2018 51st Annual IEEE/ACM International Symposium on Microarchitecture

LerGAN: A Zero-free, Low Data Movement and PIM-based GAN Architecture*

1st Haiyu Mao
Department of Computer Science and Technology, Tsinghua University
Beijing, China
mhy15@mails.tsinghua.edu.cn

2nd Mingcong Song
Department of Electrical and Computer Engineering, University of Florida
Gainesville, FL, USA
songmingcong@ufl.edu

3rd Tao Li
Department of Electrical and Computer Engineering, University of Florida
Gainesville, FL, USA
taoli@ece.ufl.edu

4th Yuting Dai
College of Computer Science and Technology Guizhou University
Guizhou, China
yutingdai90@gmail.com

5th Jiwu Shu**
Department of Computer Science and Technology Tsinghua University
Beijing, China
shujw@tsinghua.edu.cn

***Abstract*—As a powerful unsupervised learning method, Generative Adversarial Network (GAN) plays an important role in many domains such as video prediction and autonomous driving. It is one of the ten breakthrough technologies in 2018 reported in MIT Technology Review. However, training a GAN imposes three more challenges: (1) intensive communication caused by complex train phases of GAN, (2) much more ineffectual computations caused by special convolutions, and (3) more frequent off-chip memory accesses for exchanging inter-mediate data between the generator and the discriminator.**

In this paper, we propose LerGAN[1], 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. Our proposed techniques reduce data movement to a great extent, avoiding I/O to become a bottleneck of training GANs. Finally, we propose LerGAN based on these two techniques, providing different levels of accelerating GAN for programmers. Experiments shows that LerGAN achieves 47.2×, 21.42× and 7.46× speedup over FPGA-based GAN accelerator, GPU platform, and ReRAM-based neural network accelerator respectively. Moreover, LerGAN achieves 9.75×, 7.68× energy saving on average over GPU platform, ReRAM-based neural network accelerator respectively, and has 1.04× energy consuming over FPGA-based GAN accelerator.

I. Introduction

Tremendous success has been fueled by supervised deep learning in image classification, speech recognition, and so on [31] [58] [33] [60] [50] [22] [25]. 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] [27] and unsupervised learning [24] [20] [17] 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 [20], autonomous driving [21] and photo resolution upgrading [34].

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 [19]. 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) [39] [15] [56] [7] [59], 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) complex dataflow patterns between the two models. These two features degrade the efficiency of 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.

*This work is supported by the National Major Project of Scientific Instrument of National Natural Science Foundation of China (Grant No.61327902), National Key Research & Development Projects of China (Grant No.2018YFB1003301), and in part by NSF grants 1822989, 1822459, 1527535, 1423090, and 1320100.

**Jiwu Shu is the corresponding author of this paper.

[1] "Ler" comes from removing "o" from "zero" which represents removing 0, and changing "z" to "l" to represent shortening wire connection.

978-1-5386-6240-3/18/$31.00 ©2018 IEEE
DOI 10.1109/MICRO.2018.00060

(2) complex dataflow of GAN exists irregular data dependencies. Therefore, limited I/O bandwidth stalls GAN training.

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, a reconfigurable 3D connection architecture is proposed, 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. Putting ZFDR and reconfigurable 3D interconnection architecture together, we propose LerGAN, a ReRAM-based 3D connection GAN accelerator, 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 flexibly I/O connection configuration for the complex dataflows in GAN training. Experiments shows that LerGAN achieves 47.2×, 21.42× and 7.46× speedup over FPGA-based GAN accelerator, GPU platform and ReRAM-based neural network accelerator respectively. Moreover, LerGAN achieves 9.75×, 7.68× energy saving on average over GPU platform, ReRAM-based NN accelerator respectively, and has 1.04× energy overhead over FPGA-based GAN accelerator.

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 (ZFDR) 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 LerGAN based on ZFDR 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 LerGAN 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 II. Then we analyze the challenges of using PIM to accelerate GAN training in Section III. We present our ZFDR and 3D-connected PIM in Section IV. The design of LerGAN which employs ZFDR and 3D-connected PIM is in Section V. Section VI evaluates the proposed algorithms, 3D-connected PIM and LerGAN system. Finally, we present related works and conclusions in Section VII and Section VIII respectively.

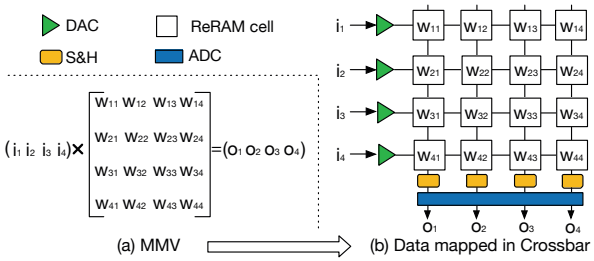

Fig. 1. Mapping MMV to ReRAM Crossbar.

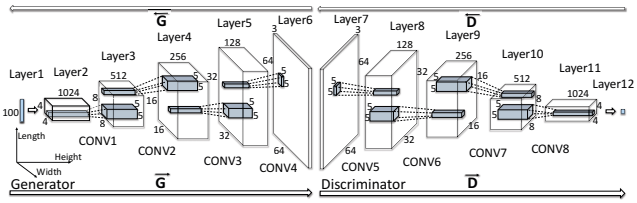

Fig. 2. DCGAN Outline

II. Background

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

A. ReRAM-based PIM

ReRAM stands out from other non-volatile memories (NVMs) since it has high density, low write latency (less than 10% performance degradation, compared to an ideal DRAM) [65], and low write energy (up to 72% lower than DRAM) [46]. Moreover, it has high endurance ($> 10^{10}$ [35] [36], up to 10^{12} [36] [26], much higher than that of PCM , which is $10^7 \sim 10^8$ [53]). If a network needs to be trained for 10^5 times [42], ReRAM-based PIM can train $10^5 \sim 10^7$ such networks. Due to these benefits of ReRAM, recent studies [15] [56] [59] [14] 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.

B. 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

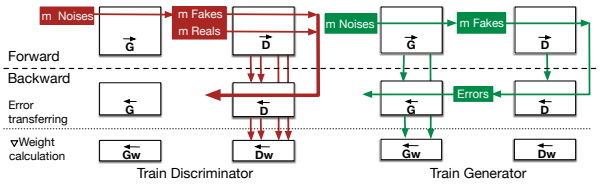

Fig. 3. Dataflows of Training Discriminator and Generator of DCGAN

to compete with each other iteratively. A minibatch stochastic gradient descent method can be used to train this model, where in each training iteration, a minibatch of m noise samples $\{n_1, n_2, ..., n_m\}$ and m true examples $\{x_1, x_2, ..., x_m\}$ 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.

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

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

We take the most popular Deep Convolutional Generative Adversarial Network (DCGAN) [54] 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.

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

TABLE I
NOTATIONS USED FOR EXPLANATION OF TRAINING DCGAN.

Fig.3 shows dataflows of training DCGAN and Table I 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

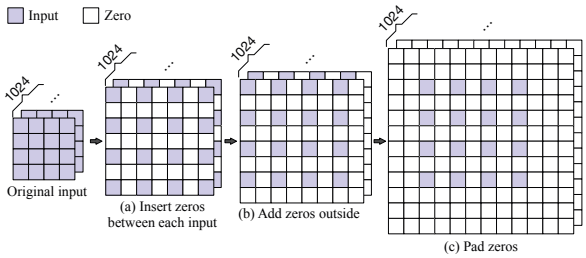

Fig. 4. Steps of Adding Zeros in Inputs of CONV1.

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 $\overleftarrow{D_w}$. Firstly, ∇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.

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

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.

$$\nabla W^l = a^{l-1} \nabla z^l \quad (4)$$

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

III. CHALLENGES OF PIM-BASED GAN ACCELERATOR

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.

A. 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 II and take CONV1 of the generator in Fig. 2 as an example of T-CONV. As

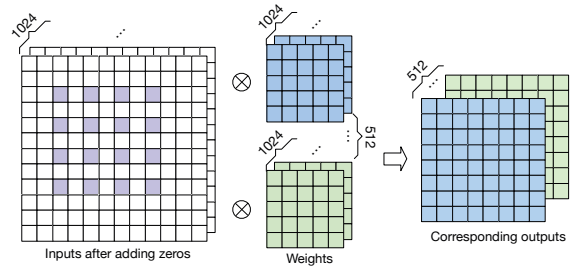

Fig. 5. Convolution on Inserted Zeros Inputs with Stride of 1.

shown in Fig. 2, $I_w = I_l = 4$ 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 (Fig.4(a)), then add one zero at the end of input (Fig.4(b)) and finally use zero padding of 2 (Fig. 4(c)). After that, we convolute it with 512 kernels, whose W_w = W_l = 5 and W_h is 1024. Eventually, we obtain an output whose size is $8 \times 8 \times 512$. In this example, we store and transfer 147456 input values while only 16384 of them are useful. Moreover, we conduct 1638400 multiplications while 295936 of them are useful, whose efficiency is only 18.06%.

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

TABLE II
NOTATIONS USED FOR EXPLANATION OF CONVOLUTION OPERATIONS.

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 relationship is described in Equation 5.

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

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}.

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

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

From Equation 6 and Equation 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

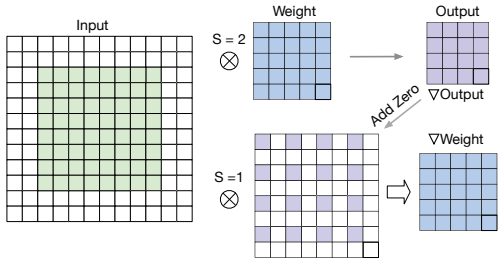

Fig. 6. An Example of W-CONV of Discriminator.

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.

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

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

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

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.

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

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

B. Inefficient I/O Connection

For training where massive memory reads/writes are required to update kernel weights, PipeLayer [59] 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

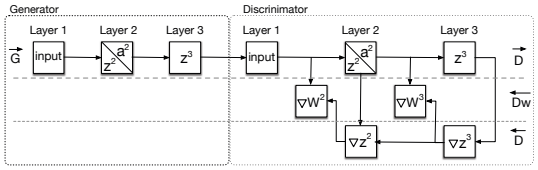

Fig. 7. Dataflow of Training Discriminator.

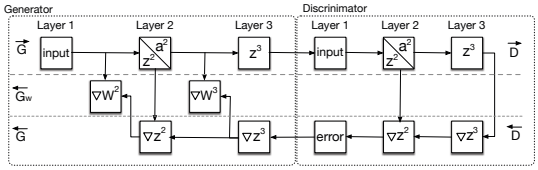

Fig. 8. Dataflow of Training Generator.

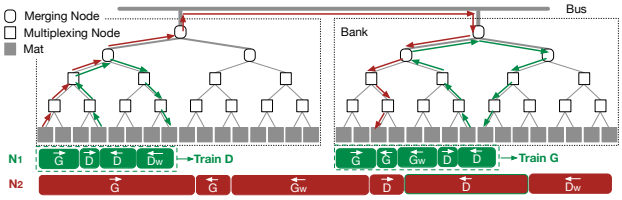

Fig. 9. Networks Mapped to H-tree Connected Tiles.

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 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 [32]. Since the dataflow of GAN training is much more complicated, simply doing so will not achieve desirable performance of speedup.

IV. Our Proposed Solutions

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

A. PIM-Based Zero-Free Scheme

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

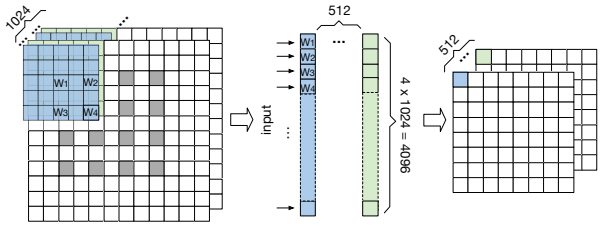

Fig. 10. Example of Zero Free Data Reshaping.

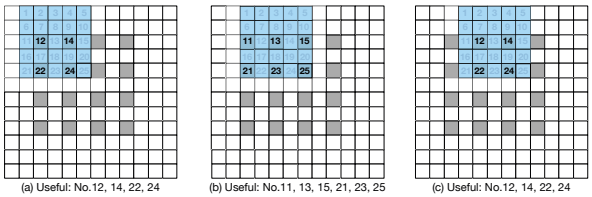

Fig. 11. Example of How Useful Weights Change When Sliding.

that multiply non-zero inputs, as shown in Fig.10. After reshaping all the 512 weight kernels into a 512×4096 matrix, we map this weight matrix into the Carray and feed the corresponding 4096 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.10, we slide kernel weights with stride of 1. When sliding, the useful kernel weights change. Fig.11 gives an example of how useful kernel weights change when sliding. Thus, in step (c), the weight matrix can be reused since it is the same with that in step (a). 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. 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 179200 weights. It means that in order to achieve the same performance as ZFDR, duplicating weights directly not only consumes 75% 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:

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

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

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 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.

B. 3D-Connected PIM for GAN Training

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

Fig.12 (a) shows the 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

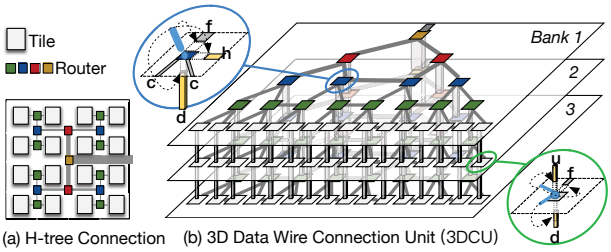

Fig. 12. 3D Connection Based on Original H-tree Connection.

blue nodes shown in Fig.12 (b). 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.12 (b) (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.13 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.13 (a). 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.13, 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

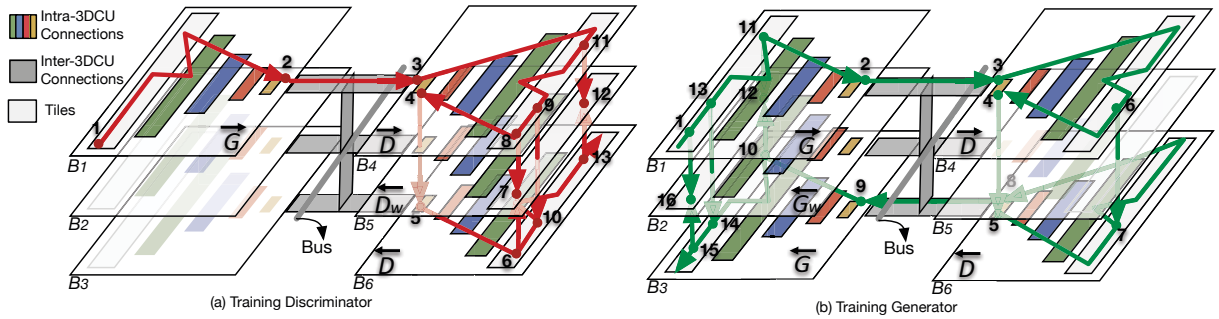

Fig. 13. Dataflows of GAN Training Using 3DCUs.

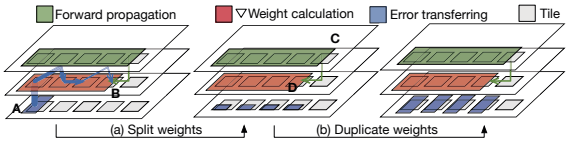

Fig. 14. Data mappings on 3D-connected PIM.

the computation of one layer, we map the corresponding part of $\overleftarrow{D_w}$ and $\overleftarrow{D}$ to B_5 and B_6 respectively. Note that we continue forward propagation of the discriminator when we map $\overleftarrow{D_w}$ and $\overleftarrow{D}$. 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 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.

Fig.13 (b) 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 $\overleftarrow{D}$, 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.14 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.14). 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 (a) is space-saving but less parallelism. Also, we can duplicate weights after splitting, like (b) shows. This improves the parallelism but turns out to be space consuming. The detailed design will be introduced in Section V.

V. LerGAN Design

In this section, we present how Zero-Free Scheme in Section IV-A and 3D Connected PIM in Section IV-B work together in LerGAN.

Fig.15 elaborates the outline of LerGAN 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

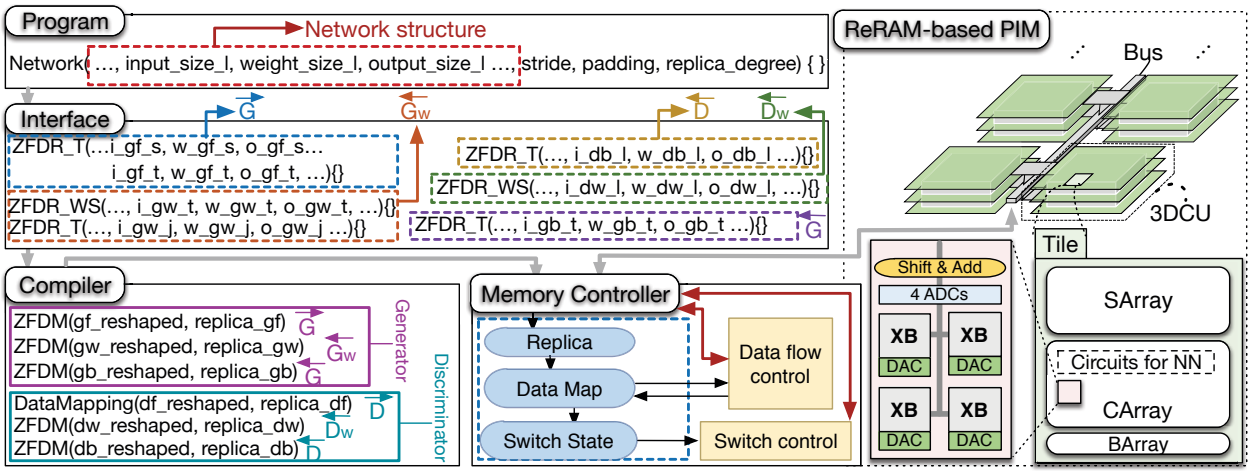

Fig. 15. Outline of LerGAN (an architecture combined techniques of $ZFDR$ and 3DCUs).

for further removing zeros, just like the way of traditional NN frameworks. Their parameters are passed from programming a network. These two functions also process the network layer by layer. The interface component in Fig.15 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.15). 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 IV-A). Based on parameters defined above, we can define $replica\_gf$ as Table III shows.

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	$replica\_e_{max}$	$replica\_i_{max}$

TABLE III
VALUE OF $replica\_gf$.

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 $\overrightarrow{D}$.

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

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.15). 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 LerGAN. It is configured with several 3DCU pairs introduced in Fig.13 in Section IV-B. Each tile in 3DCU contains $SArray$, $CArray$ and $BArray$, using the design in PRIME [15], which has been already introduced in Section II-A. The ReRAM crossbars in a $CArray$ (marked in light pink in Fig.15) employ the design of that in ISAAC [56], 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.

VI. 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.

A. Experimental Setup

We compare LerGAN with (1) GANs running on GPU platform; (2) FPGA-based GAN accelerator [47]; 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 LerGAN are listed in Table IV. The configurations of ReRAM are from [48].

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	32.8ns/41.4ns read/write latency; 413pJ/665pJ read/write energy
	H-Tree	29.9ns latency, 386pJ energy
	Tile	2.9ns/11.5ns read/write latency; 3.3pJ/34.8pJ read/write energy
I/O Frequency		1.6GHz

TABLE IV
Hardware Configurations.

For LerGAN configuration, we use 4-bit for each ReRAM cell, and 16-bit for input, weight and output (i.e. same as [59]). The size of ReRAM array is 128×128 cells. We configure half of a tile for CArray (64MB), 1/64 of the tile for BArray (2MB) and the remaining 62MB for SArray. We use CACTI-6.5 [49], CACTO-IO [28] to model our interconnects and off-chip connects respectively.

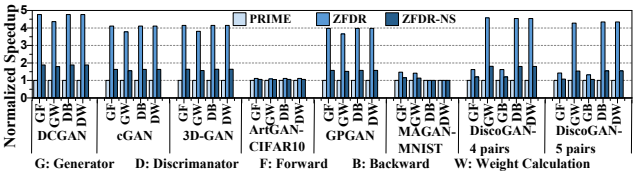

Fig. 16. Performance Comparison on Phases used ZFDR with PRIME

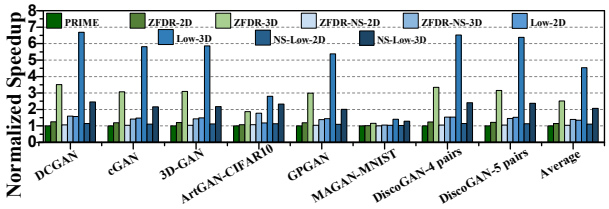

Fig. 17. Performance Comparison between 3D-Connection and H-tree Connection with ZFDR

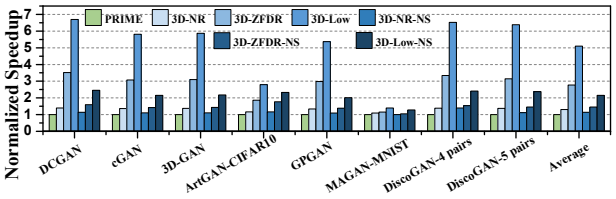

Fig. 18. Performance Comparison between ZFDR and Normal Reshape with 3D-Connection

B. Benchmarks

We employ 8 state-of-the-art GAN networks as our benchmarks, shown in Table V. 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.

C. Results

We fully train the networks in Table V 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 LerGAN and alternative PIM design such as PRIME. Moreover, we compare LerGAN 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.16 shows the performance of $ZFDR$ in different GAN phases. We use NS to represent normalized space, which

Name	Generator	Item Size	Discriminator
DCGAN [54]	100f-(1024t-512t-256t-128t)(5k2s)-t3	64 × 64	(3c-128c-256c-512c-1024c)(5k2s)-f1
cGAN [52]	100f-(256t-128t-64t)(4k2s)-t3	64 × 64	(3c-64c-128c-256c)(4k2s)-f1
3D-GAN [64]	100f-(512t-256t-128t)(4k2s)-t3	64 × 64 × 64	(1c-64c-128c-256c-512c)(4k2s)-f1
ArtGAN-CIFAR-10 [61]	100f-1024t4k1s-512t4k2s-256t4k2s-128t4k2s-128t3k1s-t3	32 × 32	3c4k2s-128c3k1s-(128c-256c-512c-1024c)(4k2s)-f11
GPGAN [63]	100f-(512t-256t-128t-64t)(4k2s)-t3	64 × 64	(3c-64c-128c-256c-512c)(4k2s)-f1
MAGAN-MNIST [62]	50f-128t7k1s-64t4k2s-t1	28 × 28	784f-256f-256f-784f-f11
DiscoGAN-4pairs [30]	(3c-64c-128c-256c-512t-256t-128t-64t)(4k2s)-t3	64 × 64	(3c-64c-128c-256c-512c)(4k2s)-f1
DiscoGAN-5pairs [30]	(3c-64c-128c-256c-512c)(4k2s)-100f-(512t-256t-128t-64t)(4k2s)-t3	64 × 64	(3c-64c-128c-256c-512c)(4k2s)-f1

TABLE V
TOPOLOGIES OF GAN BENCHMARKS. (f:$fully$-$connected$ c: $convolution$ t:$transposed\ convolution$ k:$kernel$ s:$stride$)

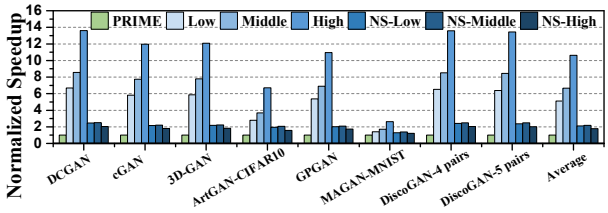

Fig. 19. Performance Comparison between LerGAN and PRIME

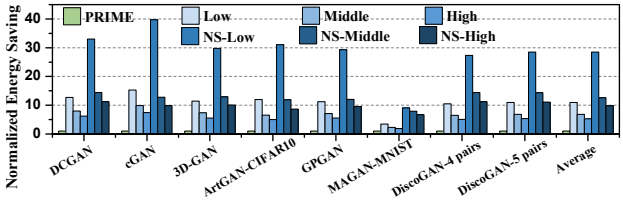

Fig. 20. Energy Saving Comparison between LerGAN and PRIME

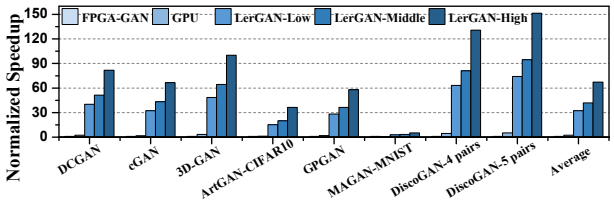

Fig. 21. Performance Comparison among FPGA-based GAN accelerator, GPU platform and LerGAN

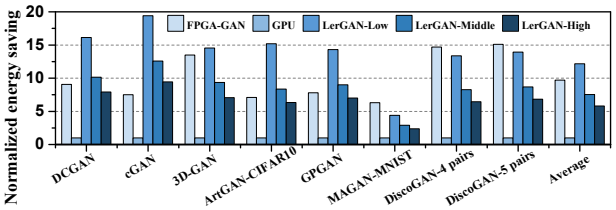

Fig. 22. Energy Saving Comparison among FPGA-based GAN accelerator, GPU platform and LerGAN

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.17 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.

Fig.18 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 performance of LerGAN which combines these two techniques. 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.19. First of all, with our design applied, 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.20 shows the results of energy saving. Note that LerGAN-low-NS achieves 28.47× energy saving on average. This high 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 reads/writes when updating GANs; and (2) more complex and energy-consuming switch configurations.

We also compare LerGAN with FPGA-based GAN accelerator and GPU platform. Fig.21 and 22 show the performance and energy consumption of aforementioned architectures, respectively. In terms of the performance, LerGAN achieves 47.2× and 21.42× speedup on average over FPGA-GAN and

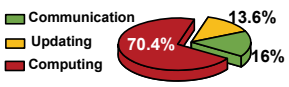

Fig. 23. The Breakdown of Energy Consumption in LerGAN (Overall)

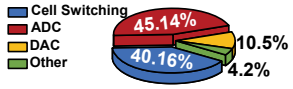

Fig. 24. The Breakdown of Energy Consumption of a ReRAM Tile

GPU, respectively. What's more, DiscoGAN manifests more speedup over others because (1) it has more T-CONVs, which means more zeros. Our LerGAN 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 LerGAN to perform better. Moreover, GANs with small sizes, such as MAGAN-MNIST, and lacking of T-CONVs, cause less speedup. For the energy saving, LerGAN-low saves more energy than FPGA-based GAN accelerator for GANs with small size but with more frequent T-CONVs (the left five GANs in Fig.21). However, for GANs with small size and less T-CONVs (MAGAN-MNIST) and GANs with big size (DiscoGAN), LerGAN shows slightly less energy saving than what FPGA-GAN accelerator performs. This is because LerGAN consumes more energy when updating networks, consequently extra energy cost can not be amortized by the energy saving opportunity. On average, LerGAN has 1.04× energy consumption than FPGA-GAN accelerator. Moreover, as shown in Fig.21 and 22, though more duplication (e.g. LerGAN-high) brings more speedup, it results in more energy consumption.

D. Energy Distribution

Fig.23 shows the overall energy distribution of LerGAN executed across the experimented benchmarks. The energy of computing dominates 70.4% of the total energy in LerGAN since it has a large amount of ReRAM-tile-related operations, while that of communication occupies 16%, benefited from our 3D-connected PIM design. Moreover, we break down the energy distribution of a ReRAM tile, as shown in Fig.24. The results show that cell switching (40.16%) and ADC (45.14%) are the two main energy-consuming contributors. Several studies [66] [37] contribute on reducing energy consumption of cell switching and ADC. If LerGAN is equipped with 1-pJ cell switching [66], and a more energy-saving ADC (e.g. 60% [37]), it can achieve nearly 3× power reduction.

E. Overhead

The overhead of LerGAN has two parts: software overhead and hardware overhead. For the software overhead caused by $ZFDR$ and $ZFDM$, LerGAN spends 32.52% 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, since we add some switches and wires, it causes 13.3% space overhead compared with PRIME. However, this space overhead can be justified by the higher performance (2.1× speedup) delivered by LerGAN, compared with PRIME using the same space.

VII. Related Work

3D Network on Chip (NoC) There are several prior studiess on 3D NoC [38] [29] [51] [8] [1], 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 [67] [45] [69] [57] [3] and ASICs [18] [23] [68] [40] [12] [43] [55] [44] [2]. Diannao family was proposed based on Near-Data Processing (NDP) [11] [13] [16] [41], 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 [56] proposed a pipeline solution to accelerate inference of CNNs. PipeLayer [59] 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. [47] 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 [5] [4] [6], by removing zeros in GAN training. Chen et.al. proposed ReGAN, a ReRAM-based GAN accelerator using pipeline [10] design. Our LerGAN design is PIM-based and flexible to handle all zero-related scenarios in GAN training.

VIII. Conclusions

This paper proposes a PIM-based GAN accelerator: LerGAN, which achieves low data movement and zero-free computation. LerGAN has two main techniques: (1) Zero-Free Data Reshaping ($ZFDR$) designed for ReRAM-based PIM to remove computations with zeros; and (2) reconfigurable 3D connection in PIM which removes the bottleneck of long data movement. LerGAN also combines these techniques with minor modifications of software and memory controller. Experiments show that both these techniques are critical in accelerating GAN training. Experiments show that LerGAN achieves 47.2×, 21.42× and 7.46× speedup over FPGA-based GAN accelerator, GPU platform and PRIME respectively. Moreover, LerGAN achieves 9.75×, 7.68× energy saving on average over GPU platform, PRIME respectively, and has 1.04× energy consuming over FPGA-based GAN accelerator.

Acknowledgment

This work is supported by the National Major Project of Scientific Instrument of National Natural Science Foundation of China (Grant No.61327902), National Key Research & Development Projects of China (Grant No.2018YFB1003301), and in part by NSF grants 1822989, 1822459, 1527535, 1423090, and 1320100.

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 *Embedded Multicore Socs (MCSoC), 2012 IEEE 6th International Symposium on*. IEEE, 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," in *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3. IEEE Press, 2016, pp. 1–13.

[3] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer cnn accelerators," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 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 *IEEE/ACM International Symposium on Computer Architecture (ISCA)*, 2018.

[5] Y. Amir, S. Kambiz, E. Hadi, and S. K. Nam, "A simd-mimd acceleration with access-execute decoupling for generative adversarial networks," in *SysML Conference (SysML)*, 2018.

[6] Y. Amir, B. Michael, K. Behnam, G. Soroush, S. Kambiz, E. Hadi, and S. K. Nam, "Fxgan: A unified mimd-simd fpga acceleration with decoupled access-execute units for generative adversarial networks," in *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2018.

[7] M. N. Bojnordi and E. Ipek, "Memristive boltzmann machine: A hardware accelerator for combinatorial optimization and deep learning," in *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*. IEEE, 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 *Networks-on-Chip (NOCS), 2010 Fourth ACM/IEEE International Symposium on*. IEEE, 2010, pp. 223–230.

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

[10] F. Chen, L. Song, and Y. Chen, "Regan: a pipelined reram-based accelerator for generative adversarial networks," in *Proceedings of the 23rd Asia and South Pacific Design Automation Conference*. IEEE Press, 2018, pp. 178–183.

[11] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "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 Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2017.

[13] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun *et al.*, "Dadiannao: A machine-learning supercomputer," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2014, pp. 609–622.

[14] M. Cheng, L. Xia, Z. Zhu, Y. Cai, Y. Xie, Y. Wang, and H. Yang, "Time: A training-in-memory architecture for memristor-based deep neural networks," in *Proceedings of the 54th Annual Design Automation Conference 2017*. ACM, 2017, p. 26.

[15] 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 *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3. IEEE Press, 2016, pp. 27–39.

[16] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: Shifting vision processing closer to the sensor," in *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3. ACM, 2015, pp. 92–104.

[17] D. Erhan, Y. Bengio, A. Courville, P.-A. Manzagol, P. Vincent, and S. Bengio, "Why does unsupervised pre-training help deep learning?" *Journal of Machine Learning Research*, vol. 11, no. Feb, pp. 625–660, 2010.

[18] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun, "Neuflow: A runtime reconfigurable dataflow processor for vision," in *Computer Vision and Pattern Recognition Workshops (CVPRW), 2011 IEEE Computer Society Conference on*. IEEE, 2011, pp. 109–116.

[19] 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 *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*. IEEE, 2015, pp. 283–295.

[20] C. Finn, I. Goodfellow, and S. Levine, "Unsupervised learning for physical interaction through video prediction," in *Advances in neural information processing systems*, 2016, pp. 64–72.

[21] A. Ghosh, B. Bhattacharya, and S. B. R. Chowdhury, "Sad-gan: Synthetic autonomous driving using generative adversarial networks," *arXiv preprint arXiv:1611.08788*, 2016.

[22] A. Graves, A.-r. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *Acoustics, speech and signal processing (icassp), 2013 ieee international conference on*. IEEE, 2013, pp. 6645–6649.

[23] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: efficient inference engine on compressed deep neural network," in *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*. IEEE, 2016, pp. 243–254.

[24] T. Hastie, R. Tibshirani, and J. Friedman, "Unsupervised learning," in *The elements of statistical learning*. Springer, 2009, pp. 485–585.

[25] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath *et al.*, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal processing magazine*, vol. 29, no. 6, pp. 82–97, 2012.

[26] C.-W. Hsu, I.-T. Wang, C.-L. Lo, M.-C. Chiang, W.-Y. Jang, C.-H. Lin, and T.-H. Hou, "Self-rectifying bipolar tao x/tio 2 rram with superior endurance over 10 12 cycles for 3d high-density storage-class memory," in *VLSI Technology (VLSIT), 2013 Symposium on*. IEEE, 2013, pp. T166–T167.

[27] G. Huang, S. Song, J. N. Gupta, and C. Wu, "Semi-supervised and unsupervised extreme learning machines," *IEEE transactions on cybernetics*, vol. 44, no. 12, pp. 2405–2417, 2014.

[28] N. P. Jouppi, A. B. Kahng, N. Muralimanohar, and V. Srinivas, "Cacti-io: Cacti with off-chip power-area-timing models," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 23, no. 7, pp. 1254–1267, 2015.

[29] J. Kim, C. Nicopoulos, D. Park, R. Das, Y. Xie, V. Narayanan, M. S. Yousif, and C. R. Das, "A novel dimensionally-decomposed router for on-chip communication in 3d architectures," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2, pp. 138–149, 2007.

[30] T. Kim, M. Cha, H. Kim, J. Lee, and J. Kim, "Learning to discover cross-domain relations with generative adversarial networks," *arXiv preprint arXiv:1703.05192*, 2017.

[31] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[32] H. Kwon, A. Samajdar, and T. Krishna, "Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects," 2018.

[33] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, p. 436, 2015.

[34] C. Ledig, L. Theis, F. Huszár, J. Caballero, A. Cunningham, A. Acosta, A. Aitken, A. Tejani, J. Totz, Z. Wang *et al.*, "Photo-realistic single image super-resolution using a generative adversarial network," *arXiv preprint*, 2016.

[35] H. Lee, Y. Chen, P. Chen, P. Gu, Y. Hsu, S. Wang, W. Liu, C. Tsai, S. Sheu, P. Chiang *et al.*, "Evidence and solution of over-reset problem for hfo x based resistive memory with sub-ns switching speed and high endurance," in *Electron Devices Meeting (IEDM), 2010 IEEE International*. IEEE, 2010, pp. 19–7.

[36] M.-J. Lee, C. B. Lee, D. Lee, S. R. Lee, M. Chang, J. H. Hur, Y.-B. Kim, C.-J. Kim, D. H. Seo, S. Seo *et al.*, "A fast, high-endurance and scalable non-volatile memory device made from asymmetric ta 2 o 5-x/tao 2- x bilayer structures," *Nature materials*, vol. 10, no. 8, p. 625, 2011.

[37] 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 *Proceedings of the 52nd Annual Design Automation Conference*. ACM, 2015, p. 13.

[38] F. Li, C. Nicopoulos, T. Richardson, Y. Xie, V. Narayanan, and M. Kandemir, "Design and management of 3d chip multiprocessors

using network-in-memory," in *ACM SIGARCH Computer Architecture News*, vol. 34, no. 2. IEEE Computer Society, 2006, pp. 130–141.

[39] 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 *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*. IEEE, 2016, pp. 1–6.

[40] R. LiKamWa, Y. Hou, J. Gao, M. Polansky, and L. Zhong, "Redeye: analog convnet image sensor architecture for continuous mobile vision," in *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3. IEEE Press, 2016, pp. 255–266.

[41] D. Liu, T. Chen, S. Liu, J. Zhou, S. Zhou, O. Teman, X. Feng, X. Zhou, and Y. Chen, "Pudiannao: A polyvalent machine learning accelerator," in *ACM SIGARCH Computer Architecture News*, vol. 43, no. 1. ACM, 2015, pp. 369–381.

[42] M.-Y. Liu and O. Tuzel, "Coupled generative adversarial networks," in *Advances in neural information processing systems*, 2016, pp. 469–477.

[43] S. Liu, Z. Du, J. Tao, D. Han, T. Luo, Y. Xie, Y. Chen, and T. Chen, "Cambricon: An instruction set architecture for neural networks," in *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3. IEEE Press, 2016, pp. 393–405.

[44] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li, "Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks," in *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*. IEEE, 2017, pp. 553–564.

[45] Y. Ma, Y. Cao, S. Vrudhula, and J.-s. Seo, "Optimizing loop operation and dataflow in fpga acceleration of deep convolutional neural networks," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2017, pp. 45–54.

[46] M. Mao, Y. Cao, S. Yu, and C. Chakrabarti, "Optimizing latency, energy, and reliability of 1t1r reram through cross-layer techniques," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 6, no. 3, pp. 352–363, 2016.

[47] S. Mingcong, Z. Jiaqi, C. Huixiang, and L. Tao, "Towards efficient microarchitectural design for accelerating unsupervised gan-based deep learning," in *High Performance Computer Architecture (HPCA), 2018 IEEE 24st International Symposium on*, 2018.

[48] S. Mittal, M. Poremba, J. Vetter, and Y. Xie, "Exploring design space of 3d nvm and edram caches using destiny tool," *Oak Ridge National Laboratory, USA, Tech. Rep. ORNL/TM-2014/636*, 2014.

[49] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2007, pp. 3–14.

[50] J. Ngiam, A. Khosla, M. Kim, J. Nam, H. Lee, and A. Y. Ng, "Multimodal deep learning," in *Proceedings of the 28th international conference on machine learning (ICML-11)*, 2011, pp. 689–696.

[51] D. Park, S. Eachempati, R. Das, A. K. Mishra, Y. Xie, N. Vijaykrishnan, and C. R. Das, "Mira: A multi-layered on-chip interconnect router architecture," in *Computer Architecture, 2008. ISCA'08. 35th International Symposium on*. IEEE, 2008, pp. 251–261.

[52] D. Pathak, P. Krahenbuhl, J. Donahue, T. Darrell, and A. A. Efros, "Context encoders: Feature learning by inpainting," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 2536–2544.

[53] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3, pp. 24–33, 2009.

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

[55] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3. IEEE Press, 2016, pp. 267–278.

[56] 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," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 14–26, 2016.

[57] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From high-level deep neural models to fpgas," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 2016, pp. 1–12.

[58] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[59] L. Song, X. Qian, H. Li, and Y. Chen, "Pipelayer: A pipelined reram-based accelerator for deep learning," in *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*. IEEE, 2017, pp. 541–552.

[60] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," in *International conference on machine learning*, 2013, pp. 1139–1147.

[61] W. R. Tan, C. S. Chan, H. Aguirre, and K. Tanaka, "Artgan: artwork synthesis with conditional categorial gans," *arXiv preprint arXiv:1702.03410*, 2017.

[62] R. Wang, A. Cully, H. J. Chang, and Y. Demiris, "Magan: Margin adaptation for generative adversarial networks," *arXiv preprint arXiv:1704.03817*, 2017.

[63] H. Wu, S. Zheng, J. Zhang, and K. Huang, "Gp-gan: Towards realistic high-resolution image blending," *arXiv preprint arXiv:1703.07195*, 2017.

[64] 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 *Advances in Neural Information Processing Systems*, 2016, pp. 82–90.

[65] C. Xu, D. Niu, N. Muralimanohar, R. Balasubramonian, T. Zhang, S. Yu, and Y. Xie, "Overcoming the challenges of crossbar resistive memory architectures," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2015, pp. 476–488.

[66] 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 *Electron Devices Meeting (IEDM), 2012 IEEE International*. IEEE, 2012, pp. 10–4.

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

[68] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-x: An accelerator for sparse neural networks," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 2016, pp. 1–12.

[69] 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," *arXiv preprint arXiv:1606.06234*, 2016.