ReGAN: A Pipelined ReRAM-Based Accelerator for Generative Adversarial Networks

Fan Chen, Linghao Song, Yiran Chen

Department of Electrical and Computer Engineering, Duke University, Durham, NC, USA
{fan.chen, linghao.song, yiran.chen}@duke.edu

Abstract— Generative Adversarial Networks (GANs) have recently drawn tremendous attention in many artificial intelligence (AI) applications including computer vision, speech recognition, and natural language processing. While GANs deliver state-of-the-art performance on these AI tasks, it comes at the cost of high computational complexity. Although recent progress demonstrated the promise of using ReRMA-based Process-In-Memory for acceleration of convolutional neural networks (CNNs) with low energy cost, the unique training process required by GANs makes them difficult to run on existing neural network acceleration platforms: two competing networks are simultaneously co-trained in GANs, and hence, significantly increasing the need of memory and computation resources. In this work, we propose ReGAN – a novel ReRAM-based Process-In-Memory accelerator that can efficiently reduce off-chip memory accesses. Moreover, ReGAN greatly increases system throughput by pipelining the layer-wise computation. Two techniques, namely, *Spatial Parallelism* and *Computation Sharing* are particularly proposed to further enhance training efficiency of GANs. Our experimental results show that ReGAN can achieve $240\times$ performance speedup compared to GPU platform averagely, with an average energy saving of $94\times$.

I. Introduction

Generative Adversarial Networks (GANs) have recently been extensively deployed in various image processing applications [1, 2, 3, 4] thanks to their superior performance compared to previous state-of-the-art approaches. For instance, Yeh *et al.* proposed using GAN to predict missing content in an image based on the surrounding values and consequently, to recover corrupted images [5]. CAN [6] presents a creative GAN for generating novel art that fits within established styles. The results demonstrate that GAN has an ability to generate high-resolution images that are visually appealing and significantly sharp. Most importantly, GAN and its variations can be used for unsupervised and semi-supervised learning to reduce large volume of annotations and labels that commonly required by supervised deep leaning algorithms.

In a GAN framework, two adversaries – a discriminator and a generator, are trained in parallel. Both discriminator and generator typically are modeled as Deep Neural Networks (DNNs). In recent years, we have seen a continuously increasing trend of deploying large and deep networks in GAN (e.g. 101-layer ResNet generator and discriminator [4]). These models are often trained with large data sets and therefore, incurring large consumption of computation and storage resources. However, conventional CPU and GPU platforms are not able to accommodate this intensive training process due to the well-known *power wall* [8] and *memory wall* [9] phenomena.

Emerging non-volatile memory-metal-oxide resistive random access memory (ReRAM), has become a promising candidate for hardware accelerator implementations of Neural Networks (NNs) due to its high density, fast access, and low leakage power. In addtion, ReRAM-based Process-In-Memory (PIM) accelerator is particularly an appealing option to alleviate the *memory wall* problem because ReRAM provides both computation and storage capabilities. Recent works PRIME [12] and ISAAC [13] demonstrate how to utilize ReRAM to perform neural computations in memory. PipeLayer [14] is presented to support both trainings and testings of Convolution Neural Networks (CNNs).

Unfortunately, it is still difficult to efficiently accelerate GANs by adopting the existing schemes. First, two networks are co-trained in GAN so that the training is more sophisticated and computationally intensive. In a GAN framework, the structure of its generator and discriminator are nearly symmetric and trained alternatively. Normally, the generator is trained to produce fake samples, the discriminator, however, is trained with both real samples and fake samples created by the generator. Hence, a direct adoption of PipeLayer [14] to implement GAN takes $3\times$ latency and hardware overhead compared to a CNN counterpart. Second, existing schemes all focus on accelerating discriminative CNN [12, 13], but the generator core is an up-sampling Fractional-strided CNN, which is not well studied in prior arts.

In this work, we propose ReGAN, a ReRAM-based PIM accelerator for GAN training. Our contributions include:

- First, we analyze the general training procedure in GAN. Then we present a pipelined architecture to improve the system throughput by utilizing the structural layer-wise computation. This architecture directly leverages ReRAM cells to perform computation without the need for extra processing units. ReRAM memory are used as buffers to store intermediate results. Such a design greatly reduces data movements and energy consumption by preventing data from being transferred across memory hierarchy.
- We propose *Spatial Parallelism* and *Computation Sharing* that explore the parallelism and data dependency in GAN to support training more efficiently.
- We evaluate ReGAN and compare it against the state-of-the-art GPU platform. Our experimental results show that it improves the performance and energy efficiency by $240\times$ and $94\times$, respectively.

This paper is organized as follows: Section II introduces the background of Generative Adversarial Networks, ReRAM, and its application in neural computations. Section III elaborates

978-1-5090-0602-1/18/$31.00 ©2018 IEEE

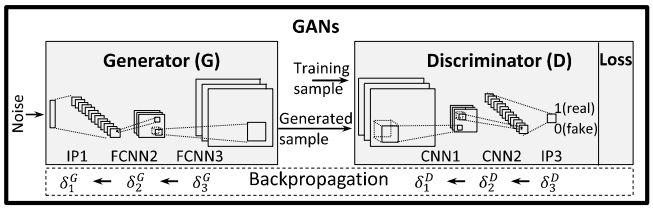

Fig. 1. GAN system.

ReGAN architecture and optimization. Section IV introduces the implementation of ReGAN. Section V presents the evlauation methodology and experimental results. Section VI concludes this paper.

II. Background

A. Basics of Generative Adversarial Networks

A generative adversarial network has two subnetworks, a Generator (G) and a Discriminator (D) as illustrated in Fig. 1. Both D and G typically are modeled as deep neural networks. In general, the discriminator is a binary classifier which is trained by distinguishing real samples from generated ones, and the generator is optimized to produce samples that can fool the discriminator.

In the context of image processing, most of the GAN variations are (or at least partially) based on the Deep Convolutional Generative Adversarial Networks (DCGAN) [2, 6, 5, 7]. Thus we focus on DCGAN in this work. In DCGAN, D uses discriminative CNN to "downsample" the input to produce classification [11]. In contrast, G takes a uniform noise distribution as input, which is then projected to a small spatial extent convolutional representation with many feature maps and used as the start of a series of Fractional-Strided Convolution layers (FCNN). The results are then converted into high-dimension images.

The training process of GAN consists of two phases:

1) **Propagation.** An input example propagates through a series of neural networks in order to generate the network's output value(s). The function of NNs can be formulated as:

$$\boldsymbol{y} = \boldsymbol{W} \cdot \boldsymbol{x} + \boldsymbol{b} \text{ and } \boldsymbol{z} = h(\boldsymbol{y}), \tag{1}$$

where the output neuron vector $\boldsymbol{z}$ is determined by the input neuron vector $\boldsymbol{x}$, the weight matrix of connections $\boldsymbol{W}$ and the bias vector $\boldsymbol{b}$. Usually, $h(\cdot)$ is a non-linear activation function. A *cost function* J is defined to quantitatively evaluate the difference between the network output and its expected output. The error for layer l is defined as $\delta_l = \frac{\partial J}{\partial \boldsymbol{b}_l}$ and propagates backwards to the previous layer as:

$$\boldsymbol{\delta_{l-1}} = \boldsymbol{W_l}^T \cdot \boldsymbol{\delta_l} \cdot \boldsymbol{h}^{'}(\boldsymbol{y_l}), \tag{2}$$

2) **Update.** The update of each weight (W_{ji}) and bias (b_{ji}) is:

$$W_{ji,l} \leftarrow W_{ji,l} - \eta \cdot z_{i,l-1} \cdot \delta_{j,l}, \tag{3}$$

$$b_{ji,l} \leftarrow b_{ji,l} - \eta \cdot \delta_{j,l}, \tag{4}$$

where η is the learning rate and $\delta_{j,l}$ is the error back propagated from the node j in layer l. $z_{i,l-1}$ is the input of the node i in layer $l-1$.

Note that in practice, training examples are placed in batches, and the error is averaged at the end of each batch, which is then used to update the weights.

```
for(row = 0; row = O_W, row++){
  for(col = 0; row = O_H, col++){
    for(to = 0; to = O_C, to++){
      for(ti = 0; ti = I_C, ti++){
        for(i = 0; i = K, i++){
          for(j = 0; j = K, j++){
            Output_fm[to][row][col] +=
                kernel[to][ti][i][j] * input_fm[ti][S*row + i][s*col+j]
} } } } } }
```

Fig. 2. Convolution in CPU.

B. Convolution Arithmetic

The core component of a discriminator is convolutional neural networks. **CNN** has three types of layers: convolution layer, batch normalization layer (BN) and activation layer (e.g. *sigmoid, rectified linear unit (ReLU), Tanh,* etc.). The generator consists of a series of fractional-strided convolutions networks. **FCNN** is also composed of three types of layers: fractional-strided convolution layer, batch normalization layer and activation layer. Different from the traditional convolution operations, FCNN inserts many zeros into its input feature maps and then performs the up-sampling convolutional computation. The output feature maps of FCNN are larger than its input maps.

The pseudo code of a convolution layer implemented in a CPU is shown in Fig. 2. Fractionally-strided convolutions work by swapping the forward and backward passes of a convolution [10]. As we will show in Section III.B, it is always possible to implement fractionally-strided convolutions with a direct convolution. The disadvantage is that it usually requires adding several columns and rows of zeros to the input, resulting in a less efficient implementation.

Activation layer is a nonlinear function which serves as a threshold. Batch normalization layer is another important component. It is applied before activation to stabilize the training process of GANs. **All variable names** in Fig. 1 and Fig. 2 will be used and consistent in following sections.

C. Neuromorphic Computing with ReRAM

Resistive random access memory is a type of non-volatile memory that stores information by changing cell resistances. Recent works [12, 13, 14] demonstrated that ReRAM is a promising candidate to realize area efficient matrix-vector multiplication for neuromorphic computation in a crossbar architecture.

Fig. 3 (a), (b) shows an example of mapping matrix-vector

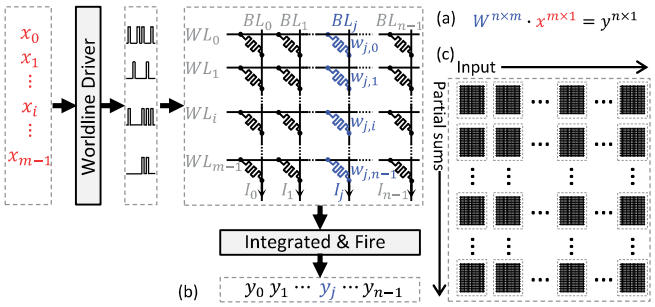

Fig. 3. Mapping matrix-vector multilication (a) to ReRAM crossbar array (b). (c) Mapping large matrix to multiple ReRAM arrays.

multiplication to a ReRAM crossbar. The vector is represented by the input signals on the wordlines (WL). Each element of the matrix is programmed into the cell conductance in the crossbar array. Thus, the current flowing to the end of each bitline (BL) is viewed as the result of the matrix-vector multiplication. For a large matrix that can not fit in a single array, the input and the output shall be partitioned and grouped into multiple arrays as shown in Fig. 3 (c). The output of each array is a partial sum, which is collected horizontally and summed vertically to generate the actual results.

Note that matrices with positive and negative elements are implemented as two separate crossbar arrays and share the same input ports. We adopt the same spike-based scheme in [15], in which the *worldline driver* converts the input to a sequence of weighted spikes and the *integrate-and-fire* circuit integrates and converts the output currents into digital values. Due to space limit, readers could refer to [15] for more details.

III. ReGAN Architecture

In this section, we first analyze the training procedure of ReGAN. Then we describe the method of mapping ReGAN to ReRAM crossbars. Finally, we present the idea of pipelined GAN training and two schemes to further improve the computational efficiency.

A. Training Procedure

For simplicity, Fig. 4 demonstrates the configuration of ReGAN to process the training of the networks shown in Fig. 1. The discriminator(D) and the generator(G) are trained alternatively until they reache an equilibrium.

Train D. 1) ❶~❷ shows the dataflow of training D on the training samples. A training sample is fed into D at T0. This sample flows through layers of D consecutively in forward direction. A loss function is then computed at T4 based on accurate labels ('1' for training sample). Finally, the error and partial derivatives propagate all the way back to the first layer of D and are stored. This process takes 7 logical cycles. 2) ❸~❹ depicts the dataflow of training D on generated samples. In this case, G is concatenated with D to form a large network. G maps a random vector to a sample which has the same dimension with real samples. This sample follows the layers in D and a loss function is performed with the label ('0' for generated sample). Similarly, the partial derivatives propagate back to the first layer of D at T10. Therefore, in T11, the derivatives stored in 1) and 2) are summed accordingly and then used to update the weights of D. It take $7+10+1=18$ cycles in total. During this process, G is used but not updated.

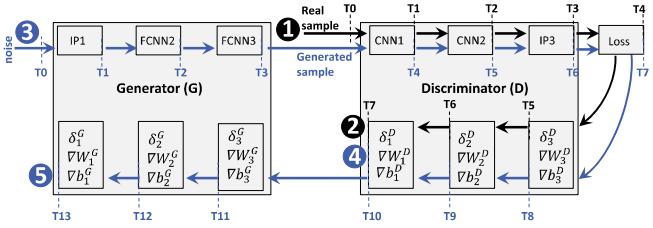

Fig. 4. Training loops in GAN.

Train G. ❸~❺ shows the data flow of training G. It is similar to ❸~❹ except: 1) The error is computed with inaccurate labels ('1' for generated sample) in T7, since the goal of G is to fool D; 2) The error propagates all the way back to the first layer of G; 3) The weights of G have been updated in T14 while D is fixed.

B. Mapping ReGAN to ReRAM

Fig. 5(a) visualizes how a typical convolution layer exhibited in Fig. 2 works. The three steps required to implement the convolution layer are: 1) element-wisely multiplying a $K \times K$ kernel by the input pixel window starting at $i_h \times S, i_w \times S$ across all the input channels; 2) adding the results in step 1) to a local area in the output feature map; 3) sliding the input window by stride S, repeat 1) and 2) for all the possible locations in the input feature maps.

In general, we can decompose the convolution between the input and the kernel to multiplication and addition. Each element in the output feature maps is obtained by multiplying a region in the input feature maps with the kernel. Therefore, we reorder each kernel as a vector with the size of $K \times K \times I_C$. Accordingly, the input feature map is converted to a matrix with $K \times K \times I_C$ rows and $O_W \times O_H$ columns. Thus, the output feature map can be computed as:

$$\boldsymbol{W}^{n\times m} \cdot \boldsymbol{x}^{m\times 1} = \boldsymbol{y}^{n\times 1} \tag{5}$$

where $n = O_W * O_H, m = K \times K \times I_C$. Naturally, this matrix-vector multiplication can be easily mapped to one or multiple ReRAM arrays.

Given the parameters in Fig. 5(a), Fig. 5(b) shows an example about how error backpropagates backwards in CNN: 1) zero padding the error map for $P = 2$; 2) rotating the kernel used in the forward path for 180°; 3) computing the error backpropagation for CNN as the convolution between the modified error maps and kernel.

Different from the CNN downsampling, the generator employs FCNN to project the input feature maps to a higher-dimensional space. Mathematically we can implement a FCNN with a direct CNN [10] as illustrated in Fig. 6(a): 1) adding zeros between each input in the feature maps with zero padding;

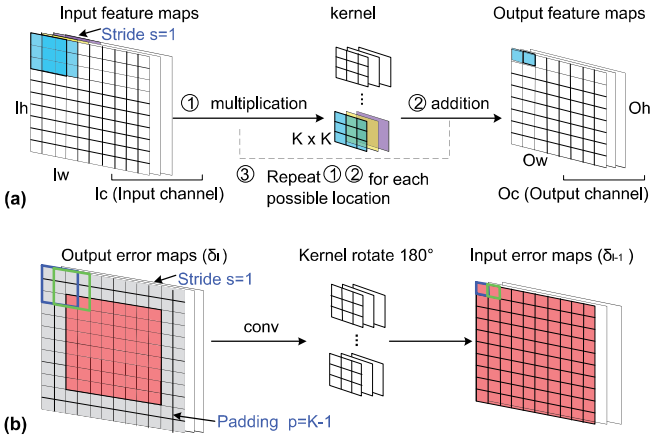

Fig. 5. Visualization of a single convolution layer (a) Data forward, (b) Error backpropagation.

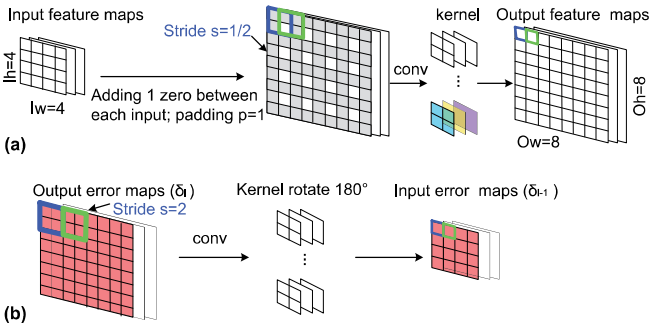

Fig. 6. Visualization of a single fractional-strided convolution layer (a) Data forward, (b) Error backpropagation.

2) computing the convolution between the extended input feature maps and the kernel. Fig. 6(b) describes how the error propagates backwards in FCNN, which is performed as a typical convolution with strides.

Every convolution boils down to an efficient implementation of a matrix-vector multiplication, making the mapping to ReRAM crossbar much simpler. In practice, FCNN involves adding many columns and rows of zeros to the input, resulting in a structurally sparse input matrix and hence, less efficient implementation. However, simply change the data layout often produces non-structured memory access that adversely impacts practical acceleration in hardware platforms. Optimizing ReGAN to further improve computation efficiency requires non-trivial effort and is left for future work.

C. Pipeline Design

In training, the input data are normally processed in batch of size B (e.g. 64). It means inputs within the same batch are all processed based on the same parameters. The backpropagated error due to each input are stored and only applied at the end of each batch. Therefore, no dependency exists among data inputs inside a batch. We propose an architecture to increase the system throughput by pipelining the training process.

In the execution, a new input can enter the pipeline every cycle within a batch. At the end of a batch, a new input belonging to the next batch cannot enter the pipeline until all inputs in previous batch are processed and weights are updated.

TABLE I
LATENCY OF PIPELINE

Loop	Forward	Backward	Update
❶~❷	L_D	L_D+1	0
❸~❹	L_G+L_D	L_D+1	1
❸~❺	L_G+L_D	L_G+L_D+1	1

Assuming the discriminator has L_D layers, the generator has L_G layers, the batch size is B. The pipelined execution cycles of the three training phases are shown in Table. I. To update D, we first train D on real samples, which takes $L_D+L_D+1+B-1$ cycles (a new batch has to wait $B-1$ cycles for the previous batch to drain from the pipeline); then $L_G+L_D+L_D+1+B-1$ cycles to train D on generated samples; finally, it take one cycle to update D. Similarly, it takes $2L_G+2L_D+B+1$ cycles to train G. Without the pipeline, the training of D and G consume $(4L_D+L_G+2)B$ cycles and $(2L_G+2L_D+1)B$ cycles, respectively.

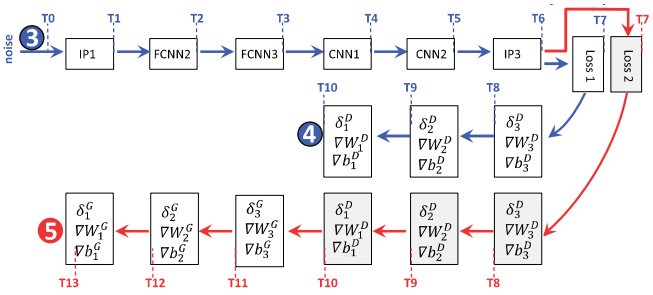

Fig. 7. Improved pipeline by computation sharing.

D. Computation Optimization

We propose *spatial parallelism* and *computation sharing* to further optimize the pipeline performance.

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

Computation Sharing (CS). Fig. 7 highlights the difference between training phases ❸~❹ and ❸~❺. They 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.

Note that spatial parallelism and Computation Sharing are two orthogonal methods which can be combined to improve the performance.

IV. IMPLEMENTATION OF REGAN

A. Overall Architecture

we propose ReGAN, a novel ReRAM-based Process-In-Memory accelerator, which leverages ReRAM cells to perform computation for GANs acceleration without the need of extra processing units. Fig. 8 shows the overview of ReGAN. Similar to [12], ReGAN partitions the ReRAM main memory into three regions: memory (Mem) subarrays, full function (FF) subarrays, and Buffer subarrays. Mem subarrays are the same as conventional memory subarrays and have data storage capability. FF subarrays can be configured in both computation and storage modes. In computation mode, FF subarrays execute matrix-vector multiplications; in memory mode, they are used as Mem subarrays for data storage. We reserve Mem subarrays that are closest to the FF subarrays as Buffer subarrays, which are used to store the intermediate results between layers (e.g. generated images, data required for compute partial derivatives, etc.). They are connected to FF subarrays through private data ports, so that buffer accesses do not consume the bandwidth of Mem subarrays.

B. FF Subarray Design

The design goal for FF subarrays is to support both storage and computation with a minimum area and energy overhead.

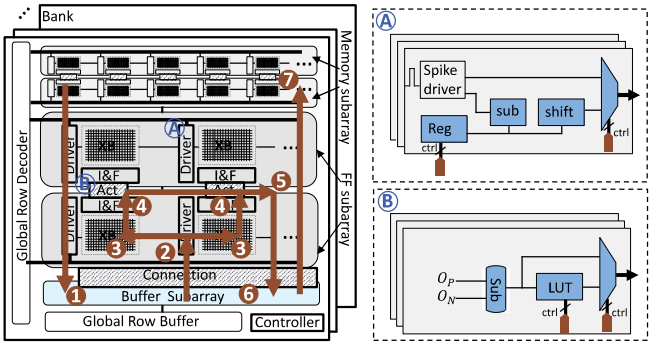

Fig. 8. The ReGAN architecture.

To achieve this goal, we replace the power-hungry ADC and DAC in [12] with the spike-based scheme as [15]. In this section we demonstrate how ReGAN supports **batch normalization, activation function**, and **fully connected layers** as follows.

Batch normalization (BN) greatly improves the training stability in the original DCGAN [2]. However, it causes the output of a neural network for an input example to be highly dependent on several other inputs in the same batch. Therefore, VBN, *virtual batch normalization* is introduced in [7]. In VBN, each example is normalized based on the statistics collected on a reference batch. The reference batch are chosen once and fixed at the start of training. Moreover, results in the recent works [3, 4] shows that BNs are not as necessary as previously claimed. In this work, we use the state-of-the-art GAN [3, 4] without BN to compose our benchmark suit. However we still add several components in Wordline drivers marked as light blue in Fig. 8Ⓐ to support VBN. We deploy the *sub* and *shift* to do the minus and division in BN layers. Note that the divisor should be 2^n in this case. When complex BN layers are applied, ReGAN requires the help of CPU for BN computation.

Fig. 8Ⓑ depicts the design of activation function. Computation results from both the positive subarray and negative subarray are first merged by the subtractor, and then sent to the configurable Look Up Table (LUT) to realize the activation function. The LUT can be bypassed in certain scenarios, e.g. when a large NN is mapped to multiple crossbar arrays.

DCGAN are fully convolution layers except the last layer in D and the first layer in G. However, the first layer of G is only matrix multiplication, and hence can be easily mapped to ReRAM crossbars. While the last layer of D just flattened the output of previous CNN layers, therefore, it does not require extra computation. Fully connected layers in CNNs can be converted to convolutional layers.

C. Pipeline Implementation

Fig. 8 highlights the pipelined design of ReGAN to perform the training of GAN. ❶ When the input are latched into the buffer subarry, the FF subarrys start the computation. When the first input vector is sent to the ReRAM crossbar, ❷ the wordline driver can continue to process the next input vector. ❸ The input are multiplied element-wisely with the two crossbar with positive and negative weights, respectively. ❹ *Integrated-and-Fire* circuits collect a current on the bitline and produce output pulse representing results. ❺ The activation function units perform non-linear functions on results. ❻ The intermediate data (e.g. partial sum, derivatives, etc.) are stored in the buffer subarray. ❼ The results are write back to the memory subarry when the compuation of one layer is finished or when the buffer subarray is full.

TABLE II
BENCHMARKS (C: CONVOLUTION/FRACTION-STRIDE CONVOLUTION LAYER; F: FULLY CONNECTED LAYER.)

Name	Database	G Topology	D Topology
MLP1A	mnist	128-dim 2-layer	2-layer MLP
MLP1B	mnist	128-dim 2-layer	3C1F
MLP2A	mnist	512-dim 4-layer	4-layer MLP
MLP2B	mnist	128-dim 2-layer	3C1F
GAN1	cifar-10	4C1F	4C1F
GAN2	celebA/LSUN	5C1F	5C1F
ResnetGAN	cifar-10	[4]	[4]

V. EXPERIMENTS

Workload. We ran experiments on image generations. Since the image size is critical to the topology of the generator, we selected a variety of dataset including: MNIST [16], cifar-10 [17], celebA [18] and LSUN [19].

MNIST contains 60K gray written digits labeled from 0 to 9 with a size of 28×28 pixels. CIFAR-10 is 32×32 color image dataset. LSUN-Bedrooms dataset is a collection of natural images of indoor bedrooms. The generated samples are 3-channel images of 64×64 pixels in size. CelebA includes 200K celebrity images with large pose variations and background clutter. We scaled and center-cropped CelebA images into 108×108 pixels, then re-size them into 64×64 pixels.

For MNIST, we built four GANs, each of which has a MultiLayer Perceptron (MLP) generator and an MLP/CNN discriminator. For CIFAR-10 and SVHN, we built a symmetric GAN with four CNN/FCNN layers in G/D. In particular, we also selected the 101-layer ResNet GAN in [4] as benchmarks on CIFAR-10. For CelebA and LSUN, we built a five-layer CNN/FNN GAN as benchmark. Table II details the networks and dataset.

Experimental Setup. We compare ReGAN against the a GPU-based platform. On the GPU platform, the training of GAN are based on the widely used framework Tensorflow [20]. The parameters of our GPU platform are shown in Table III. We built a ReRAM simulator based on NVSim [21] to evaluate ReGAN. We adopted the read/write latency, read/write energy from [14] as 29.31 ns/50.88 ns per spike, 1.08 pJ/3.91 nJ per spike. The area model are based on [22].

Performance Results. The performance comparison is shown in Fig. 9. For each application, we report the per-

TABLE III
CONFIGURATION OF THE GPU PLATFORM.

Memory	128 GB
Storage	1 TB4
Graphic Card	NVIDIA Geforce GTX 1080
Architecture	Pascal
CUDA Cores	2560
Base Clock	1607 MHz
Compute Capability	6.1
Graphic Memory	8 GB GDDR5X
Memory Bandwidth	320 GB/s
CUDA Version	8

formance comparison of pipelined ReGAN (*pipe*), pipelined ReGAN with *Spatial Parallerism* (*pipe+sp*), pipelined ReGAN with *Computation Sharing* (*pipe+cs*) and pipelined ReGAN with both *Spatial Parallerism* and *Computation Sharing* (*pipe+sp+cs*). The GPU platform is used as our baseline and all performance and energy results on different settings are normalized to it.

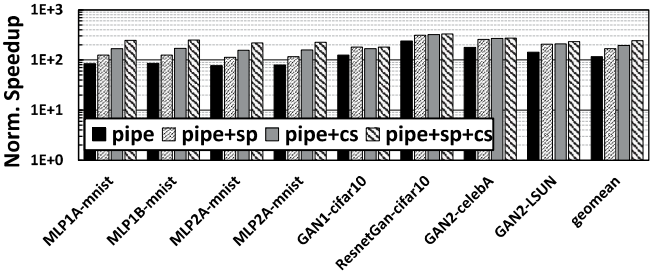

Fig. 9. Normalized speedup.

For simple MLP generators, *cs* achieves better improvement than *sp*. This is because *cs* hides the long latency training phase ❸~❹ since we train D and G alternatively. For complex generators, e.g. *GAN1, GAN2, ResnetGAN*, *sp* and *cs* achieve marginal improvement since G is updated several times before D is updated to avoid the case that D quickly maintaines near its optimal solution and G changes slowly.

On average, ReGAN achieves $116\times$ speedup. *sp* and *cs* can further improve the the performace to $240\times$.

Energy Results. Fig. 10 shows the energy consumption of all applications. Although, *sp* and *cs* boost the performance of simple MLP generators, it also increases power consumption for the extra FF subarrays and Buffer subarrays. Thus, they have similar energy consumption in different settings.

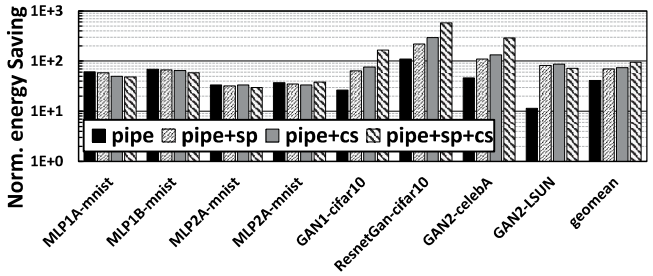

Fig. 10. Normalized energy saving.

GAN2 trained celebA and LSUN show different energy consumption. This is due to trainings on generating faces (celebA) are apparently easier than generating bedroom images (LSUN). GAN converges faster on celebA dataset and therefore results in higher energy saving. We can see from the results that ReGAN provides significant energy savings for deep networks. The highest energy saving are $578\times$ (*R*esnetGan on cifar10).

VI. Conclusions

In this paper, we analyze the training procedure in GAN and present a ReRAM-based PIM, ReGAN, to accelerate GAN training. Two orthogonal methods – *Spatial Parallelism* and *Computation Sharing* are proposed to further improve efficiency. Compared to state-of-the-art GPU platform, on average ReGAN improves performance by $240\times$ and achieves $94\times$ energy saving.

Acknowledgment

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

References

[1] Ian Goodfellow, *et al.*, "Generative adversarial nets," In *Advances in neural information processing systems (NIPS)*, 2014.

[2] Alec Radford, *et al.*, "Unsupervised representation learning with deep convolutional generative adversarial networks," In *International Conference on Learning Representation (ICLR)*, 2016.

[3] Martin Arjovsky, *et al.*, "Wasserstein gan," In *arXiv preprint*, arXiv:1701.07875, 2017.

[4] Ishaan Gulrajani, *et al.*, "Improved training of wasserstein gans," In *arXiv preprint*, arXiv:1704.00028, 2017.

[5] Raymond Yeh, *et al.*, "Semantic image inpainting with perceptual and contextual losses," In *arXiv preprint*, arXiv:1607.07539, 2016.

[6] Ahmed Elgammal, *et al.*, "CAN: Creative Adversarial Networks, Generating "Art" by Learning About Styles and Deviating from Style and Deviating from Style Norms" In *arXiv preprint*, arXiv:1706.07068, 2017.

[7] Tim Salimans, *et al.*, "Improved techniques for training gans," In *Advances in Neural Information Processing Systems (NIPS)*, 2016.

[8] X. Guo, *et al.*, "Resistive computation: avoiding the power wall with low-leakage, STT-MRAM based computing," In *ACM SIGARCH Computer Architecture News*, pp. 371-382, 2010.

[9] W.A. Wulf, *et al.*, "Hitting the memory wall: implications of the obvious," In *ACM SIGARCH computer architecture news*, vol. 23, no. 1, pp. 20-24, 1995.

[10] Vincent Dumoulin, *et al.*, "A guide to convolution arithmetic for deep learning," In *arXiv preprint*, arXiv:1603.07285, 2016.

[11] J. Schmidhuber, *et al.*, "Deep learning in neural networks: An overview," In *Neural networks*, vol. 61, pp. 85117, 2015.

[12] Ping Chi, *et al.*, "Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory," In *Proceedings of the 43rd International Symposium on Computer Architecture(ISCA)*, 2016.

[13] Ali Shafiee, *et al.*, "ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," In *Proceedings of the 43rd International Symposium on Computer Architecture(ISCA)*, 2016.

[14] Linghao Song, *et al.*, "PipeLayer: A pipelined ReRAM-based accelerator for deep learning," In *High Performance Computer Architecture (HPCA)*, 2017.

[15] Chenchen Liu, *et al.*, "A spiking neuromorphic design with resistive crossbar," In *Design Automation Conference (DAC)*, 2015.

[16] Y. LeCun, *et al.*, "The MNIST database of handwritten digits," 1998.

[17] Alex Krizhevsky, *et al.*, "Learning multiple layers of features from tiny images," In *Technical report, University of Toronto*, 2009.

[18] Ziwei Liu, *et al.*, "Deep Learning Face Attributes in the Wild," In *Proceedings of International Conference on Computer Vision (ICCV)*, 2015.

[19] Fisher Yu, *et al.*, "Lsun: Construction of a large-scale image dataset using deep learning with humans in the loop," In *arXiv preprint*, arXiv:1506.03365 2015.

[20] M Abadi, *et al.*, "TensorFlow: Large-scale machine learning on het-erogeneous systems," 2015. Software available from https://www.tensorflow.org/.

[21] Xiangyu Dong, *et al.*, "Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory," In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 31.7 (2012): 994-1007.

[22] R. Fackenthal, *et al.*, "A 16Gb ReRAM with 200MB/s write and 1GB/s read in 27nm technology," In *ISSCC Tech*, 2014.