Efficient Process-in-Memory Architecture Design for Unsupervised GAN-based Deep Learning using ReRAM

Fan Chen, Linghao Song, Hai "Helen" Li
Department of Electrical and Computer Engineering, Duke University, Durham, North Carolina, USA
{fan.chen,linghao.song,hai.li}@duke.edu

ABSTRACT

The ending of Moore's Law makes domain-specific architecture as the future of computing. The most representative is the emergence of various deep learning accelerators. Among the proposed solutions, resistive random access memory (ReRAM) based process-in-memory (PIM) architecture is anticipated as a promising candidate because ReRAM has the capability of both data storage and in-situ computation. However, we found that existing solutions are unable to efficiently support the computational needs required by the training of unsupervised generative adversarial networks (GANs), due to the lack of the following two features: 1) Computation efficiency: GAN utilizes a new operator, called transposed convolution. It inserts massive zeros in its input before a convolution operation, resulting in significant resource under-utilization; 2) Data traffic: The data intensive training process of GANs often incurs structural heavy data traffic as well as frequent massive data swaps. Our research follows the PIM strategy by leveraging the energy-efficiency of ReRAM arrays for vector-matrix multiplication to enhance the performance and energy efficiency. Specifically, we propose a novel computation deformation technique that can skip zero-insertions in transposed convolution for computation efficiency improvement. Moreover, we explore an efficient pipelined training procedure to reduce on-chip memory access. The implementation of related circuits and architecture is also discussed. At the end, we present our perspective on the future trend and opportunities of deep learning accelerators.

CCS CONCEPTS

• **Hardware → Hardware accelerators**.

KEYWORDS

GAN; process-in-memory; ReRAM; unsupervised learning

ACM Reference Format:
Fan Chen, Linghao Song, Hai "Helen" Li. 2019. Efficient Process-in-Memory Architecture Design for Unsupervised GAN-based Deep Learning using ReRAM. In *Great Lakes Symposium on VLSI 2019 (GLSVLSI '19), May 9–11, 2019, Tysons Corner, VA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3299874.3319482

GLSVLSI '19, May 9–11, 2019, Tysons Corner, VA, USA

ACM ISBN 978-1-4503-6252-8/19/05…$15.00
https://doi.org/10.1145/3299874.3319482

1 INTRODUCTION

Generative adversarial network (GAN) based unsupervised deep learning has recently been extensively deployed in various modern computer vision tasks [11, 12, 20]. GANs have demonstrated great potential in extending deep learning into a wider range of domains [8, 13, 30] where sufficient labeled training data is generally unavailable or at least costly to obtain. In contrast to traditional supervised learning which requires a large amount of labeled training data, GANs are capable to learn reusable feature representations from unlabeled samples. In a GAN model, two deep neural networks (DNNs) – a generator and a discriminator, are simultaneously trained against each other via an adversarial process: the generator captures the data distribution and attempts to generate synthetic samples, while the discriminator implements a binary classifier to differentiates the samples generated by a generator against real data. This learning process is performed iteratively until we obtain a generator with strong generative capability and a discriminator with high classification accuracy.

Given the abundant DNN accelerator designs [4, 5, 22, 23], it was believed that the computational needs of GANs can be accommodated by conventional DNN accelerators. As such, for a long period, there was a lack of hardware accelerators for GAN execution. By studying the algorithmic characteristics of GAN models, researchers realized that the training process of GANs involves a special mathematical operator (e.g. transposed convolution), more complex training phases, and massive computation and data movement. These problems, however, have been exacerbated in recent years because of the continuously increasing trend of deploying larger and deeper networks in GANs, e.g. 101-layer ResNet generator and discriminator [12].

Although some designs [24, 28, 29] have recently been proposed for GANs acceleration, these solutions are subject to various constraints. For instance, the dataflow optimization approach in [24] limits the shape and structure of the generator and/or discriminator being trained in a GAN model and fails to support many popular GAN variations. FlexiGAN [28] and GANAX [29] require a sophisticated interleaving MIMD and SIMD microarchitecture to support their proposed dataflow. In this paper, we present our research on efficient architecture design for GANs. We utilize the energy efficiency of the emerging resistive random access memory (ReRAM) array for vector-matrix multiplication, and the design follows the Process-In-Memory (PIM) concept to improve performance and energy efficiency. Specifically, we propose a novel computation deformation technique that can skip zero-insertions in transposed convolution for computation efficiency. Moreover, we explore an efficient pipelined training procedure to reduce on-chip memory access. In summary, we make the following key contributions:

- We completely eliminate inserted zeros in transposed convolution by a novel computation deformation. The optimized computing model exhibits improved computation efficiency and resource utilization.
- We analyze the general training process in GAN and proposed a pipeline architecture that improves system throughput by leveraging structural layer-wise computation. The architecture uses ReRAM array to perform calculations directly, without the need for additional processing units. The ReRAM memory is also used as buffers to store intermediate results. This design greatly reduces data movement and energy consumption by preventing data from being transferred across the memory hierarchy.
- To further improve the training efficiency, we propose *spatial parallelism* and *computation sharing* by exploring the data dependency and the trade-off between parallelism and hardware resource.
- We use various applications and the most recent GAN models on distinct benchmarks to evaluate the effectiveness of the proposed design. Our experimental results show significant acceleration and energy savings compared to prior arts.

2 BACKGROUND

In this section, we describe the background of GAN and some of the concepts involved in its development. We also briefly introduce the emerging ReRAM memory and its basic operations.

2.1 GAN Basics

GAN is an unsupervised training paradigm based on an adversarial game between a generator (G) and a discriminator (D) as illustrated in Figure 1. Both D and G typically are modeled as DNNs and trained simultaneously. In general, G is optimized to produce good samples from noise to simulate real data distribution, while D is a binary classifier trained by distinguishing between real samples and generated samples. The objective of GAN training can be represented as a minimax game with these two players:

$$\min_G \max_D V(D,G) = E_{x\sim P_{data}}[logD(x)] + E_{z\sim P_z}[log(1-D(G(z)))] \quad (1)$$

In image processing, most of the GAN variations are (or at least partially) based on the deep convolutional generative adversarial networks (DCGAN) [20]. Therefore, DCGAN is the targe model for GAN acceleration. In DCGAN, D uses discriminative convolutions to "downsample" the input to produce a classification. Instead, G takes a uniform noise distribution as input, then projects it onto a small spatial extent convolutional representation with many feature maps which serves as the beginning of a series of transposed convolutional (TCONV) layers. The results are eventually converted

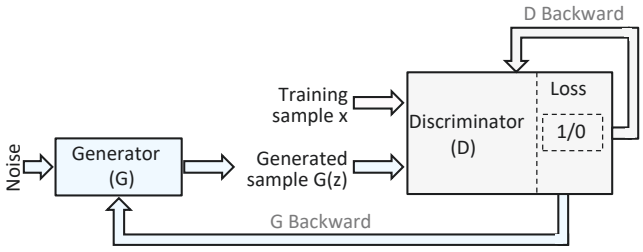

Figure 1: A GAN system.

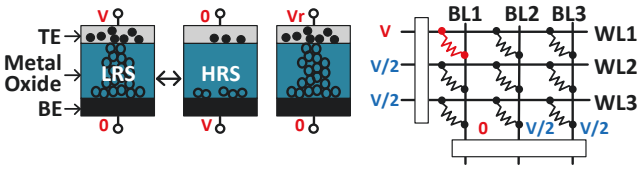

Figure 2: The ReRAM basics.

into high-dimension images. Different from traditional convolution operation (CONV), TCONV inserts many zeros into its input feature maps and then performs the up-sampling convolutional computation. The output feature maps of TCONV are larger than its input. TCONV can essentially be implemented using the CONV operator. However, it requires adding several columns and rows of zeros to the inputs, resulting in severe resource under-utilization.

Activation layers are typically cascaded after convolutional layers to act as a threshold by performing a non-linear function. The batch normalization layer is another important component. It is applied before activation layers to stabilize the training process of GANs.

2.2 ReRAM Basics

ReRAM is an emerging type of non-volatile memory that stores information with cell resistances. As demonstrated in Figure 2(a), each ReRAM cell has a metal oxide layer sandwiched between top electrode (TE) and bottom electrode (BE). The resistance of a ReRAM cell can be programmed by applying a current or voltage with proper pulse-width or magnitude between TE and BE. The data stored in a cell can be represented by the resistance level accordingly: a low resistance state (LRS) denotes logic bit '1', while a high resistance state (HRS) represents logic bit '0'. For read operations, a small sense voltage is applied across the device, and then the cell resistance can be determined based on the magnitude of the detected current to further obtain a corresponding logic value stored in the cell.

Recent work has demonstrated a high density ReRAM crossbar implementation. Figure 2(b) illustrates the array biasing scheme when writing '1' on a 3×3 ReRAM crossbar array. The wordline (WL) and bitline (BL) of the selected cell are driven to V and 0 respectively, while unselected WLs and BLs are set to $V/2$. To write a '0', the biasing polarity must be alternated. For a read, the selected WL is charge to a small read voltage V_r and the current changes on the corresponding BL will be sensed to determine the resistance.

3 PRIOR ARTS

A major challenge in GAN acceleration is the high demand for computing resource and large data transfer because a GAN model consists of two DNNs. Moreover, GANs training involves more complex operations and more computing phases compared to traditional supervised deep learning. For instance, a direct adoption of the accelerator PipeLayer [23] designed for CNN training to implement GAN consumes 3× latency and hardware compared to its CNN counterpart. Moreover, existing schemes mainly focus on accelerating discriminative convolution. The generator core leverages TCONV, a fundamentally different operator that first inserts many zeros into its input feature maps and then performs the up-sampling CONV computation. This operator manifests unique challenges for hardware acceleration, which is not well studied in prior arts.

Table 1: Comparison of GAN Accelerators

Accelerator	Year	Description	Architecture Optimization
[32]	2017	TCONV acceleration on FPGA	reversing loop; stride hole skipping
FCN-engine [27]	2018	CMOS TCONV inference accelerator	reschedule TCONV data path with on-chip buffer
Red [10]	2019	ReRAM TCONV inference accelerator	pixel-wise mapping; zero-skipping data flow
[24]	2018	CMOS GAN accelerator	time multiplexed design; zero skipping
FlexiGAN [28]	2018	GAN acceleration on FPGA	TCONV computation reorder; architecture optimizer on FPGA
GANAX [29]	2018	CMOS GAN acclerator	TCONV computation reorder; MIMD-SIMD; decoupled access/execute μengine

We summarize recent works on GAN acceleration in Table 1. The work in [32] attempted to accelerate TCONV on FPGA. FCN-engine [27] and Red [10] implemented a fully convolutional accelerator that can handle both CONV and TCONV operations using CMOS and ReRAM, respectively. These three works only focus on inference task but do not support the more sophisticated and intensive training function. Song et al. [24] proposes a CMOS-based GAN training accelerator where zero operands can be skipped by carefully mapping the network onto its process elements (PEs). The proposed dataflow, however, imposes strong limitations on the shape and topology of G and/or D, making it less flexible to serve various GAN variations. FlexiGAN [28] and GANAX [29] reorder the output computation and allocate computing rows with similar patterns of zeros to adjacent PEs for efficient TCONV execution. To support such reorganized dataflow, these two works propose specific architectures to realize interleaving MIMD and SIMD operations on FPGA and ASIC, respectively. FlexiGAN [28] and GANAX [29] both implement decoupled access and execute micro-engines, resulting in increased design complexity.

4 EFFICIENT PIM ARCHITECTURE FOR GAN

In this section, we present our research on two ReRAM-based accelerator architectures for GANs, namely ReGAN [2] and ZARA [3]. We analyze the GAN training process and then present ReGAN – a training architecture that implements both the inter-layer and intro-layer parallelism in a pipelined fashion for improving the throughput. Our latest work, ZARA, focuses on optimizing the GAN calculation model and completely eliminating the insertion of zeros in the TCONV convolution so as to improve computational efficiency and alleviate under-utilization of resources.

4.1 Training Process

Figure 3 demonstrates the complex training phases in a three-layer GAN. A discriminator and a generator are both modeled as a DNN and are alternately trained until they reach equilibrium or meet predetermined constraints. D is trained to differentiate generated fake samples from genuine data. The dataflow of training D with

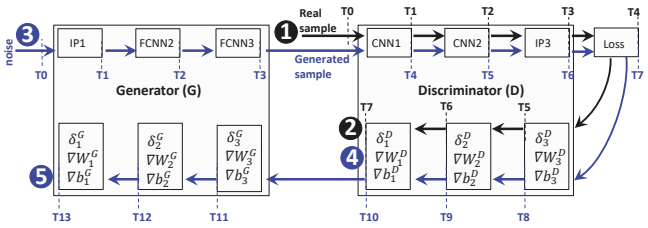

Figure 3: Training processes in a GAN [2].

real samples is denoted by ❶~❷. When a training sample enters the first layer in D at $T0$, it continuously flows through all layers in D in the forward direction. A loss function is then calculated at $T4$ based on the exact label ('1' for the training sample). Finally, the error and partial derivatives are propagated back to the first layer of D and stored. This process requires 7 logical cycles.

The data stream of training D with the generated samples is represented by ❸~❹. Here, G is connected to D to form a large network. G maps a random vector to a sample with the same dimension as a real sample. The sample is processed through all layers in D, and a loss function is finally executed using the exact label ('0' for the generated sample). Similarly, the partial derivative propagates back to the first layer of D at $T10$. In $T11$, the two sets of previously calculated and stored derivatives are added together and used to update the weight of D. It takes a total of $7 + 10 + 1 = 18$ cycles. In this process, G is used but not updated.

G is optimized to generate samples that simulate the distribution of real data, thereby tricking D into misprediction. The data stream is represented by ❸~❺, which is similar to ❸~❹, except that: 1) The error is calculated in $T7$ using a different label ('1' for the generated sample), since the target of G is to mislead D; 2) The error is always propagated back to the first layer of G; and 3) The weight of G is updated in $T14$, while D is fixed. Thus, it takes 14 cycles to process one sample with G. Note that the cycles in Figure 3 (e.g. $T0$, $T1$, ...) are logical cycles, which could take several physical clock cycles depending on the implementation details.

4.2 Pipelined Design & Compute Optimization

In practice, the training data are processed in batches of size B (e.g., 64 or 128). B is a hyperparameter which controls the number of training samples to process before updating the internal weights of a DNN. For example, when performing DNN training with $B = 64$, 64 inputs in the same batch are processed continuously. The backpropagation error caused by each input is stored and applied only at the end of the batch. There is no dependency among the data inputs in a same batch. The training process can be essentially pipelined to increase system throughput.

We use the same GAN model as demonstrated in Figure 3 to explain the pipelined training execution in ReGAN. Assuming the discriminator has L_D layers, the generator has L_G layers, and batch size is B. In order to update D, we first feed the real samples into D in a pipelined fashion. Both the forward and backward path take L_D cycles. It also consumes 1 cycle for calculating the loss function and $B - 1$ cycles to drain the previous batch from the pipeline. Therefore, it takes a total of $L_D + L_D + 1 + B - 1$ cycles. To train D on generated fake samples, it takes $L_G + L_D + L_D + 1 + B - 1$. Finally, 1 addtional cycle is required to update D. Similarly, training

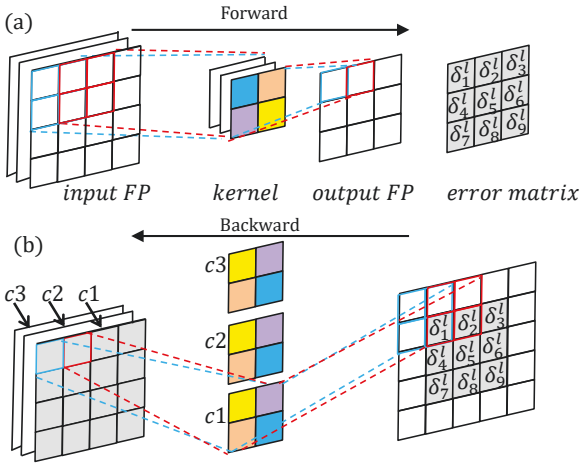

Figure 4: Computation in CONV. (a) Forward pass and error matrix; (b) Error backpropagation.

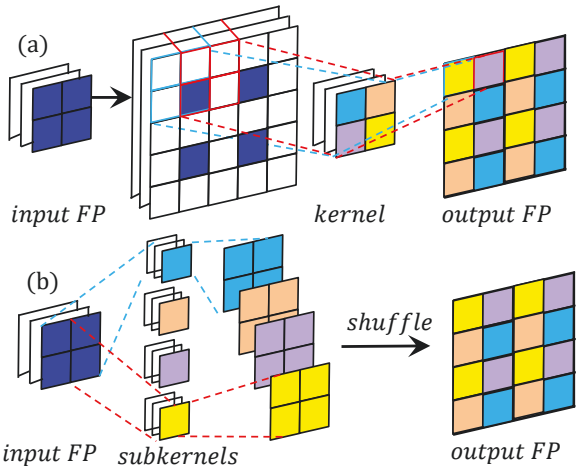

Figure 5: (a) TCONV operation; (b) TCONV deformation.

G requires $2L_G + 2L_D + B + 1$ cycles. As a comparison, if the training process is performed without a pipeline, D and G training consume $(4L_D + L_G + 2)\times B$ cycles and $(2L_G + 2L_D + 1)\times B$ cycles, respectively. Furthermore, ReGAN proposes *spatial parallelism* and *computation sharing* to further optimize pipeline performance. The design principle behind both solutions is to take advantage of the trade-off between system performance and hardware requirement.

Spatial parallelism (SP). Since D remains unchanged in ❶~❷ and ❸~❹, we can duplicate D for two copies and perform training processes ❶~❷ and ❸~❹ in parallel. In this way, the total latency is as the same as the latency of ❸~❹ for the spatial parallelism hides the latency of ❶~❷.

Computation sharing (CS). The data stream of training D with generated samples (❸~❹) and training G (❸~❺) share the same forward path. However, they are distinguished from the definition of loss function, and hence have different backpropagation data path. With this observation, we propose to co-train D and G by duplicating the memory for intermediate computation storage (e.g. memory to store the error and partial derivatives). In this case, the training of D and G share the forward path $T0 - T6$. At cycle $T7$, two backward branches are computed in parallel. The weights of G are updated at T14 and D can be updated at $T11$.

4.3 Dataflow Optimization

Table 2 summarizes the symbols used to explain the GAN calculation. Note that ∘ denotes elementwise multiplication. We do not provide the complete description of these parameters or mathematical model of the GAN training process due to page limits. We recommend interested readers to refer [23] and [2].

Figure 4(a) visualizes the main calculations of the forward path in CONV corresponding to Equations (1) and (2) in Table 2. As shown,

Table 2: Notations used for explaining GAN training process.

Symbols/equations		Description
$u_l = W_l d_{l-1} + b_l$	(1)	forward function
$d_l = h(u_l)$	(2)	activation function
$\delta_{l-1} = {W_l}^T \delta_l \circ h'(u_l)$	(3)	error backpropagation
$\nabla W_l = d_{l-1} {\delta_l}^T$	(4)	weight partial derivative
$\nabla b_l = \delta_l$	(5)	bias partial derivative

a set of kernels are convoluted with the input or the output of the previous layer. The sliding convolution window is repeated for all possible positions in the input feature map (FP), and the results are stored in the output FP. These results are then used as the inputs of the following layer. Once the forward layerwise computation finished, a *cost function* $\mathcal{J}$ is defined to quantitatively evaluate the difference between the network output and its expected output. The error for layer l is defined as $\delta_l = \frac{\partial J}{\partial b_l}$ which will be propagated backwards to the previous layer as depicted in Figure 4(b). The backpropagation process can be described by Equation (3)~(5) in Table 2.

Compared to CONV, TCONV first transforms its input to a larger representation by zero paddings around the image and zero insertions between adjacent rows/columns and then performs CONV on the expanded input. For instance, Figure 5(a) illustrates the zero-insertion step for a 2×2 input FP with $stride = 1$ and $padding = 1$. A kernel convolves with this transformed input FP and correspondingly generate an output FP with a size of 4×4. In this example, 21 zeros are inserted, that is, only 4/25 the source operands are valid input and contributes to the final results. Previous work have shown that executing GAN by following conventional convolution dataflow illustrated in Figure 5(a) leads to a low computational efficiency because generally more than 60% of the computations are multiplications and additions with zero operands due to the zero-insertion step in TCONV [29].

To address this issue, we propose ZARA – an ReRAM-based GAN accelerator. Through the novel deformation of TCONV, ZARA can completely skip over zero-insertions in both forward and backward computation phases. Taking the example in Figure 5(a), we observed that the convolution between the expanded FP and the kernels only contains four distinct valid patterns. The elements in odd output rows only related to a subset of kernel weights (e.g. the second row in this example), whereas the even output rows use a different kernel weights subset (e.g. the first row) for their computations. Building upon this observation, we introduce a series of dataflow optimization covers both the forward and backward phases in TCONV to mitigate the aforementioned inefficiency in TCONV execution. For simplicity, we only demonstrate the data optimization in a forward phase in Figure 5(b).

Table 3: Benchmarks (C: convolution layer; T: transposed convolution layer; F: fully connected layer.).

ReGAN				ZARA			
Model	**Dataset**	**D Topology**	**G Topo.**	**Model**	**Dataset**	**D Topo.**	**G Topo.**
MLP1	mnist [15]	2/4-layer MLP	2/4-layer MLP	FCN [27]	POSVAL VOC [9]	5C2F	N/A
MLP2	mnist [15]	3C1F	4-layer MLP	DCGAN [20]	LSUN [31]	4C1F	1F4T
GAN1	CIFAR-10 [14]	4C1F	1F4T	iGAN [21]	CIFAR-10 [14]	3C1F	1F3T
GAN2	celebA [18]/LSUN [31]	5C1F	1F5T	3DGAN [26]	IKEA [16]	5C1F	1F4T
ResnetGAN	CIFAR-10 [14]	[12]	[12]	ArtGAN [25]	CIFAR-10 [14]	6C1F	1F5T

4.4 Hardware Implementation

Normally, ReRAM-based DNN accelerators use ReRAM arrays as precessing units for vector-matrix multiplication. As demonstrated in Figure 6, the weight matrix is written as cell conductance states in the crossbar. Input FPs are applied to the WLs as voltage levels. In this way, all BLs in the crossbar produce the current sums simultaneously with the same voltage along WLs, and hence a vector-matrix multiplication is performed in each step. . Representative ReRAM-based DNN accelerators include PRIME [7], ISAAC [22], and PipeLayer [23].

Similar to [7, 23], ReGAN [2] partitions the ReRAM main memory into three regions: *memory subarrays* for data storage, *full function subarrays* for both in-site VMM and data storage, and *buffer subarrays* for storing the intermediate results between layers (e.g., generated images, data required for compute partial derivatives, etc.). The *buffer subarrays* are connected to the *full function subarrays* through private high-throughput local data ports, so that buffer accesses do not consume the bandwidth of memory subarrays. It is worth mentioning that we adopted the spike-based scheme [17] in ReGAN to minimize the energy overhead of power-hungry analog-digital converters. ReGAN realizes the non-linear activation function by using a *look-up table*. Moreover, we implemented various function units (e.g. *substractor, shift & add, etc.*) to support the related computation involved during the training process.

ZARA leverages the ReRAM crossbar to perform in-memory GAN execution. However, different from ReGAN, ZARA is a domain-specific standalone accelerator. More specific, ZARA aims to solve the inconsistent computational delay caused by TCONV deformation, which breaks the pipeline execution model. A *mapper* and a *scheduler* are designed to effectively map and schedule the operations of the optimized dataflow onto PEs to eliminate the inconsistency in execution time. ZARA PIM is implemented in a tiled

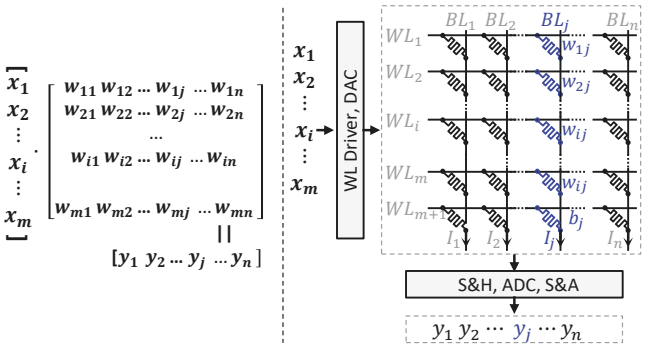

Figure 6: Mapping a VMM onto a ReRAM crossbar.

computation architecture. The top level is ZARA nodes. Each ZARA node consists of a ReRAM memory to store input/output values, a *mapper* and a *scheduler* that map the optimized dataflow onto target PEs and control the computation flow respectively, an IO interface to communicate with other ZARA nodes, and a number of PEs connected via on-chip mesh. Each PE contains multiple vector-matrix multipliers, a ReRAM buffer to cache temporal data, activation units to perform nonlinear function, and output register to aggregate results, all connected with a shared bus. A PE also has simple algorithm and logic units (sALU). The crossbar arrays within a vector-matrix multipliers share a driver by global wordline. At a time, only one crossbar array in a vector-matrix multipliers can be activated and used to perform in-situ VMM. The results can be selected via a bitline multiplex. The details of these components are described in [3].

5 EVALUATION

Table 3 summarizes the topological structures of evaluated networks and dataset in ReGAN [2] and ZARA [3]. To evaluate ReGAN, we built four MLP GANs which has a multiLayer perceptron (MLP) generator and a MLP/CNN discriminator. We test them on mnist [15]. For CIFAR-10 [14] and LSUN [31], we built a symmetric GAN with four CNN/FCNN layers in G/D. In particular, we also selected the 101-layer ResNet GAN in [12] as benchmarks on CIFAR-10 [14]. For CelebA [18] and LSUN [31], we built a five-layer CNN/FNN GAN as benchmarks. We compare ReGAN against the GTX1080 GPU platform. Our experimental results show that ReGAN can achieve 240× performance speedup compared to GPU platform averagely, with an average energy saving of 94×.

We evaluated the ZARA architecture against four counterparts [2, 23, 27, 29] using five state-of-the-art GANs. For comparison purpose, we also include fully convolution network [27] in evaluation. Experimental results show that ZARA can improve the training performance of GAN by averagely 1.6 × ~23× over CMOS-based GAN accelerators. Compared to state-of-the-art ReRAM-based accelerator designs, ZARA also provides 1.15 × ~2.1× performance improvement.

6 DISCUSSION AND FUTURE WORK

In recent years, we have seen a continuously increasing trend of deploying emerging frameworks such as generative adversarial networks with hierarchical structure into various complex application domains. These models are often trained and tested with huge dataset, and therefore, consume high computation and storage resources. Moreover, data processing in these models often incurs heavy data traffic as well as frequent massive data interaction, which makes the memory wall issue even worse. Almost all of the current machine learning acceleration schemes, however, focus

on the execution of DNN inference but pay less attention on the more sophisticated and intensive training phases.

In this paper, we review the current advance in accelerator designs for GANs and point out their limitations. Then we introduce our solutions. Different from previous research, we use ReRAM-based in-memory computation since ReRAM is capable of both computation and storage, significantly reducing the data movement. ReGAN [2] explored an efficient pipelined training procedure to reduce on-chip memory access. *Spatial parallelism* and *computation sharing* were proposed to further improve performance. ZARA [3] proposed a novel computation deformation technique that can skip zero-insertions in TCONV. A dataflow mapper and an operation scheduler were also implemented to support the proposed execution model.

There is still much work unsolved on ReRAM-based GANs acceleration. For example, many current studies on ReRAM-based accelerators assume ideal ReRAM cells. However, real devices suffers from severe process variations [19], circuit noise [1] and short retention time and endurance [6] issues. How to realize reliable accelerators using the real device model is a common challenge, not only for ReRAM-based designs but also all emerging memory based solutions. We are encouraged by the broad design space enabled by ReRAM crossbar arrays and we also intend to explore the future opportunity of performing GANs learning in end devices because it would be desirable to have the intelligent edge devices capable of adaptively learning and tuning their parameters to achieve a desirable accuracy.

ACKNOWLEDGMENTS

This work was supported in part by NSF 1725456, NSF 1744082 and DOE DE-SC0017030. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of grant agencies or their contractors.

REFERENCES

[1] Stefano Ambrogio, Simone Balatti, A Cubeta, A Calderoni, N Ramaswamy, and Daniele Ielmini. 2013. Understanding switching variability and random telegraph noise in resistive RAM. In *2013 IEEE International Electron Devices Meeting*.

[2] F. Chen, L. Song, and Y. Chen. 2018. ReGAN: A pipelined ReRAM-based accelerator for generative adversarial networks. In *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*.

[3] Fan Chen, Linghao Song, and Yiran Chen. 2019. ZARA: A Novel Zero-free Dataflow Accelerator for Generative Adversarial Networks in 3D ReRAM. In *Proceedings of the 56th annual design automation conference*. ACM.

[4] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. DianNao: A Small-footprint High-throughput Accelerator for Ubiquitous Machine-learning. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*.

[5] Y. Chen, T. Krishna, J. S. Emer, and V. Sze. 2017. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE Journal of Solid-State Circuits* (2017).

[6] Yang Yin Chen, Ludovic Goux, Sergiu Clima, Bogdan Govoreanu, Robin Degraeve, Gouri Sankar Kar, Andrea Fantini, Guido Groeseneken, Dirk J Wouters, and Malgorzata Jurczak. 2013. Endurance/retention trade-off on HfO2/metal cap 1T1R bipolar RRAM. *IEEE Transactions on electron devices* (2013).

[7] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie. 2016. PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*.

[8] Ahmed Elgammal, Bingchen Liu, Mohamed Elhoseiny, and Marian Mazzone. 2017. CAN: Creative Adversarial Networks, Generating "Art" by Learning About Styles and Deviating from Style Norms. *arXiv e-prints* (2017), arXiv:1706.07068.

[9] M. Everingham, S. M. A. Eslami, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. 2015. The Pascal Visual Object Classes Challenge: A Retrospective. *International Journal of Computer Vision* (2015).

[10] Zichen Fan, Ziru Li, Bing Li, Yiran Chen, and Helen Hai Li. 2019. RED: A ReRAM-based Deconvolution Accelerator. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*.

[11] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative Adversarial Nets. In *Advances in Neural Information Processing Systems 27*. Curran Associates, Inc., 2672–2680.

[12] Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron Courville. 2017. Improved Training of Wasserstein GANs. *arXiv e-prints* (2017), arXiv:1704.00028.

[13] Taeksoo Kim, Moonsu Cha, Hyunsoo Kim, Jung Kwon Lee, and Jiwon Kim. 2017. Learning to Discover Cross-Domain Relations with Generative Adversarial Networks. In *Proceedings of the 34th International Conference on Machine Learning (Proceedings of Machine Learning Research)*.

[14] A. Krizhevsky and G. Hinton. 2009. Learning multiple layers of features from tiny images. In *Technical report, University of Toronto*.

[15] Y. LeCun. 1998. In *Proceedings of the 56th annual design automation conference*. http://yann.lecun.com/exdb/mnist/

[16] J. J. Lim, H. Pirsiavash, and A. Torralba. 2013. Parsing IKEA Objects: Fine Pose Estimation. In *2013 IEEE International Conference on Computer Vision*.

[17] C. Liu, B. Yan, C. Yang, L. Song, Z. Li, B. Liu, Y. Chen, H. Li, and and. 2015. A spiking neuromorphic design with resistive crossbar. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*.

[18] Ziwei Liu, Ping Luo, Xiaogang Wang, and Xiaoou Tang. 2015. Deep Learning Face Attributes in the Wild. In *The IEEE International Conference on Computer Vision (ICCV)*.

[19] Dimin Niu, Yiran Chen, Cong Xu, and Yuan Xie. 2010. Impact of process variations on emerging memristor. In *Proceedings of the 47th Design Automation Conference*.

[20] Alec Radford, Luke Metz, and Soumith Chintala. 2016. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. *CoRR* abs/1511.06434 (2016).

[21] Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. 2016. Improved Techniques for Training GANs. *ArXiv e-prints* (2016), arXiv:1606.03498.

[22] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar. 2016. ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*.

[23] L. Song, X. Qian, H. Li, and Y. Chen. 2017. PipeLayer: A Pipelined ReRAM-Based Accelerator for Deep Learning. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[24] M. Song, J. Zhang, H. Chen, and T. Li. 2018. Towards Efficient Microarchitectural Design for Accelerating Unsupervised GAN-Based Deep Learning. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[25] Wei Ren Tan, Chee Seng Chan, Hernan Aguirre, and Kiyoshi Tanaka. 2017. ArtGAN: Artwork Synthesis with Conditional Categorical GANs. *ArXiv e-prints* (Feb. 2017), arXiv:1702.03410.

[26] Jiajun Wu, Chengkai Zhang, Tianfan Xue, William T Freeman, and Joshua B Tenenbaum. 2016. Learning a probabilistic latent space of object shapes via 3d generative-adversarial modeling. In *Advances in Neural Information Processing Systems*. 82–90.

[27] Dawen Xu, Kaijie Tu, Ying Wang, Cheng Liu, Bingsheng He, and Huawei Li. 2018. FCN-engine: Accelerating Deconvolutional Layers in Classic CNN Processors. In *ICCAD*.

[28] A. Yazdanbakhsh, M. Brzozowski, B. Khaleghi, S. Ghodrati, K. Samadi, N. Kim, and H. Esmaeilzadeh. 2018. FlexiGAN: An End-to-End Solution for FPGA Acceleration of Generative Adversarial Networks. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE Computer Society, Los Alamitos, CA, USA, 65–72. DOI: http://dx.doi.org/10.1109/FCCM.2018.00019

[29] Amir Yazdanbakhsh, Kambiz Samadi, Nam Sung Kim, and Hadi Esmaeilzadeh. 2018. GANAX: A Unified MIMD-SIMD Acceleration for Generative Adversarial Networks. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA '18)*. IEEE Press, Piscataway, NJ, USA, 650–661. DOI: http://dx.doi.org/10.1109/ISCA.2018.00060

[30] Raymond A. Yeh, Chen Chen, Teck Yian Lim, Alexander G. Schwing, Mark Hasegawa-Johnson, and Minh N. Do. 2016. Semantic Image Inpainting with Deep Generative Models. *arXiv e-prints* (2016), arXiv:1607.07539.

[31] Fisher Yu, Ari Seff, Yinda Zhang, Shuran Song, Thomas Funkhouser, and Jianxiong Xiao. 2015. LSUN: Construction of a Large-scale Image Dataset using Deep Learning with Humans in the Loop. *arXiv e-prints* (2015), arXiv:1506.03365.

[32] Xinyu Zhang, Srinjoy Das, Ojash Neopane, and Ken Kreutz- Delgado. 2017. A Design Methodology for Efficient Implementation of Deconvolutional Neural Networks on an FPGA. *ArXiv e-prints* (2017), arXiv:1705.02583.