PIM-TGAN: A Processing-in-Memory Accelerator for Ternary Generative Adversarial Networks

Adnan Siraj Rakin, Shaahin Angizi, Zhezhi He, and Deliang Fan
Department of Electrical and Computer Engineering
University of Central Florida, Orlando, FL 32816
{adnanrakin, angizi, elliot.he}@knights.ucf.edu, dfan@ucf.edu

***Abstract*—Generative Adversarial Network (GAN) has emerged as one of the most promising semi-supervised learning methods where two neural nets train themselves in a competitive environment. In this paper, as far as we know, we are the first to present a statistically trained Ternarized Generative Adversarial Network (TGAN) with fully ternarized weights (i.e. -1,0,+1) to massively reduce the need for computation and storage resources in the conventional GAN structures. In the proposed TGAN, the computationally expensive convolution operations (i.e. Multiplication and Accumulation) in both generator and discriminator's forward path are converted into hardware-friendly Addition/Subtraction operations. Accordingly, we propose a Processing-in-Memory accelerator for TGAN called (PIM-TGAN) based on Spin-Orbit Torque Magnetic Random Access Memory (SOT-MRAM) computational sub-arrays to efficiently accelerate the training process of GAN within non-volatile memory. In addition, we propose a parallelism technique to further enhance the training efficiency of TGAN. Our device-to-architecture co-simulation results show that, with almost the same inception score to the baseline GAN with floating point number weights on different data-sets, the proposed PIM-TGAN can obtain ∼25.6× better energy-efficiency and 22× speedup compared to GPU platform averagely, and, 9.2× better energy-efficiency and 5.4× speedup over the best processing-in-ReRAM accelerators.**

I. Introduction

Generative Adversarial Networks (GANs) have recently shown the considerable success in a variety of image processing problem such as image generation [1], super resolution generation [2], text2image [3] and etc [4], [5]. GANs succeed through the idea of adversarial training as depicted in Fig. 1, where two adversaries, a generator and a discriminator are co-trained simultaneously. The generator strives to create fake samples from the same distribution as the real data. However, discriminator tries to distinguish the fake samples from the real ones. The adversaries are nearly symmetric and modeled as Deep Neural Networks (DNNs). Thus, the training of GAN is very sophisticated and computationally-intensive [1], [6].

To eliminate the need for massive MAC operations and memory usage in DNN deployments, researchers have come up with various quantized/binarized DNNs by constraining inputs, weights or gradients to be quantized/ binarized specifically in forward propagation [7], [8]. For example, DoReFa-Net shows acceptable accuracy over SVHN and ImageNet data-sets under different low bit-width configurations after applying its quantization method [7]. Even recent extremely-quantized Binary Neural Networks (BNNs) have successfully shown close accuracy to state-of-the-art DNNs [8]. A similar yet modified version of binary training has been adopted for GAN as well [9] to lower the memory utilization, thus significantly improve the hardware deployment .

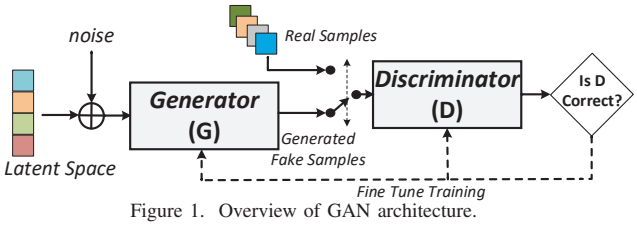

Figure 1. Overview of GAN architecture.

In the domain of DNN accelerator architecture, the isolated memory and processing units (i.e., GPU or CPU) are interconnected via buses which has encountered serious challenges, including long memory access latency, significant congestion at I/Os, limited memory bandwidth, huge data communication energy and large leakage power consumption for storing network parameters in the volatile memory [10]. To address the aforementioned concerns, Processing-in-Memory (PIM) architecture as a potentially viable way to address memory wall challenge, have been widely explored in recent accelerators design [10], [11], [12]. The key idea of PIM is to embed logic units within memory to process data by leveraging the inherent parallel computing mechanism and exploiting large internal memory bandwidth. It could lead to remarkable saving in the off-chip data communication energy and latency. The proposals for exploiting SRAM-based PIM architectures can be found in recent literature [13]. While, PIM in context of DRAM [14] provides more benefits in recent years owing to the larger memory capacity and off-chip data communication reduction as opposed to SRAM-based PIM. However, the existing DRAM-based PIM architecture [14] encounters several inevitable drawbacks, such as high refresh/leakage power, multi-cycle logic operations, operand data overwritten, operand locality, etc.

For bringing up the countermeasure of the aforementioned problems, massive research efforts have been investigated in emerging Non-Volatile Memory (NVM) technology, such as Phase Change Memory (PCM) [15], resistive RAM (ReRAM) [10], Magnetic RAM (MRAM) and etc. Since the PIM architecture requires great amount of writing operations, MRAM

This work is supported in part by the National Science Foundation under Grant No. 1740126 and Semiconductor Research Corporation nCORE.

2576-6996/18/$31.00 ©2018 IEEE
DOI 10.1109/ICCD.2018.00048

is considered as the most promising PIM candidate while PCM and ReRAM shows high latency and power dissipation in memory write [15]. Through leveraging the resistive property of MRAM, the bulk bit-wise logical operations (e.g., AND/OR/XOR) can be perform through the technique of routed sensing [16], which open a new way to realize efficient PIM paradigms.

In this paper, we intertwine the novelties from both algorithm and hardware architecture perspectives to efficiently accelerate GAN's training. First, we propose a new weight ternarization method to massively reduce the need for computation and storage resources in the forward inference paths with minimal or no performance loss compared to the model with floating-point weights. Second, we develop a PIM accelerator for TGAN called (PIM-TGAN) based on SOT-MRAM computational sub-arrays to efficiently accelerate TGAN training within non-volatile memory. Third, we present a parallelism technique to further enhance the training efficiency of TGAN in hardware level.

II. Ternary GAN Training Method

In this section, we first provide an overview of the generative adversarial training architectures and algorithms that we are using later to evaluate our TGAN. Then we propose our TGAN training method which adapts the traditional GAN training to make it more suitable for efficient hardware implementation.

A. GAN training

Deep convolutional GAN (DC-GAN) [1], Wasserstein GAN (WGAN) [17] and Wasserstein GAN with gradient clipping (WGAN-GP) [18] are some of the most recent advancement in GAN training algorithms. DC-GAN architecture consists of two separate models. A discriminator model (D) that estimates the probability of a given sample being real or fake. It is trained as a critic to differentiate between fake samples and real ones. Whereas, the other model, popularly known as the generator (G), samples a random/uniform noise input (z). It is trained to trick the discriminator. In other words, G captures the real data distribution so that the generated images can be as real as possible. Basically, this is a zero-sum game between the two models which motivates each of them for improving performance as shown in Fig. 1. Therefore, both models are non-cooperative and trained simultaneously to find a Nash equilibrium [19]. This raises several issues, such as no guarantee for convergence. One of the improvement was to change the loss function and measuring distance of GAN through introducing Wasserstein GAN [17]. WGAN algorithm uses Wasserstein Distance as a quantitative method to measure the distance between two probability distributions. Later, in the work of WGAN-CP [18], WGAN training was improved even further as they proposed to restrict the trained weights of the discriminator within a certain range. Beyond the weight clipping in WGAN-CP, another variant WGAN algorithm (WGAN-GP) would be to use gradient penalty [18]. For each of the aforementioned GANs, we use the same architecture containing four convolution and deconvolution layers for discriminator and generator, respectively. In what follows, We will evaluate the proposed TGAN using DC-GAN, WGAN-CP and WGAN-GP to demonstrate its effectiveness against a variety of bitwidth-reduced GAN training algorithms.

B. TGAN training

GAN requires a lot of computational power for training in consideration of the fact that two separate models are trained parallely. In order to improve the training efficiency and enable efficient hardware mapping, we introduce a novel training algorithm for GAN. We train both the generator and discriminator model using tenarized weights (-1,0,+1). It can then eliminate the needs for computationally-expensive multiplication units. So all the convolution operations can be implemented using addition and subtraction. We later show that such conversion not only can provide a comparable Inception Score (IS) to the full-precision GANs in software level, but also can reduce memory access and energy consumption of convolutional layers while accelerating the forward path in the hardware implementation.

We train both generator and discriminator from scratch using ternarized weights. The training process can be summarized into three steps, which are operating sequentially in an iterative manner: ①-statistical weight scaling and weight ternarization, ②-ternary weight-based inference for loss function computation and ③-back propagation to update full precision weights. ① will first ternarize current full precision weights and compute the corresponding scaling factor based on current statistical distribution of full precision weight. For weight ternarization (i.e. -1, 0, +1), we adopt the variant staircase ternarization function, which compares the full precision weight with the symmetric threshold $\pm\Delta_{th}$ [20]. Such weight ternarization function in the forward path for inference can be described as:

$$\boldsymbol{w}_l' = \alpha \times Tern(\boldsymbol{w}_l) = \begin{cases} \alpha \times Sign(w_{l,i}) & |w_{l,i}| \geq \Delta_{th} \\ 0 & |w_{l,i}| < \Delta_{th} \end{cases} \quad (1)$$

$$\alpha = E(|\boldsymbol{w}_{l,i}|), \quad \forall\{i||\boldsymbol{w}_{l,i}| \geq \Delta_{th}\} \quad (2)$$

where $\boldsymbol{w}_l$ denotes the full precision weight tensor of layer l, $\boldsymbol{w}_l'$ is the weight after ternarization. α is the layer-wise weight scaling factor. In this work, we employ single scaling factor for both positive and negative weights, since such symmetric weight scaling factor can be easily extracted and integrated into following Batch Normalization layer or ReLU activation function for the hardware implementation. It is also worth to note that the whole layer share the same scaling factor, which is negligible compared to the whole model size.

Then, in ②, the input mini-batch takes the ternarized model for inference and calculates loss w.r.t targets. In this step, since all of the weights are in ternary values, all the dot-product operations in layer l can be expressed as:

$$\boldsymbol{x}_l^T \cdot \boldsymbol{w}_l' = \boldsymbol{x}_l^T \cdot (\alpha \cdot Tern(\boldsymbol{w}_l)) = \alpha \cdot (\boldsymbol{x}_l^T \cdot Tern(\boldsymbol{w}_l)) \quad (3)$$

where $\boldsymbol{x}_l$ is the vectorized input of layer l. Since $Tern(\boldsymbol{w}_{l,i}) \in \{-1, 0, +1\}$, $\boldsymbol{x}_l^T \cdot Tern(\boldsymbol{w}_l)$ can be easily realized through addition/subtraction without multi-bit or floating point multiplier in hardware, which greatly reduces the computational complexity.

In ③, the full precision weights will be updated during back-propagation. Then, the next iteration begins to recompute weight scaling factor and ternarize weights as described in step-①. Meanwhile, since the ternarization function owns zero derivatives almost everywhere, which makes it impossible to calculate the gradient using chain rule in backward path. Thus, the Straight-Through Estimator (STE) [21][7] of such ternarization function in the back-propagation is applied to calculate the gradient as follow:

$$\text{Backward}: \quad \frac{\partial g}{\partial w} = \begin{cases} \frac{\partial g}{\partial w'} & if\ |w| \leq 1 \\ 0 & otherwise \end{cases} \tag{4}$$

where the gradient clipping prevents the full precision weight growing too large and constrains the scaling factor α as well. Meanwhile, it sets the upper bound of threshold Δ_{th} as β.

III. Experimental Setup and Results:

Datasets: We conduct experiments of TGAN on several datasets to evaluate the performance of proposed algorithm, including Fashion-MNIST [22], CIFAR-10 [23] and STL-10 [24]. Fashion-MNIST is a gray-scale dataset which contains 10,000 training images and 10,000 testing images. The image is in the size of 28×28 and there are 7000 images in each of the 10 fashion categories. We also use CIFAR-10 [23] for RGB images of size 32×32. It has 60,000 images evenly distributed in 10 different classes. Among them 50,000 examples for training and remaining 10,000 for testing. Finally, we use STL-10 [24] along side CIFAR-10 to evaluate the performance quantitatively. It is similar to CIFAR-10 dataset except that it has 100,000 unlabeled images for unsupervised learning and only 500 labeled images for training.

Evaluation Metrics: There is yet a universal figure of merit to evaluate the performance of GAN [25]. Most of the recent works have followed qualitative analysis by showing the images generated by the generator [1]. Observing the values of loss function or Wasserstein distance gives a good overview of the training stability [19]. However, most of them do not give proper information on the quality and variation of the generated image. One way to measure these information quantitatively is to use Inception Score (IS) [26]. IS uses a pre-trained inception V3 [27] network to calculate the quality of generated images. Considering we have an image x and expecting the network to predict output label y for that image. Then the generator's inception score would be:

$$IS(G) = exp(E_{x \sim p_g} D_{KL}(p(y|x)||p(y))) \tag{5}$$

Here $D_{KL}(p|q)$ is the KL divergence between p and q distribution. If the images generated contain clear objects then p($y|x$) should be low entropy. This means that the Inception Network is highly confident that there is a single object. A better performing GAN algorithm should generate a high diversity of images from all classes meaning p(y) should be high entropy. If both of these conditions are satisfied then we expect a large IS for the GAN. A high IS indicates clarity and diversity among the generated images.

Results and Analysis: We conduct several sets of experiments on both CIFAR-10 and STL10 using DCGAN,WGAN-CP and WGAN-GP. First, we use 32-bit floating point number weights as the baseline to train GAN. Since we are the first to ternarize GAN, as ablation study, we train several variant GANs from scratch using 2-bit weight quantization method in [7], trained weight ternarization method in [20] and our proposed ternarized-weight training method, respectively, to prove the effectiveness of our proposed method. Note that, all above weight ternarization or quantization method could achieve the same model compression rate. Inception Score (IS) is used to evaluate GAN's performance and higher score is better.

Table I
INCEPTION SCORE (IS) FOR DIFFERENT ARCHITECTURE ON CIFAR-10 AND STL10 DATASETS.

MODEL		CIFAR10	STLL10
DCGAN	32-bit	5.71 ± 0.2	2.91 ± 0.2
	Dorefa-Net 2-bit [7]	1.24 ± 0.003	1.39± 0.007
	TWN [20]	1.09 ± 0.003	1.45 ± 0.008
	Proposed TGAN	**4.52 ± 0.1**	**2.91 ± 0.3**
WGAN-CP	32-bit	4.97 ± 0.15	3.01 ± 0.1
	Dorefa-Net 2-bit [7]	3.84 ± 0.09	2.37 ± 0.05
	TWN [20]	4.26 ± 0.07	2.78± 0.06
	Proposed TGAN	**3.76 ± 0.07**	**2.31 ± 0.09**
WGAN-GP	32-bit	5.65 ± 0.008	2.86 ± 0.09
	Dorefa-Net 2-bit [7]	4.70 ± 0.05	2.31 ± 0.012
	TWN [20]	4.45± 0.05	2.68 ± 0.015
	Proposed TGAN	**5.11 ± 0.01**	**2.49 ± 0.05**

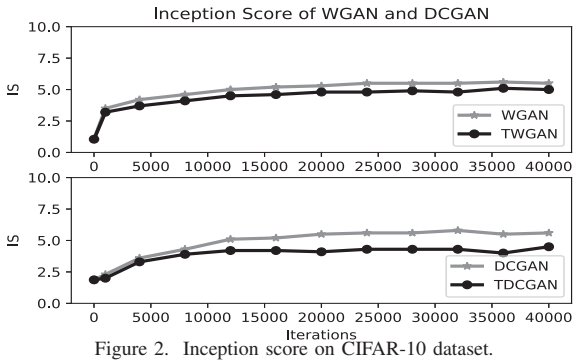

Figure 2. Inception score on CIFAR-10 dataset.

Table I summarizes the IS of ternary/quantized and full-precision weight training. As for CIFAR-10, WGAN-GP provides the best IS (5.11). We observe that IS of our proposed

TGAN only degrades by ∼0.5 compared to its full precision training. Moreover, the score achieved using ternary WGAN-GP shows better performance than full-precision WGAN-CP. The similarity in inception score between TGAN and full-precision training indicating our proposed TGAN can be effectively used for training GAN. For STL10 dataset, our ternary trained weight DC-GAN gives almost similar score as the 32-bit weight training. On the other hand, we again observe a small degradation of IS for WGAN-CP as was the case for CIFAR 10. Overall the training performance of our proposed ternary GAN does not deteriorate much compared to 32-bit weight training and outperforms existing weight ternarization/ quantization methods, making it a suitable choice for hardware implementations. One of the major concern of GAN training

Figure 3. Fashion-MNIST generated images by Ternary WGAN-GP

is that it suffers from training instability and convergence problem. We plot the change of IS after each epoch to observe the stability of the proposed method compared to the full precision one. In Fig. 2, we see TGAN actually converges in a similar manner as the original full-precision training for both DC-GAN and WGAN. Additionally, we display the images generated by TGAN architectures in Fig. 3. This figure gives us a more qualitative understanding of the performance of TGAN. Images generated by our TGAN looks almost similar to the original Fashion MNIST images. Both quantitative and qualitative evidence demonstrate that our proposed ternary GAN performs on par with the state-of-the-art GAN training algorithms. So considering the algorithm's suitability for efficient hardware implementation, it becomes an obvious choice for training of GAN using accelerators.

IV. Accelerating TGAN training

In this section, we show that performing the most computationally-intensive convolution and deconvolution operations in GAN, converted to addition/subtraction (add/sub) in TGAN, within a PIM accelerator brings significant advantages for training: (1) Reducing main memory access which leads to great reduction of data transfer between CPU/GPU and memory, (2) Reducing the energy consumption of convolutional layers through utilizing intrinsic and efficient in-memory computing, and (3) Accelerating the forward propagation by employing parallel in-memory processing. In the following, we first analyze GAN training mechanism from computation perspective and then propose a PIM-TGAN accelerator capable of implementation of each layer. Then, we describe the mapping method and explore a parallelism scheme to further improve GAN training efficiency.

A. GAN Training

The computational blocks of a sample GAN and the dataflow of training for both Generator (G) and Discriminator (D) are shown in Fig. 4. Each block is divided into different computational layers. As can be seen, the main computation in G and D are deconvolution ($DConv$) and convolution ($Conv$), respectively followed by Batch Normalization and/or an activation function. GAN training is basically accomplished in n consecutive phases (3 for DC-GAN) that are depicted by different colors in Fig. 4 (i.e. D1, D2 and G). In D1 training phase, a real sample is fed to D and propagates forward through different computational blocks until the calculation of Loss function. The loss function for D1 is calculated based on accurate label ('1' for real sample). The procedure continues by back propagation of the error and partial gradients towards the the first layer ($GD1$). During D2 training phase, injection of a random noise continued by up-sampling of G, generates a fake sample with the same dimension as real samples. The generated sample flows through D and a loss function is calculated based on accurate labels. Again, the gradients are calculated and propagates all the way back to the first layer of D ($GD1$). After D2 training phase, the weights of D is *updated* according to gradients achieved in D1 and D2 training phases and based on designated GAN architecture; in case of DC-GAN by summing up the gradients. During G training phase, G and D are concatenated again to make a large network, at which a random noise propagates all the way through G and D and loss function is calculated based inaccurate labels. The errors propagates backward to GG1 and finally the weights of G is *updated* while D remains unchanged. Note that there is no computation difference between $Conv$ and $DConv$ layers. Mathematically, we can implement a $DConv$ with a direct $Conv$ [28], [6]. This is readily accomplished by adding zeros between each input in the feature maps with zero padding and then computing the convolution between the extended input feature maps and kernels. As we are using the ternarized weights in the forward path, all the $Conv$ and $DConv$ operations are converted to add/sub.

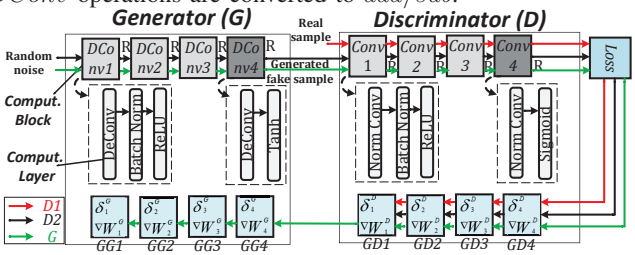

Figure 4. GAN training. D1 and D2 show the dataflow for first and second training phases of Discriminator, G represents Generator's training phase.

B. PIM-TGAN

In this subsection, we develop a processing-in-memory accelerator for ternary GAN (PIM-TGAN) as a potential solution to better address computation and data transfer bottlenecks of different GAN architectures. As discussed in Section III,

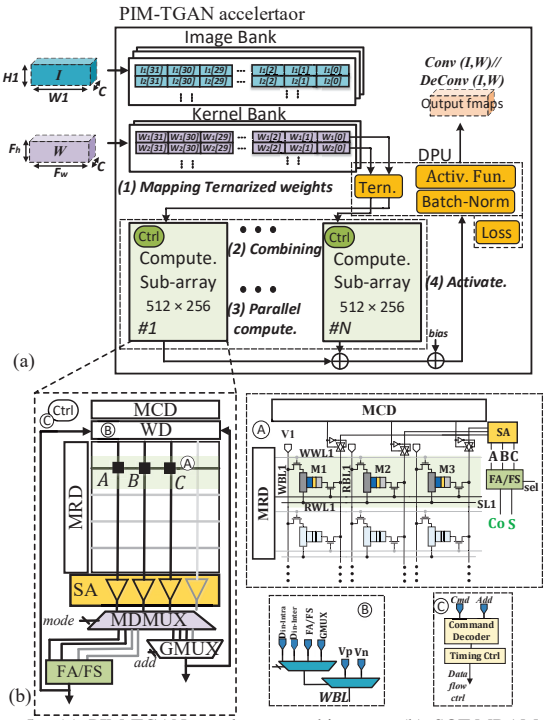

Figure 5. (a) PIM-TGAN accelerator architecture, (b) SOT-MRAM based computational sub-array architecture of PIM-TGAN.

TGAN imposes the least possible computational complexity to underlying hardware due to the ternarization of weights in forward path compared to well-trained GANs which use floating point operations on both CPU and GPU. Thus, in this work, we mainly focus on accelerating the forward path of training in both generator and discriminator units, however PIM-TGAN is utilized to process entire GAN training phases discussed in previous subsection. The architectural diagram of PIM-TGAN accelerator is shown in Fig. 5a consisting of Image and Kernel Banks, SOT-MRAM-based computational sub-arrays and a Digital Processing Unit (DPU) including four ancillary units (i.e. Ternarizer, Batch Normalization, Activation Function and Loss Function). The proposed SOT-MRAM array architecture, as will be thoroughly discussed in next subsection, can support flexible bit-width add/sub operation required in both forward and backward paths in GAN's training.

Assume Input fmaps (I) and Kernels (W) are initially stored in Image Banks and Kernel Banks of memory, respectively. As depicted in Fig. 5a, weights can be constantly ternarized before mapping into computational sub-arrays, which is required for both $Conv$ and $DConv$ blocks. Therefore, *mapping of ternarized weights* is the first processing step that need be accomplished by the accelerator. This step is performed using DPU's Ternarizer (Tern. unit in Fig. 5a) and then the results are mapped to computational sub-arrays. Note that, ternarized shared kernels can be utilized for different inputs. Next two processing steps are *combining* and *parallel computation* that will be performed within computational sub-arrays.

1) Computational Sub-array: The proposed SOT-MRAM sub-array architecture could work in dual mode that perform both memory read-write and logic operations. Fig. 5b (R.H.S.) shows the architecture of a 2×3 memory array. Each SOT-MRAM cell is associated with the Write Word Line (WWL), Read Word Line (RWL), Write Bit Line (WBL), Read Bit Line (RBL), and Source Line (SL) to perform typical memory operations. Moreover, in our design, any three cells in identical row could be sensed simultaneously to implement an in-memory logic function after Sense Amplifiers (SA). The peripheral decoders (active-high output) control the activation of current path through the array. Voltage drivers are used with the WBLs for providing the required write voltage. A voltage mode SA [29] is connected to the RBL for sensing the total resistance in the selected current path during Read and Computing mode. We have devised a Mode demultiplexer (MDMUX) right after SAs to switch between memory mode and computing method based on Ctrl unit. As can be seen in block-level sub-array architecture (Fig. 5b (L.H.S.)), the output of each SA is routed to MDMUX. According to the mode selector, output data can be routed to either GMUX (i.e. global Mux for interfacing with other sub-arrays) or FA/FS unit.

Memory Mode: To *write* a bit in any of the SOT-MRAM cells, for example in the cell of 1st row and 1st column, write current should be injected through the heavy metal substrate of SOT-MRAM. To activate this write current path, WWL1 should be activated by Memory Row Decoder (MRD) and SL1 is grounded, while all the other lines are kept floating. Now, in order to write '1' (/'0'), the voltage driver (V1) connected with WBL1 is set to positive (/negative) write voltage. This allows sufficient charge current flows from V1 to ground (/ground to V1), leading to MTJ resistance in High-R_{AP} (/Low-R_P). For typical memory *read*, a read current flows from the selected SOT-MRAM cell to ground, generating a sense voltage at the input of SA, which is compared with memory mode reference voltage ($V_{sense,P}<V_{ref}<V_{sense,AP}$). Now, if the path resistance is higher (/lower) than R_{ref}, i.e. R_{AP} (/R_P), then the SA produces High (/Low) voltage indicating logic '1' (/'0').

Computing Mode: The key idea behind exploiting a CMOS FA/FS unit is to realize a fast in-memory full adder (/subtractor) after SAs to efficiently process the data avoiding inevitable operand write-back in conventional in-memory adder designs as well as accelerating in-memory processing. Memory Column Decoder (MCD) is modified such that it can activate more than one RBL at the same time. As a result, more than one column can be sensed and routed from SAs to FA/FS unit. Assume A, B and C operands (in Fig. 5b L.H.S.) correspond to M1, M2 and M3 memory cells in Fig. 5b (A), respectively. The operands are readily read out and fed to FA/FS unit. According to the selector, this unit yields Sum(/Difference) and Carry(/Borrow) bits in a single cycle.

2) Mapping Method: As shown in Fig. 4, $Conv$ and $DConv$ are the main computational blocks of TGAN that are replaced by add/sub operations due to ternarization, thus add/sub is the most critical unit of the accelerator, as it is responsible for the most iterative computational blocks which takes up the vast majority of the run-time. The add/sub units must keep high throughput and resource efficiency while

handling different input widths at run-time. Here we present an in-memory convolver within PIM-TGAN to efficiently handle multi-bit add/sub operations. While there are few designs for in-memory adder/subtractor in literature [30], to the best of our knowledge, this work is the first proposing a fast (2-cycle) and parallelable in-memory add/sub method.

We use the data organization depicted in Fig. 6 as the main mapping method to perform an n-bit addition (/subtraction) operation within PIM-TGAN's computational sub-arrays. As shown, operands ($a_{n-1}...a_1a_0$ and $b_{n-1}...b_1b_0$) are initially loaded and organized into different memory rows of PIM-TGAN's sub-array such that one computation can be performed per memory cycle. Here in Fig. 6 L.H.S., LSBs of two operands along with carry-in (ci) are selected and processed to generate Carry-out ($c0$) and Sum ($s0$). After the computation, the results need to be stored in the sub-array to be prepared for next computing round. This can be fulfilled using the modified Write Driver (WD) and MRD in one cycle (memory write). The same process continues towards the MSBs computation (Fig. 6 R.H.S.). Therefore, each computational sub-array is able to implement an n-bit add/sub operations in $2\times n$ cycles (n read + n write). Finally, $c_{n-1}s_{n-1}...s_1s_0$ is produced and stored in designated computational sub-array.

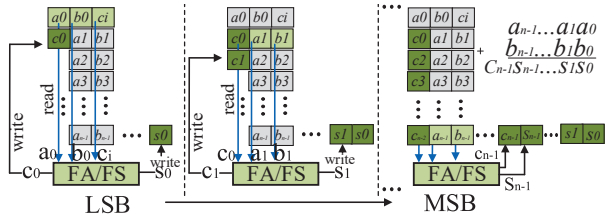

Figure 6. Realization of n-bit in-memory addition in *PIM-TGAN*.

Leveraging the presented idea in multiple PIM-GAN's sub-arrays can provide a parallel computation for PIM-TGAN. Fig. 7a shows the requisite data organization and computation of $Conv$ and $DConv$ layers. Initially, c channels (here, 4) in the size of $kh \times kw$ (here, 3×3) are selected from input batch and accordingly produce a combined batch w.r.t. the corresponding {-1,+1} kernel batch. This combination is readily accomplished by changing the sign-bit of input data corresponding to its kernel data. The combined batch is then mapped to the designated computational sub-arrays. Considering 16-activated sub-arrays (within 4 memory matrix (mat)), each combined batch's channel (Ch) can be processed using four parallel sub-arrays as depicted Fig. 7a. Here, Ch-1 to Ch-4 are respectively mapped to mat-1 to mat-4. After mapping, the parallel activated sub-arrays of PIM-TGAN operate to produce the output feature maps leveraging the same add/sub method shown in Fig. 6.

C. Parallelism

In this subsection, we develop a Fully-Pipelined Computation optimization method called *FPC* leveraging spatial parallelism method (*SP*) presented in [6] and a new method to further accelerate PIM-TGAN by increasing the system throughput. In training phase, the input data are typically processed in a batch size b (e.g. 8 or 64). Fig. 8 intuitively shows

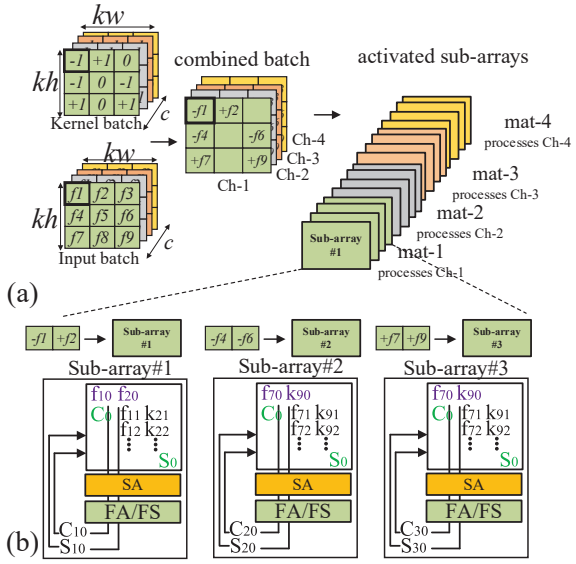

Figure 7. (a) Combining and (c) Parallel computing steps in *PIM-TGAN*.

the presented method with a small batch size of 2. A new batch can enter the pipeline only if all inputs in previous batch are processed. However, the pipelining can be readily achieved in PIM-TGAN by duplicating the data for intermediate layers. The pipeline for b1 and b2 are depicted in Fig. 8. Considering DL as the number of discriminator's layers, D1 training phase takes DL+1+DL+(b-1) cycles to be done (b-1 cycles to drain a batch from pipeline). With a careful observation of GAN training phases, the authors in [6] have shown that the discriminator's phases can be simultaneously co-trained for each batch. Fig. 8 shows that there is no data-dependency between discriminator's phases. We exploit the same method; D1 and D2 phases take different computational sub-arrays where $Conv$ and $DConv$ layers are being processed at a same time. Considering GL as the number of generator's layers, D2 latency is GL+DL+1+DL+1+(b-1) in order to update the D. In addition, we observe that after calculation of D2's Loss function and back-propagation of the results to GD4 layer, the generator training phase (G) for different batches can be started while corresponding GD3 is being processed in D2. This phase takes 2DL+2GL+2+(b-1). Regardless of pipelining, GAN training imposes b×(D1+D2+G) cycles.

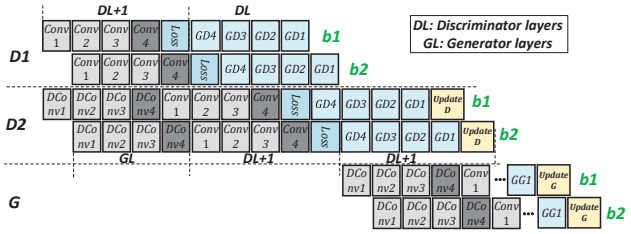

Figure 8. Fully-paralleled training method of PIM-TGAN.

V. Performance Evaluation

In the following, we compare PIM-TGAN with other possible GAN acceleration solutions (based on ReRAM, ASIC and GPU) running two GAN architectures (WGAN-GP and

DC-GAN). It is obvious that enlarging the chip area brings in a higher performance for PIM-TGAN and other designs due to the increased number of sub-arrays or computational units, though the die size directly impacts the chip cost. Therefore, in order to have a fair comparison, the area-normalized results (performance/energy per area) will be reported henceforth.

A. Accelerators' Setup

PIM-TGAN: We setup the PIM-TGAN's memory sub-array organization with 256 rows and 512 columns per mat organized in a H-tree routing manner, 2×2 mats per bank, 8×8 banks per group; in total 16 groups and 512Mb total capacity. To assess the performance of PIM-TGAN as a new PIM platform, a comprehensive device-to-architecture evaluation framework is developed. First, at the device level, we jointly use the Non-Equilibrium Green's Function (NEGF) and Landau-Lifshitz-Gilbert (LLG) with spin Hall effect equations to model SOT-MRAM bitcell [16]. For the circuit level simulation, a Verilog-A model of 2T1R SOT-MRAM device is developed to co-simulate with the interface CMOS circuits in Cadence Spectre and SPICE. 45nm NCSU Product Development Kit (PDK) library [31] is used in SPICE to verify the proposed design and acquire the performance of designs. Second, an architectural-level simulator is built based on NVSim [32]. The controllers and add-on circuits are synthesized by Design Compiler [33] with an industry library. Third, a behavioral-level simulator is developed in Matlab calculating the latency and energy that PIM-TGAN spends. ***ReRAM:*** A Prime-like [10] accelerator with two full functional (FF) sub-arrays and one buffer sub-array per bank (totally 64 sub-arrays) were considered for evaluation. In FF subarrays, for each mat, there are 256×256 ReRAM cells and eight 8-bit reconfigurable SAs. For evaluation, NVSim simulator [32] was extensively modified to emulate Prime functionality. Note that the default NVSim's ReRAM cell file (.cell) was adopted for the assessment. ***ASIC:*** We developed a YodaNN-like [34] ASIC accelerator. To have a fair comparison, we select two versions with either 64 tiles or 256 tiles. We synthesized the designs with Design Compiler [33] under 45 nm process node. The eDRAM and SRAM performance were estimated using CACTI [35]. ***GPU:*** We used the NVIDIA GTX 1080Ti Pascal GPU. It has 3584 CUDA cores running at 1.5GHz (11TFLOPs peak performance). The energy consumption was measured with NVIDIA's system management interface. We scaled the achieved results by 50% to exclude the energy consumed by cooling, etc.

B. Energy Efficiency

Fig. 9a shows the PIM-TGAN's energy-efficiency results (frames per joule) using two parallelism methods (i.e. FPC and SP) compared to different accelerators for performing a similar task with a batch size of 8 and 64. As can be seen, the larger the b is, the lower energy-efficiency is obtained. As shown, PIM-TGAN solution offers the highest energy-efficiency normalized to area compared to others owning to its fast, energy-efficient and parallel operations. We observe that PIM-TGAN's FPC solution shows $\sim 1.6\times$ better energy-efficiency than that of SP. We also observe that PIM-TGAN's FPC solution is $\sim 1.8\times$, $9.2\times$ and $25.6\times$ more energy-efficient than the best ASIC solution, ReRAM design and GPU. This energy reduction mainly comes from three sources: 1) standard $Conv$ and $DeConv$ operations in forward path are replaced with energy-efficient add/sub operations due to ternarization, 2) PIM-TGAN's parallelism which massively reduces the latency of operations and 3) bulk and energy-efficient in-memory operations of PIM-TGAN.

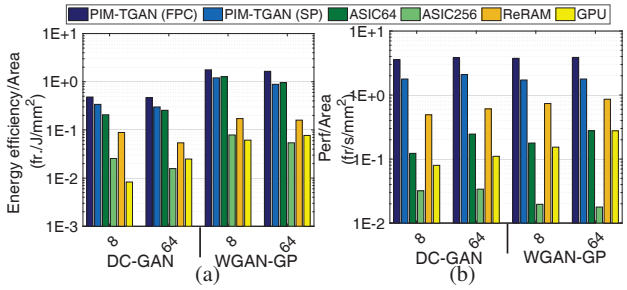

Figure 9. (a) Energy-efficiency and (b) Performance evaluation of different accelerators normalized to area (Y-axis=Log scale).

C. Performance

Fig. 9b shows and compares the PIM-TGAN performance (frames per second) results normalized with area with different accelerators. Based on the results, we observe that PIM-TGAN's FPC is $\sim 2\times$ faster than SP method. In addition, it is $\sim 22\times$ and $18\times$ faster on average than GPU and ASIC-64 solutions. This is mainly because of (1) ultra-fast and parallel in-memory operations of PIM-TGAN compared to multi-cycle ASIC and GPU operations and (2) the existing mismatch between computation and data movement in ASIC designs and even GPU solution. As a result, ASIC-256 with more tiles does not necessarily show a higher performance. We can also observe that the larger the batch is, the higher performance is obtained for PIM-TGAN's FPC solution compared SP owning to the its more paralleled computations as discussed earlier. Additionally, it can be seen that PIM-TGAN is $5.4\times$ faster that ReRAM solution. Note that ReRAM design employs matrix splitting due to intrinsically limited bit levels of RRAM device so multiple sub-arrays are occupied. This can further limit the parallelism methods. Besides, ReRAM crossbar has a large peripheral circuit's overhead such as buffers and DAC/ADC which contribute more than 85% of area [10].

D. Memory Wall

Fig. 10a depicts the memory bottleneck ratio i.e. the time fraction at which the computation has to wait for data and on-/off-chip data transfer obstructs its performance (memory wall happens) for DC-GAN structure under different implementations. The evaluation is performed according to the peak performance and experimentally extracted results for each platform considering number of memory access. The results[1] show the PIM-TGAN's favorable solution for solving memory wall issue. (1) We observe that PIM-TGAN and ReRAM solutions spend less than $\sim$ 20% time for memory

[1]GPU data could not be accurately reported for this evaluation.

access and data transfer. However, ASIC accelerator spends more than 90% time waiting for the loading data. (2) In the larger batch size, ReRAM solution shows even lower memory bottleneck ratio compared with PIM-TGAN. This comes from two sources: (1) increased number of computational cycles and (2) unbalanced computation and data movement of PIM-TGAN due to limitation in number of activated sub-arrays when number of operands increases. The less memory wall ratio can be interpreted as the higher resource utilization ratio for the accelerators which is plotted in Fig. 10b. We observe that PIM-TGAN can efficiently utilize more than 55% of its computation resources. Overall, PIM-TGAN and ReRAM solutions can demonstrate the highest ratio (up to 65% which reconfirms the results reported in Fig. 10a.

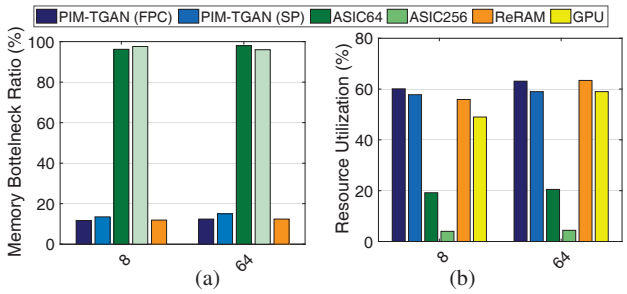

Figure 10. (a) The memory bottleneck ratio and (b) resource utilization ratio for DC-GAN structure.

VI. Conclusion

Conventional training methodology to train generative adversarial network using full-precision weights introduces a lot of memory and computational expenses. Here, we introduced a novel GAN training algorithm using ternary weights which eliminates the need for computationally-expensive multiplication operations in hardware. Additionally, we presented a novel processing-in-memory accelerator (PIM-TGAN) based on SOT-MRAM to further accelerate the ternary training process. Our simulation results showed that, with almost the same inception score to the baseline GAN on different datasets, PIM-TGAN can obtain $\sim$25.6$\times$ better energy-efficiency and 22$\times$ speedup compared to GPU platform.

References

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

[2] 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.

[3] S. Reed, Z. Akata, X. Yan, L. Logeswaran, B. Schiele, and H. Lee, "Generative adversarial text to image synthesis," *arXiv preprint arXiv:1605.05396*, 2016.

[4] A. Kadurin, S. Nikolenko, K. Khrabrov, A. Aliper, and A. Zhavoronkov, "drugan: an advanced generative adversarial autoencoder model for de novo generation of new molecules with desired molecular properties in silico," *Molecular pharmaceutics*, vol. 14, no. 9, pp. 3098–3104, 2017.

[5] Y. Taigman, A. Polyak, and L. Wolf, "Unsupervised cross-domain image generation," *arXiv preprint arXiv:1611.02200*, 2016.

[6] F. Chen, L. Song, and Y. Chen, "Regan: A pipelined reram-based accelerator for generative adversarial networks," in *Design Automation Conference (ASP-DAC), 2018 23rd Asia and South Pacific*. IEEE, 2018, pp. 178–183.

[7] S. Zhou *et al.*, "Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients," *arXiv preprint arXiv:1606.06160*, 2016.

[8] M. Courbariaux *et al.*, "Binaryconnect: Training deep neural networks with binary weights during propagations," in *Advances in Neural Information Processing Systems*, 2015, pp. 3123–3131.

[9] J. Song, "Binary generative adversarial networks for image retrieval," *arXiv preprint arXiv:1708.04150*, 2017.

[10] P. Chi *et al.*, "Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory," in *ISCA*. IEEE Press, 2016.

[11] S. Angizi *et al.*, "Imce: energy-efficient bit-wise in-memory convolution engine for deep neural network," in *Proceedings of the 23rd ASP-DAC*. IEEE Press, 2018, pp. 111–116.

[12] S. Angizi, Z. He *et al.*, "Cmp-pim: an energy-efficient comparator-based processing-in-memory neural network accelerator," in *Proceedings of the 55th Annual Design Automation Conference*. ACM, 2018, p. 105.

[13] S. Aga *et al.*, "Compute caches," in *HPCA*. IEEE, 2017, pp. 481–492.

[14] V. Seshadri *et al.*, "Ambit: In-memory accelerator for bulk bitwise operations using commodity dram technology," in *Micro*. ACM, 2017, pp. 273–287.

[15] B. C. Lee *et al.*, "Architecting phase change memory as a scalable dram alternative," in *ACM SIGARCH Computer Architecture News*, vol. 37. ACM, 2009.

[16] Z. He *et al.*, "High performance and energy-efficient in-memory computing architecture based on sot-mram," in *NANOARCH*. IEEE, 2017, pp. 97–102.

[17] M. Arjovsky, S. Chintala, and L. Bottou, "Wasserstein gan," *arXiv preprint arXiv:1701.07875*, 2017.

[18] I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin, and A. C. Courville, "Improved training of wasserstein gans," in *Advances in Neural Information Processing Systems*, 2017, pp. 5769–5779.

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

[20] F. Li *et al.*, "Ternary weight networks," in *The 1st NIPS Workshop on Efficient Methods for Deep Neural Networks*, 2016.

[21] Y. Bengio *et al.*, "Estimating or propagating gradients through stochastic neurons for conditional computation," *arXiv:1308.3432*, 2013.

[22] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms," *arXiv preprint arXiv:1708.07747*, 2017.

[23] A. Krizhevsky and G. Hinton, "Convolutional deep belief networks on cifar-10," *Unpublished manuscript*, vol. 40, p. 7, 2010.

[24] A. Coates, A. Ng, and H. Lee, "An analysis of single-layer networks in unsupervised feature learning," in *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, 2011, pp. 215–223.

[25] L. Theis, A. v. d. Oord, and M. Bethge, "A note on the evaluation of generative models," *arXiv preprint arXiv:1511.01844*, 2015.

[26] S. Barratt and R. Sharma, "A note on the inception score," *arXiv preprint arXiv:1801.01973*, 2018.

[27] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 2818–2826.

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

[29] X. Fong *et al.*, "Spin-transfer torque devices for logic and memory: Prospects and perspectives," *IEEE TCAD*, vol. 35, 2016.

[30] S. Angizi *et al.*, "Rimpa: A new reconfigurable dual-mode in-memory processing architecture with spin hall effect-driven domain wall motion device," in *ISVLSI*. IEEE, 2017, p. (accepted).

[31] (2011) Ncsu eda freepdk45. [Online]. Available: http://www.eda.ncsu.edu/wiki/FreePDK45:Contents

[32] X. Dong *et al.*, "Nvsim: A circuit-level performance, energy, and area model for emerging non-volatile memory," in *Emerging Memory Technologies*. Springer, 2014, pp. 15–50.

[33] S. D. C. P. V. . Synopsys, Inc.

[34] R. Andri *et al.*, "Yodann: An ultra-low power convolutional neural network accelerator based on binary weights," in *ISVLSI*. IEEE, 2016, pp. 236–241.

[35] N. Muralimanohar *et al.*, "Cacti 6.0: A tool to model large caches," *HP Laboratories*, pp. 22–31, 2009.