-connection.

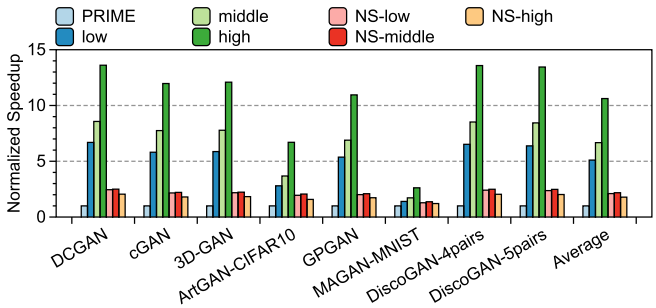


Fig. 22. Performance comparison between LerGAN and PRIME (without approximate weight update).

Fig. 21 compares the performance between $ZFDR$ and normal reshaping (marked as NR) with 3D connection. The results show that with 3D connection, $ZFDR$ with (without) duplication achieves 5.11× (2.77×) speedup on average, while normal reshaping only yields 1.31× speedup, indicating that both our 3D connection design and $ZFDR$ are critical to accelerate GAN execution.

Experiments above show that $ZFDR$ and 3D connection can achieve high speedup when they work together. We further show the performance of LerGAN which combines these two techniques. It's worth to mention that we use LerGAN and LrGAN to represent the PIM architecture without and with the approximate update scheme, respectively. We train the discriminator and generator of each GAN for ten iterations and calculate the average time of each iteration. We compare different duplication degrees of LerGAN with PRIME, shown in Fig. 22. First of all, with our design applied without approximate updating, DCGAN has more speedup than 3D-GAN and GPGAN because it has a larger kernel size than others, which leads to a larger proportion of multiplications with zeros. Besides, MAGAN-MNIST shows nearly no speedup since its discriminator is fully-connected and its generator is small with only one T-CONV.

Fig. 23 shows the results of energy saving. Note that LerGAN-low-NS achieves 28.47× energy saving on average. This energy saving owes much to our zero-free and 3D connection design, since they reduce the amount of data as well as the data movements requiring long wires. Besides, with the increase of duplications, LerGAN exhibits less energy saving, since more duplications leads to (1) more memory writes when updating GANs; and (2) more complex and energy-consuming switch configurations.

We then evaluate the performance and energy saving of *Approximate Update*. We set permissions of $(4x+3)$th and

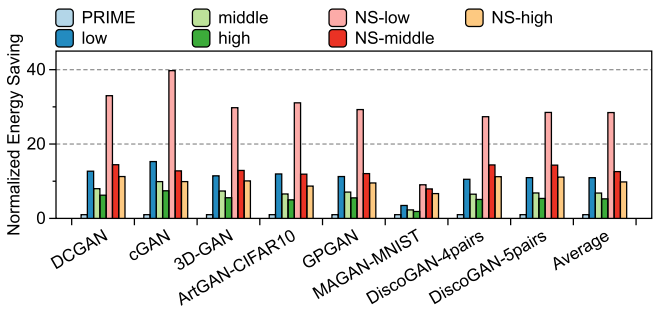


Fig. 23. Energy saving comparison between LerGAN and PRIME (without approximate weight update).

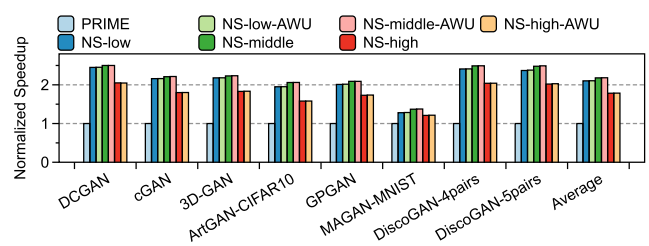


Fig. 24. Performance comparison between LerGAN and LrGAN.

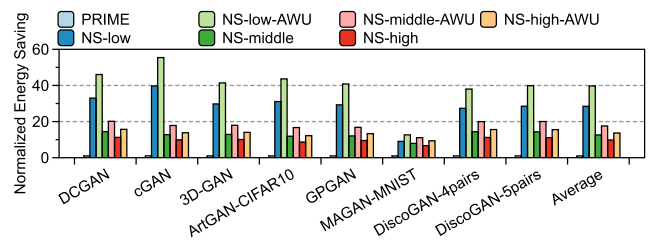


Fig. 25. Energy saving comparison between LerGAN and LrGAN.

$(4x+4)$th columns as TRUE, and others are FALSE $(0 \leq x \leq 31)$. For $(4x+3)$th columns, $[t_{down}, t_{up}]$ is [6, 9], and $[t_{down}, t_{up}]$ is [4, 11] in $(4x+4)$th columns. It is worth to mention that these configurations obtained through various experiments are not optimal. We leave how to find the optimal solution as our future work.

Figs. 24 and 25 depict the speedup and energy saving results of LrGAN with *Approximate Update* compared with PRIME and lerGAN without *Approximate Update*. In both of them, AWU represents approximate weight update algorithm. From Fig. 24 we can figure out that *Approximate Update* are unable to achieve speedup because the latency of each write operation is decided by the longest latency of programming cells in a row. Since the $(4x+1)$th and $(4x+2)$th columns employ normal programming model, they keep the longest latency unchanged. However, *Approximate Update* can achieve 1.4× energy saving on average as shown in Fig. 25, which finally achieves 40× energy saving on low-duplication mode when compared with PRIME. This mainly benefits from avoiding high energy-consuming fine-grain programming scheme.

We also compare LrGAN with the FPGA-based GAN accelerator and GPU platform. Figs. 26 and 27 show the performance and energy consumption of aforementioned architectures, respectively. In terms of the performance, since our approximate update scheme has no speedup on

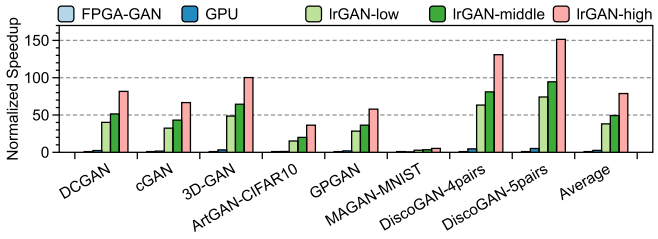


Fig. 26. Performance comparison among FPGA-based GAN accelerator, GPU platform and LrGAN.

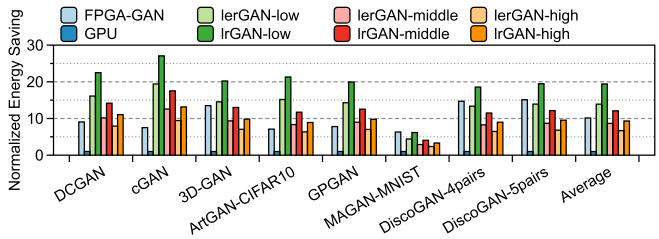


Fig. 27. Energy saving comparison among FPGA-based GAN accelerator, GPU platform, LerGAN, and LrGAN.

LerGAN, we show the results of LrGAN (the speedup of LerGAN is the same with that of LrGAN). LrGAN achieves 47.2× and 21.42× speedup on average over FPGA-GAN and GPU, respectively. What's more, DiscoGAN manifests more speedup over others because (1) it has more T-CONVs, which means more zeros. Our LrGAN with $ZFDR$ design shows higher performance; (2) the size of DiscoGAN is bigger, leading to more off-chip memory accesses for FPGA and GPU, which causes PIM-based LrGAN to perform better. Moreover, GANs with small sizes, such as MAGAN-MNIST, and lacking T-CONVs, cause less speedup. For the energy saving, LerGAN-low and LrGAN-low save more energy than FPGA-based GAN accelerator for GANs with small size but with more frequent T-CONVs (the left five GANs in Fig. 26). However, for GANs with small size and fewer T-CONVs (MAGAN-MNIST), LerGAN shows slightly less energy saving than what FPGA-GAN accelerator performs. This is because LerGAN consumes more energy when updating networks. Consequently, the extra energy cost can not be amortized by the energy-saving opportunity. Thanks to our approximate weight update algorithm, LrGAN has 1.34× energy saving than the FPGA-GAN accelerator on average. Moreover, as shown in Figs. 26 and 27, though more duplication (e.g., LrGAN-high) brings more speedup, it results in more energy consumption.

6.4 Accuracy Loss

We employ Fréchet Inception Distance (*FID*) in [46] to quantify the quality of the pictures generated. *FID* is a score that reflects *the distance between a real and a fake item in the feature level* (the next-to-last layer). Therefore, a lower FID means both higher quality and higher diversity of the generated item. We can calculate *FID* by Equation (15).

$$FID = \left|\left|\mu_r - \mu_g\right|\right|^2 + T_r(\sum\nolimits_r + \sum\nolimits_g - 2(\sum\nolimits_r \sum\nolimits_g)^{\frac{1}{2}}). \quad (15)$$

In Equation (15), r and g represent real items and generated items; μ is the mean of the feature; T_r is the sum of all the diagonal elements; $\sum$ is the covariance matrix of features. In experiments, it's difficult for a human to decipher a generated item from a real one when the difference between the two *FID*s is less than 5.

We set the same noise to train each GAN when comparing the standard training and the approximate training. The accuracy loss calculated by $FID_{app} - FID_o$ is shown in Table 7, which shows the accuracy of our proposed algorithm is guaranteed.

TABLE 7
Accuracy Loss of Approximate Weight Update

Name	FID_o	FID_{app}	Difference
DCGAN	35.6	36.5	0.9
cGAN	64.0	64.7	0.7
3D-GAN	30.3	31.6	1.3
ArtGAN-CIFAR-10	32.5	33.0	0.5
GPGAN	40.7	41.5	0.8
MAGAN-MNIST	3.2	4.7	1.5
DiscoGAN-4pairs	29.4	29.9	0.5
DiscoGAN-5pairs	29.4	29.9	0.5

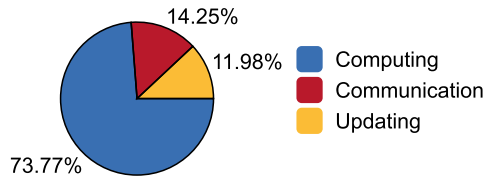


Fig. 28. The breakdown of energy consumption in LrGAN (overall).

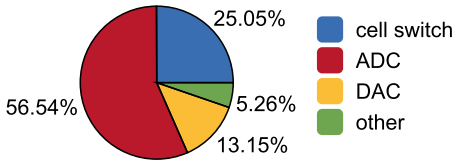


Fig. 29. The breakdown of energy consumption of a ReRAM tile.

6.5 Energy Distribution

Fig. 28 shows the overall energy distribution of LrGAN executed across the experimented benchmarks. The energy of computing dominates 73.77 percent of the total energy in LrGAN since it has a large amount of ReRAM-tile-related operations, while that of communication occupies 14.25 percent, benefited from our 3D-connected PIM design. Moreover, we break down the energy distribution of a ReRAM tile, as shown in Fig. 29. The results show that cell switching (25.05 percent) and ADC (56.54 percent) are the two main energy-consuming contributors. Several studies [38], [71] on materials contribute on reducing energy consumption of cell switching and ADC. If LrGAN is equipped with 1-pJ cell switching [71], and a more energy-saving ADC (e.g., 60 percent [38]), it can achieve nearly 3× power reduction.

6.6 Overhead

The overhead of LrGAN has two parts: software overhead and hardware overhead. For the software overhead caused

by *ZFDR* and *ZFDM*, LrGAN spends 32.52 percent more time than traditional methods on compiling. However, compared with the total time spent on training a GAN(e.g., several days), the overhead of few minutes incurred by the software overhead can be ignored. For the hardware, we add some switches, wires and modified circuit of *Approximate Update*. All of these cause 13.5 percent space overhead compared with PRIME. However, this space overhead can be justified by the higher performance (2.1× speedup) delivered by LrGAN, compared with PRIME using the same space. What's more, *Approximate Update* has only 0.9 percent time and 0.2 percent energy overheads.

7 Related Work

3D Network on Chip (NoC). There are several prior studiess on 3D NoC [1], [8], [30], [39], [54], which are proposed for shortening connections. However, their complex routing algorithms are not suitable for GAN, while our succinct 3D connection design fits GAN well.

NN Accelerators. Many recent works accelerate NN based on FPGAs [3], [47], [60], [72], [74] and ASICs [2], [12], [19], [24], [41], [44], [45], [58], [73]. Diannao family was proposed based on Near-Data Processing (NDP) [11], [13], [17], [42], which locates processors near the memory to reduce the overhead of off-chip memory access. Our design is based on ReRAM-based PIM, further reducing data movements.

ReRAM-Based NN Accelerators. PRIME [15] is an accelerator on basic computations of inference like MMV computation. ISAAC [59] proposed a pipeline solution to accelerate inference of CNNs. PipeLayer [62] further proposed a pipeline solution with intra-layer parallelism on both inference and training of CNNs. TIME [14] proposed a ReRAM-based training-in-memory architecture and further reduced the frequency of ReRAM read/write. Our work proposes a zero-free, 3D connected GAN accelerator.

GAN Accelerators. Song *et al.* [50] proposed FPGA-based GAN accelerator. It uses well-designed dataflows to remove zero operations and increase data reuse on FPGA. Amir *et al.* proposed a SIMD-MIMD acceleration for GAN [4], [5], [6], by removing zeros in GAN training. Chen *et al.* proposed ReGAN, a ReRAM-based GAN accelerator using pipeline [10] design. Our LrGAN design is PIM-based and flexible to handle all zero-related scenarios in GAN training.

8 Conclusion

In this paper, we propose a high-performance, energy-efficient PIM-based GAN accelerator: LrGAN. We design an NVM-based PIM which outperforms the FPGA-based GAN accelerator and GPU in both performance and energy consumption when training GANs. This offsets the flexibility of PIM, which is worse than FPGA and GPU. LrGAN has three main techniques: (1) the Zero-Free Data Reshaping (*ZFDR*) scheme designed for ReRAM-based PIM to remove computations with zeros; (2) the reconfigurable 3D connection in PIM which eliminates the bottleneck of long data movement; and (3) the approximate weight update scheme which prevents unnecessary energy-inefficient fine-grained MLC programming. LrGAN also combines these techniques with minor modifications of software and memory controller. Experiments show that LrGAN achieves 47.2×, 21.42×, and 7.46× speedup over the FPGA-based GAN accelerator, GPU platform, and PRIME respectively. Moreover, LrGAN delivers 13.65×, 10.75×, and 1.34× energy savings over the GPU platform, PRIME, and the FPGA-based GAN accelerator, respectively.

Acknowledgments

This work was supported in part by National Key Research & Development Program of China under Grant 2018YFB 1003301, in part by the National Natural Science Foundation of China under Grant 61832011, and in part by Research and Development Plan in Key Field of Guangdong Province under Grant 2018B010109002.

References

[1] A. B. Ahmed and A. B. Abdallah, "LA-XYZ: Low latency, high throughput look-ahead routing algorithm for 3D network-on-chip (3D-NoC) architecture," in *Proc. IEEE 6th Int. Symp. Embedded Multicore Socs*, 2012, pp. 167–174.

[2] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos,"Cnvlutin: Ineffectual-neuron-free deep neural network computing," *ACM SIGARCH Comput. Archit. News*, vol. 44, no. 3, pp. 1–13, 2016.

[3] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer CNN accelerators," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2016, pp. 1–12.

[4] Y. Amir, F. Hajar, J. W. Philip, S. Kambiz, E. Hadi, and S. K. Nam, "GANAX: A unified SIMD-MIMD acceleration for generative adversarial network," in *Proc. IEEE/ACM Int. Symp. Comput. Archit.*, 2018, pp. 650–661.

[5] Y. Amir, S. Kambiz, E. Hadi, and S. K. Nam, "A SIMD-MIMD acceleration with access-execute decoupling for generative adversarial networks," in *Proc. SysML Conf.*, 2018, pp. 1–3.

[6] Y. Amir *et al.*, "Fxgan: A unified MIMD-SIMD FPGA acceleration with decoupled access-execute units for generative adversarial networks," in *Proc. IEEE Int. Symp. Field-Programmable Custom Comput. Machines*, 2018, pp. 65–72.

[7] M. N. Bojnordi and E. Ipek, "Memristive boltzmann machine: A hardware accelerator for combinatorial optimization and deep learning," in *Proc. IEEE Int. Symp. High Perform. Comput.*, 2016, pp. 1–13.

[8] C.-H. Chao, K.-Y. Jheng, H.-Y. Wang, J.-C. Wu, and A.-Y. Wu, "Traffic-and thermal-aware run-time thermal management scheme for 3D NoC systems," in *Proc. 4th ACM/IEEE Int. Symp. Netw.-on-Chip*, 2010, pp. 223–230.

[9] O. Chapelle, B. Scholkopf, and A. Zien, "Semi-supervised learning (chapelle, o. et al., eds.; 2006)[book reviews]," *IEEE Trans. Neural Netw.*, vol. 20, no. 3, pp. 542–542, Mar. 2009.

[10] F. Chen, L. Song, and Y. Chen, "ReGAN: A pipelined ReRAM-based accelerator for generative adversarial networks," in *Proc. 23rd Asia South Pacific Des. Autom. Conf.*, 2018, pp. 178–183.

[11] T. Chen *et al.*, "DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 269–284, 2014.

[12] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE J. Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, Jan. 2017.

[13] Y. Chen *et al.*,"DaDianNao: A machine-learning supercomputer," in *Proc. 47th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2014, pp. 609–622.

[14] M. Cheng *et al.*, "TIME: A training-in-memory architecture for memristor-based deep neural networks," in *Proc. 54th Annu. Des. Autom. Conf.*, 2017, Art. no. 26.

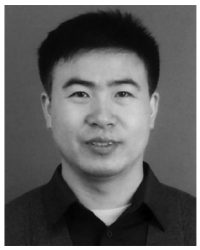
[15] P. Chi *et al.*,"PRIME: A novel processing-in-memory architecture for neural network computation in ReRAM-based main memory," *ACM SIGARCH Comput. Archit. News*, vol. 44, no. 3, pp. 27–39, 2016.

[16] X. Dong, "Modeling and leveraging emerging non-volatile memories for future computer designs," Ph.D. dissertation, Comput. Sci. Eng., Pennsylvania State Univ., University Park, PA, USA, 2011.

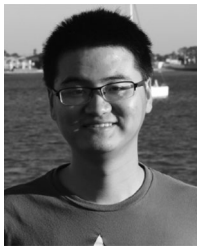
[17] Z. Du *et al.*, "ShiDianNao: Shifting vision processing closer to the sensor," *ACM SIGARCH Comput. Archit. News*, vol. 43, no. 3, pp. 92–104, 2015.
[18] D. Erhan, Y. Bengio, A. Courville, P.-A. Manzagol, P. Vincent, and S. Bengio, "Why does unsupervised pre-training help deep learning?" *J. Mach. Learn. Res.*, vol. 11, no. Feb, pp. 625–660, 2010.
[19] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun, "NeuFlow: A runtime reconfigurable dataflow processor for vision," in *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit. Workshops*, 2011, pp. 109–116.
[20] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim, "NDA: Near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules," in *Proc. IEEE 21st Int. Symp. High Perform. Comput. Archit.*, 2015, pp. 283–295.
[21] C. Finn, I. Goodfellow, and S. Levine, "Unsupervised learning for physical interaction through video prediction," in *Proc. 30th Int. Conf. Neural Inf. Process. Syst.*, 2016, pp. 64–72.
[22] A. Ghosh, B. Bhattacharya, and S. B. R. Chowdhury, "SAD-GAN: Synthetic autonomous driving using generative adversarial networks," 2016, *arXiv:1611.08788*.
[23] A. Graves, A.-R. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process.*, 2013, pp. 6645–6649.
[24] S. Han *et al.*,"EIE: Efficient inference engine on compressed deep neural network," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit.*, 2016, pp. 243–254.
[25] T. Hastie, R. Tibshirani, and J. Friedman, "Unsupervised learning," in *The Elements of Statistical Learning*. Berlin, Germany: Springer, 2009, pp. 485–585.
[26] G. Hinton *et al.*, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal Process. Mag.*, vol. 29, no. 6, pp. 82–97, Nov. 2012.
[27] C.-W. Hsu *et al.*, "Self-rectifying bipolar TaOx/TiO_2 RRAM with superior endurance over 10^{12} cycles for 3D high-density storage-class memory," in *Proc. Symp. VLSI Technol.*, 2013, pp. T166–T167.
[28] G. Huang, S. Song, J. N. Gupta, and C. Wu, "Semi-supervised and unsupervised extreme learning machines," *IEEE Trans. Cybern.*, vol. 44, no. 12, pp. 2405–2417, Dec. 2014.
[29] N. P. Jouppi, A. B. Kahng, N. Muralimanohar, and V. Srinivas, "CACTI-IO: CACTI with OFF-chip power-area-timing models," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 23, no. 7, pp. 1254–1267, Jul. 2015.
[30] J. Kim *et al.*, "A novel dimensionally-decomposed router for on-chip communication in 3D architectures," *ACM SIGARCH Comput. Archit. News*, vol. 35, no. 2, pp. 138–149, 2007.
[31] T. Kim, M. Cha, H. Kim, J. Lee, and J. Kim, "Learning to discover cross-domain relations with generative adversarial networks," 2017, *arXiv: 1703.05192*.
[32] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. 25th Int. Conf. Neural Inf. Process. Syst.*, 2012, pp. 1097–1105.
[33] H. Kwon, A. Samajdar, and T. Krishna, "MAERI: Enabling flexible dataflow mapping over DNN accelerators via reconfigurable interconnects," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 461–475, 2018.
[34] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, 2015, Art. no. 436.
[35] C. Ledig *et al.*, "Photo-realistic single image super-resolution using a generative adversarial network," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2017, pp. 4681–4690.
[36] H. Lee *et al.*, "Evidence and solution of over-RESET problem for HfOX based resistive memory with sub-ns switching speed and high endurance," in *Proc. IEEE Int. Electron Devices Meeting*, 2010, pp. 19.7.1–19.7.4.
[37] M.-J. Lee *et al.*, "A fast, high-endurance and scalable non-volatile memory device made from asymmetric Ta_2O_{5-x}/TaO_{2-x} bilayer structures," *Nat. Materials*, vol. 10, no. 8, 2011, Art. no. 625.
[38] B. Li, L. Xia, P. Gu, Y. Wang, and H. Yang, "Merging the interface: Power, area and accuracy co-optimization for RRAM crossbar-based mixed-signal computing system," in *Proc. 52nd Annu. Des. Autom. Conf.*, 2015, Art. no. 13.
[39] F. Li, C. Nicopoulos, T. Richardson, Y. Xie, V. Narayanan, and M. Kandemir, "Design and management of 3D chip multiprocessors using network-in-memory," *ACM SIGARCH Comput. Archit. News*, vol. 34, no. 2, pp. 130–141, 2006.
[40] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, and Y. Xie, "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories," in *Proc. 53nd ACM/EDAC/IEEE Des. Autom. Conf.*, 2016, pp. 1–6.
[41] R. LiKamWa, Y. Hou, J. Gao, M. Polansky, and L. Zhong, "RedEye: Analog ConvNet image sensor architecture for continuous mobile vision," *ACM SIGARCH Comput. Archit. News*, vol. 44, no. 3, pp. 255–266, 2016.
[42] D. Liu *et al.*, "PuDianNao: A polyvalent machine learning accelerator," *ACM SIGARCH Comput. Archit. News*, vol. 43, no. 1, pp. 369–381, 2015.
[43] M.-Y. Liu and O. Tuzel, "Coupled generative adversarial networks," in *Proc. 30th Int. Conf. Neural Inf. Process. Syst.*, 2016, pp. 469–477.
[44] S. Liu *et al.*, "Cambricon: An instruction set architecture for neural networks," *ACM SIGARCH Comput. Archit. News*, vol. 44, no. 3, pp. 393–405, 2016.
[45] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li, "FlexFlow: A flexible dataflow accelerator architecture for convolutional neural networks," in *Proc. IEEE Int. Symp. High Perform. Comput.*, 2017, pp. 553–564.
[46] M. Lucic, K. Kurach, M. Michalski, S. Gelly, and O. Bousquet, "Are GANs created equal? A large-scale study," in *Proc. 32nd Int. Conf. Neural Inf. Process. Syst.*, 2018, pp. 700–709.
[47] Y. Ma, Y. Cao, S. Vrudhula, and J.-S. Seo, "Optimizing loop operation and dataflow in FPGA acceleration of deep convolutional neural networks," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2017, pp. 45–54.
[48] H. Mao, M. Song, T. Li, Y. Dai, and J. Shu, "LerGAN: A zero-free, low data movement and PIM-based GAN architecture," in *Proc. 51st Annu. IEEE/ACM Int. Symp. Microarchit.*, 2018, pp. 669–681.
[49] M. Mao, Y. Cao, S. Yu, and C. Chakrabarti, "Optimizing latency, energy, and reliability of 1T1R ReRAM through cross-layer techniques," *IEEE J. Emerg. Sel. Topics Circuits Syst.*, vol. 6, no. 3, pp. 352–363, Sep. 2016.
[50] S. Mingcong, Z. Jiaqi, C. Huixiang, and L. Tao, "Towards efficient microarchitectural design for accelerating unsupervised GAN-based deep learning," in *Proc. IEEE 24st Int. Symp. High Perform. Comput. Archit.*, 2018, pp. 66–77.
[51] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0," in *Proc. 40th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2007, pp. 3–14.
[52] J. Ngiam, A. Khosla, M. Kim, J. Nam, H. Lee, and A. Y. Ng, "Multimodal deep learning," in *Proc. 28th Int. Conf. Mach. Learn.*, 2011, pp. 689–696.
[53] D. Niu, Q. Zou, C. Xu, and Y. Xie, "Low power multi-level-cell resistive memory design with incomplete data mapping," in *Proc. IEEE 31st Int. Conf. Comput. Des.*, 2013, pp. 131–137.
[54] D. Park *et al.*, "MIRA: A multi-layered on-chip interconnect router architecture," in *Proc. 35th Int. Symp. Comput. Archit.*, 2008, pp. 251–261.
[55] D. Pathak, P. Krahenbuhl, J. Donahue, T. Darrell, and A. A. Efros, "Context encoders: Feature learning by inpainting," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 2536–2544.
[56] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," *ACM SIGARCH Comput. Archit. News*, vol. 37, no. 3, pp. 24–33, 2009.
[57] Y. Yu, Z. Gong, P. Zhong, and J. Shan, "Unsupervised representation learning with deep convolutional neural network for remote sensing images," in *Proc. Int. Conf. Image Graphics*, 2017, pp. 97–108.
[58] B. Reagen *et al.*, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," *ACM SIGARCH Comput. Archit. News*, vol. 44, no. 3, pp. 267–278, 2016.
[59] A. Shafiee *et al.*, "ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," *ACM SIGARCH Comput. Archit. News*, vol. 44, no. 3, pp. 14–26, 2016.
[60] H. Sharma *et al.*, "From high-level deep neural models to FPGAs," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2016, pp. 1–12.
[61] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014, *arXiv:1409.1556*.
[62] L. Song, X. Qian, H. Li, and Y. Chen, "PipeLayer: A pipelined ReRAM-based accelerator for deep learning," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2017, pp. 541–552.

[63] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," in *Proc. Int. Conf. Mach. Learn.*, 2013, pp. 1139–1147.
[64] W. R. Tan, C. S. Chan, H. E. Aguirre, and K. Tanaka, "ArtGAN: Artwork synthesis with conditional categorial GANs," in *Proc. IEEE Int. Conf. Image Process.*, 2017, pp. 3760–3764.
[65] R. Wang, A. Cully, H. J. Chang, and Y. Demiris, "MAGAN: Margin adaptation for generative adversarial networks," 2017, *arXiv: 1704.03817*.
[66] H. Wu, S. Zheng, J. Zhang, and K. Huang, "GP-GAN: Towards realistic high-resolution image blending," in *Proc. 27th ACM Int. Conf. Multimedia*, 2019, pp. 2487–2495.
[67] J. Wu, C. Zhang, T. Xue, B. Freeman, and J. Tenenbaum, "Learning a probabilistic latent space of object shapes via 3D generative-adversarial modeling," in *Proc. 30th Int. Conf. Neural Inf. Process. Syst.*, 2016, pp. 82–90.
[68] C. Xu *et al.*, "Overcoming the challenges of crossbar resistive memory architectures," in *Proc. IEEE 21st Int. Symp. High Perform. Comput. Archit.*, 2015, pp. 476–488.
[69] C. Xu, D. Niu, N. Muralimanohar, N. P. Jouppi, and Y. Xie, "Understanding the trade-offs in multi-level cell ReRAM memory design," in *Proc. 50th ACM/EDAC/IEEE Des. Autom. Conf.*, 2013, pp. 1–6.
[70] 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," 2015, *arXiv:1506.03365*.
[71] S. Yu, B. Gao, Z. Fang, H. Yu, J. Kang, and H.-S. P. Wong, "A neuromorphic visual system using RRAM synaptic devices with Sub-pJ energy and tolerance to variability: Experimental characterization and large-scale modeling," in *Proc. IEEE Int. Electron Devices Meeting*, 2012, pp. 10.4.1–10.4.4.
[72] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2015, pp. 161–170.
[73] S. Zhang *et al.*, "Cambricon-X: An accelerator for sparse neural networks," in *Proc. 49th Annu. IEEE/ACM Int. Symp.*, 2016, pp. 1–12.
[74] M. Zhu, L. Liu, C. Wang, and Y. Xie, "CNNLab: A novel parallel framework for neural networks using GPU and FPGA-A practical study with trade-off analysis," 2016, *arXiv:1606.06234*.

Haiyu Mao (Member, IEEE) received the BS degree in software engineering from Northeastern University, Boston, Massachusetts, in 2015. She is currently working toward the PhD degree with the Department of Computer Science and Technology, Tsinghua University, China. Her research interests include non-volatile memory, processing in memory, memory security, and machine learning accelerator.



Jiwu Shu (Fellow, IEEE) received the PhD degree in computer science from Nanjing University, China, in 1998, and finished the postdoctoral position research at Tsinghua University, China, in 2000. Since then, he has been teaching at Tsinghua University, China, and is currently a professor with the Department of Computer Science and Technology, Tsinghua University, China. His current research interests include network storage systems, nonvolatile memory-based storage systems, storage security and reliability, and parallel and distributed computing.



Mingcong Song (Member, IEEE) received the BS degree in electronic and informantion engineering from the Huazhong University of Science and Technology, China, in 2010 and the PhD degree in computer engineering from the University of Florida, Gainesville, Florida, in 2018. His research area is computer architecture.

Tao Li (Fellow, IEEE) received the PhD degree in computer engineering from the University of Texas at Austin, Austin, Texas. He is a full professor with the Department of Electrical and Computer Engineering, University of Florida, Gainesville, Florida. His research interests include computer architecture, microprocessor/memory/storage system design, virtualization technologies, energy-efficient/ sustainable/ dependable data center, cloud/big data computing platforms, the impacts of emerging technologies/applications on computing, and evaluation of computer systems. He received 2009 National Science Foundation Faculty Early CAREER Award, 2008, 2007, 2006 IBM Faculty Awards, 2008 Microsoft Research Safe and Scalable Multi-core Computing Award and 2006 Microsoft Research Trustworthy Computing Curriculum Award. He co-authored two papers that won the best paper awards in ICCD 2016, HPCA 2011 and six papers that were nominated for the best paper awards in HPCA 2017, ICPP 2015, CGO 2014, DSN 2011, MICRO 2008, and MASCOTS 2006. He is one of the College of Engineering winners, University of Florida Doctor Dissertation Advisor/Mentoring Award for 2013–2014 and 2011–2012.