Towards Efficient Microarchitectural Design for Accelerating Unsupervised GAN-based Deep Learning

Mingcong Song, Jiaqi Zhang, Huixiang Chen, Tao Li
Department of Electrical and Computer Engineering, University of Florida, Gainesville, USA
{songmingcong, jiaqizhang, stanley.chen}@ufl.edu, taoli@ece.ufl.edu

***Abstract*—Recently, deep learning based approaches have emerged as indispensable tools to perform big data analytics. Normally, deep learning models are first trained with a supervised method and then deployed to execute various tasks. The supervised method involves extensive human efforts to collect and label the large-scale dataset, which becomes impractical in the big data era where raw data is largely un-labeled and uncategorized. Fortunately, the adversarial learning, represented by Generative Adversarial Network (GAN), enjoys a great success on the unsupervised learning. However, the distinct features of GAN, such as massive computing phases and non-traditional convolutions challenge the existing deep learning accelerator designs. In this work, we propose the first holistic solution for accelerating the unsupervised GAN-based Deep Learning. We overcome the above challenges with an algorithm and architecture co-design approach. First, we optimize the training procedure to reduce on-chip memory consumption. We then propose a novel time-multiplexed design to efficiently map the abundant computing phases to our microarchitecture. Moreover, we design high-efficiency dataflows to achieve high data reuse and skip the zero-operand multiplications in the non-traditional convolutions. Compared with traditional deep learning accelerators, our proposed design achieves the best performance (average 4.3X) with the same computing resource. Our design also has an average of 8.3X speedup over CPU and 6.2X energy-efficiency over NVIDIA GPU.**

I. Introduction

Recently, deep learning based techniques, such as deep convolutional neural networks (CNNs) [1], have achieved amazing success in many fields, such as computer vision and natural language processing, because of their ability to achieve accuracy close to or even better than human-level perception. Normally, working with deep neural networks is a two-stage process. First, a neural network is trained using a labeled training dataset. Then, it is deployed to run inference to classify, recognize and process unknown inputs based on the previously trained parameters. Traditionally, the neural network is trained supervisedly, which suffers from critical disadvantages, especially in terms of scalability as it requires a huge amount of labeled data. Labeling millions of images involves extensive human efforts and is time-consuming. Moreover, supervised training only includes a predefined set of classes, which limits its accuracy on recognition of new classes.

To overcome the difficulties, researchers have turned to semi-supervised and unsupervised learning techniques [2]–[8]. Generative Adversarial Networks (GANs) [5] are of particular interest in unsupervised learning. Researchers at Google have utilized GAN to create more realistic 'imaginings' of the real world and the results could lead to robots that can learn WITHOUT human input [9]. One of the most important merits for GAN is that it can autonomously learn interpretable, useful feature representation from raw big data. The feature representation learned using unsupervised method could be further leveraged to implement richer applications, such as image classification [10], object detection [6], video prediction [11], mobile robots [12] and self-driving [13]. Now, GAN has become a mainstream algorithm for unsupervised learning, which can equip IoT nodes with a self-learning ability. The intelligent IoT nodes not only perform the inference task, such as object recognition, but also can improve its recognition capability autonomously with GAN's unsupervised learning. However, IoT nodes are normally sensitive to power consumption, where the power-hungry GPU is not applicable. Therefore, it is imperative to propose a power efficient GAN accelerator deployed on IoT nodes for on-line unsupervised learning.

Compared to the traditional supervised learning, such as CNN training, the unsupervised learning represented by GAN training involves more complex computations, including strided convolutions, transposed convolutions, and four-dimension convolutions. Although these operations could still utilize traditional CNN accelerators, the differences in the computing pattern, such as zero-inserting, result in low efficiency of existing CNN accelerators. Moreover, GAN training has more computing phases since it includes two networks' (Generator and Discriminator) updates. For GAN training, one iteration requires five forward passes and four backward passes, while there are only one forward pass and one backward pass in CNN training. Therefore, the distinct features of GAN challenge the current deep learning accelerator designs and demand customized microarchitectural optimizations to efficiently implement GAN.

In this paper, we demonstrate the following challenges in GAN accelerator design by thoroughly analyzing the procedure of GAN training. First, the synchronization operation (for loss calculation) in GAN training not only demands large memory to store the intermediate data but also restricts the parallel optimization. Second, the abundant and variable computing phases and the non-traditional convolutional operations in these phases further complicate the accelerator design. We overcome these challenges with an

2378-203X/18/$31.00 ©2018 IEEE
DOI 10.1109/HPCA.2018.00016

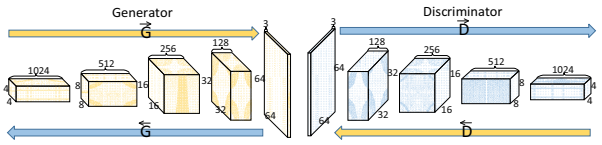


Figure 1. DCGAN

algorithm and architecture co-design approach. First, we defer the synchronization operation to the end of batch processing. We then propose a time-multiplexed design to efficiently map the abundant computing phases to our proposed microarchitectures. Specifically, Zero Free and Output STationary Microarchitecture (*ZFOST*) is responsible for the computation in forward data pass and backward error pass and the operations of weights updating are processed by Zero Free and Weight STationary Microarchitecture (*ZFWST*). By combining our high-efficiency dataflows and novel architectures, the zero-operand multiplications in non-traditional convolutions are eliminated without sacrifice of data reuse. Due to FPGA's good performance, better energy efficiency and high reconfigurability, we choose FPGA as our platform to demonstrate our GAN accelerator design. Evaluation results show that our proposed design achieves the best performance (average 4.3X) over the traditional deep learning accelerators. Our accelerator also has an average of 8.3X speedup over CPU and 6.2X energy-efficiency over NVIDIA GPU.

In summary, we make the following key contributions:

- We take the first step to help computer architecture community to understand the inefficiencies of current accelerators to support unsupervised training.
- To the best of our knowledge, this paper is the first work that presents the holistic solution for accelerating the un-supervised deep learning.
- Our proposed design features two novel microarchitectures (*ZFOST* and *ZFWST*) and high-efficiency dataflows tailored for the non-traditional convolutions: *S-CONV*, *T-CONV* and *W-CONV*. It is worth pointing out that these microarchitectural optimizations can also be widely used in traditional CNN training procedure.

The rest of this paper is organized as follows. Section II introduces the background on GAN training. Section III illustrates the challenges of GAN accelerator design. Section IV describes our design, including algorithm analysis, microarchitecture design and dataflow optimization. Section V demonstrates the implementation of our GAN accelerator on FPGA. Section VI evaluates our design. Related works and conclusions are discussed in Sections VII and VIII, respectively.

II. BACKGROUND

A. Deep Convolutional Generative Adversarial Network (DCGAN)

Currently, the most popular GAN network is Deep Convolutional Generative Adversarial Network (DCGAN) [10], shown in Fig. 1. It consists of two networks: Generator and Discriminator. These two networks are iteratively trained competing against each other in a minimax game, where Generator learns to produce more and more realistic samples, and Discriminator learns to get better and better at distinguishing generated data from real data. TABLE I summarizes the symbols we use to explain GAN. Compared with the traditional CNN architectures that consist of Convolutional layers (CONV) and Pooling layers, the discriminator in DCGAN replaces max pooling with strided convolution (*S-CONV*), allowing the network to learn its spatial down-sampling. By shifting the kernel by stride of 2, half of the convolutional operations are skipped over in both vertical and horizontal directions, so the output size will be about a quarter of the input. As shown in Fig. 1, Generator has an inverse architecture of Discriminator, which consists of four transposed convolutional layers (*T-CONV*). The output feature maps of *T-CONV* are larger than its input feature maps (four times in Fig. 1). To implement the up-sampling, *T-CONV* needs to insert many zeros into its input feature maps before performing the convolutional computation. The details of zero-inserting will be discussed in Section III.

TABLE I. NOTATIONS USED TO EXPLAIN GAN

Symbol	Description
G	Generator. Function *G()* takes a vector as input and outputs a fake image.
D	Discriminator. Function *D()* takes an image as input and outputs a scalar to measure how likely that the image is real.
m	Size of a mini-batch.
$\tilde{x}_i$	A fake image generated by Generator.
x_i	An image from the real world.
δ^l	Error matrix of the l^{th} convolutional layer.
d^l	Output data of the l^{th} convolutional layer.
W^l	Weight matrix of the l^{th} convolutional layer.
S-CONV	Strided convolution (no zero-inserting).
T-CONV	Transposed convolution (zero-inserting in input).
W-CONV	Convolution for updating weights, zero-inserting in input when updating G and zero-inserting in kernel when updating D.
$\vec{D}/\vec{G}$	Forward pass of Discriminator/Generator. Uses *S-CONV/T-CONV* to compute.
$\overleftarrow{D}/\overleftarrow{G}$	Backward error pass of Discriminator/Generator. Uses *T-CONV/S-CONV* to compute.
$\overleftarrow{D_w}/\overleftarrow{G_w}$	Backward weight updating of Discriminator/Generator. Uses *W-CONV* to compute.

B. GAN Training

1) Using backpropagation to train GAN

GAN is trained using the backpropagation algorithm [5], [14] and its training phase consists of Discriminator and Generator update. Discriminator is trained to maximize the probability of correctly classifying real and synthetic samples by descending the gradient of its loss function (1):

$$\mathrm{loss}_{\mathrm{dis.}} = -\frac{1}{\mathrm{m}}\sum_{\mathrm{i}=1}^{\mathrm{m}}[\mathrm{D}(\mathrm{x_i}) - \mathrm{D}(\tilde{\mathrm{x}}_\mathrm{i})], \quad (1)$$

where x_i is the real data, $\tilde{x}_i$ is the fake data generated by Generator, and m is the mini-batch size. Here the loss function is Wasserstein distance [14] between real data and generated data. By maximizing the distance (increasing $D(x_i)$ and decreasing $D(\tilde{x}_i)$), Discriminator is trained to distinguish the real data from the generated data.

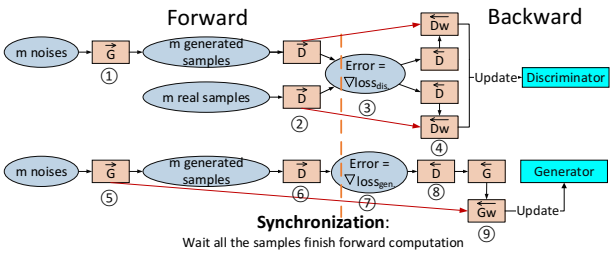


Figure 2. Computational Flow of GAN Training

Then Generator is updated to increase the value of $D(\tilde{x}_i)$ so that its generated data is closer to the real data by descending the gradient of its loss function (2),

$$\text{loss}_{\text{gen.}} = -\frac{1}{m}\sum_{i=1}^{m}[D(\tilde{x}_i)]. \quad (2)$$

By alternating gradient optimization between the two networks on new batches of real and generated data, GAN will slowly converge to producing data that is as realistic as the network is capable of modeling.

2) Computational flow of GAN training

Fig. 2 illustrates the computational flow of GAN training. In step ①, Generator generates a batch of fake samples (batch size is m). Then, one batch of real samples and one batch of fake samples are imported into Discriminator during step ②. Steps ① and ② belong to the forward phases and we name them as $\vec{G}$ and $\vec{D}$. And the core operations in $\vec{G}$ and $\vec{D}$ are *T-CONV* and *S-CONV* respectively. Step ③ calculates the error of output layer δ^L (L represents the last layer of Discriminator) based on the gradient of loss function of Discriminator ($\nabla \text{loss}_{dis.}$). In step ④, the error of the last layer is fed back into Discriminator to update its weights. This step consists of two backward phases: $\overleftarrow{D}$ and $\overleftarrow{D_w}$. First, the error δ^L is back-propagated to each layer ($1 \le l < L$) in $\overleftarrow{D}$ using (3):

$$\delta^l = (W^{l+1})^T \delta^{l+1} \circ \sigma', \quad (3)$$

where ° represents an element-wise multiplication and σ' is the derivate of activation function σ. $(W^{l+1})^T$ is the transpose of the weight matrix W^{l+1} for the $(l+1)^{th}$ layer and δ^{l+1} is error of the $(l+1)^{th}$ layer. This is a *T-CONV* computation. Its input is the error of the $(l+1)^{th}$ layer (δ^{l+1}) and its output is the error of the l^{th} layer (δ^l).

Based on the error δ^l computed in $\overleftarrow{D}$ and the intermediate output of the $(l-1)^{th}$ layer (d^{l-1}) from the earlier forward computation in $\vec{D}$, $\overleftarrow{D_w}$ calculates loss function's partial derivatives to weights (∇W^l) in each layer of Discriminator using (4):

$$\nabla W^l = d^{l-1}\delta^l. \quad (4)$$

Fig. 3 shows the core operation in $\overleftarrow{D_w}$, which is also a convolutional operation in nature. However, there is no accumulation operation after convolving the error of the l^{th} layer (δ^l) with the output of $(l-1)^{th}$ layer (d^{l-1}), which leads to four-dimension output matrices. Since this convolution is used to update the weights of GAN, we name this convolutional operation as *W-CONV*.

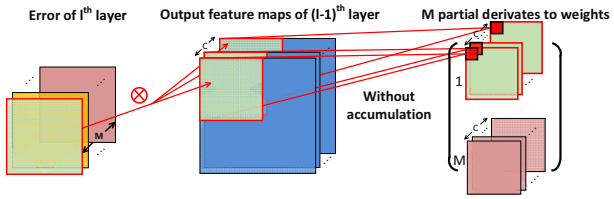


Figure 3. Calculation of Partial Derivates (*W-CONV*)

For the update of Generator, only images generated in Step ⑤ are fed into Discriminator in step ⑥. Having a batch of generated samples processed by Discriminator, we obtain the error in the output layer of Discriminator in step ⑦. Then the error is back-propagated to Generator from Discriminator (step ⑧). Finally, the weights of Generator are updated in Step ⑨ based on the errors calculated in backward $\overleftarrow{G}$ and intermediate data generated in step ⑤.

III. Challenges of GAN Accelerator Design

Although the core function of GAN is convolution, GAN's unique features, such as synchronization mechanism, a mass of computational phases and non-traditional convolutions, lead to inefficiency when existing CNN accelerators are directly applied to GAN training. In this section, we analyze the challenges of GAN accelerator design in detail.

A. Synchronization for Loss Calculation Restricts Parallel Optimization and Demands Large Memory

As shown in (1), the loss function is derived by averaging the outputs of Discriminator. Therefore, the backward computation has to wait until all the real and fake samples have completed their forward computations, which results in a synchronization operation (for loss calculation) in steps ③ and ⑦ in Fig. 2. This synchronization operation prevents the pipeline based accelerator design and implementation, such as PipeLayer [15] and ISAAC [16].

As illustrated in (4), the weight updating needs the intermediate output of each layer (d^l) generated in earlier forward computations. Therefore, we should buffer this data so that it can be used in weight updating. Since the loss function is calculated by averaging two batches (real and generated data) of Discriminator's outputs, the number of intermediate data we need to buffer is *2×batch size*, which greatly increases memory consumption. For example, DCGAN needs a ~126M-byte buffer when the batch size is 256. For GPU-based GAN training, GPU's abundant high-bandwidth memory can buffer all the intermediate data even using batch processing so the original training algorithm is good for GPU. However, for devices with limited high-bandwidth memory, such as FPGA, the intermediate data has to be stored in the off-chip memory, which will significantly increase latency and power consumption. To design a high performance and power-efficient accelerator for GAN, we should try to reduce the amount of intermediate data by optimizing the training algorithm so that it could be stored in the on-chip memory.

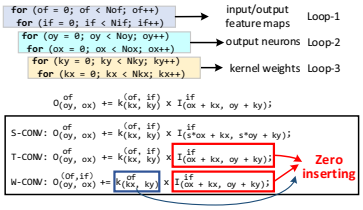


Figure 4. Pseudo Code of CONV

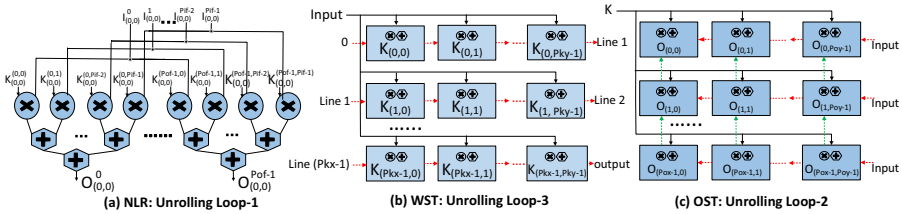


Figure 5. Traditional Computing Architectures for CONV

B. *The Abundant and Variable Computing Phases Require Tradeoffs between a Uniform and Per-Phase Design*

As shown in Fig. 2, there are four computing phases: forward $\vec{G}$, forward $\vec{D}$, backward $\overleftarrow{D}$ and backward $\overleftarrow{D_w}$, in the procedure of Discriminator update. Two more computing phases, backward $\overleftarrow{G}$ and backward $\overleftarrow{G_w}$, are needed for Generator update. Since the computation pattern varies in different phases, it is difficult for traditional CNN accelerators to efficiently handle the complex computations in all computing phases. Designing a specific architecture for each computational phase can achieve high processing efficiency. However, with the increasing number of computational phases, it becomes very costly to implement the per-phase architecture design on a single chip due to the limitation of computing and memory resource. Moreover, the alternative updating pattern between Discriminator and Generator challenges this per-phase architecture design. When one network is being updated, the other has to wait, which leads to idleness of computing architectures. For instance, since the computing architectures of $\overleftarrow{G_w}$ and $\overleftarrow{G}$ are exclusively utilized by Generator, they will be idle when updating Discriminator.

To increase utilization of the computing resources, a unified architecture is in need to enable time multiplexing among different computing phases. However, due to some essential differences among the various types of convolutional patterns, it is impossible to find an architecture that works efficiently for all of them. Therefore, the tradeoff between the per-phase, customized design and a uniform, common case design needs to be investigated.

C. *GAN Features Make Traditional Architectures Inefficient*

TABLE II lists all the terms that will be used when discussing CNN and GAN accelerators. And Fig. 4 illustrates the pseudo code for the convolutional operation, which is comprised of three levels of loops. To achieve high processing throughput, the loops are usually unrolled and mapped to a parallel computing hardware. By unrolling Loop-1, parallelism from input and output feature maps is employed in the architecture in Fig. 5(a). This computing architecture loads new kernel weights and input neurons every cycle, so it is called *NLR* (No Local Reuse). Although there is no temporal reuse in *NLR*, it supports spatial sharing of input neurons across different output feature maps and is used in [17]–[19]. Besides the spatial reuse of input neurons, the architecture in Fig. 5(b), which unrolls Loop-3, can temporally reuse the kernel weights stored in local registers during adjacent cycles, which minimizes energy consumption for weight read. Since

TABLE II. NOTATIONS USED TO EXPLAIN ACCELERATORS

Symbol	Description
$I^{if}_{(ix,iy)}$	The input neuron located at *(ix,iy)* of the if^{th} input feature map.
$O^{of}_{(ox,oy)}$	The output neuron located at *(ox,oy)* of the of^{th} output feature map (in *S-CONV,T-CONV*).
$O^{(of,if)}_{(ox,oy)}$	The output neuron located at *(ox,oy)* of the (of^{th}, if^{th}) output feature map (in *W-CONV*).
$K^{(of,if)}_{(kx,ky)}$	The weight located at *(kx,ky)* of the kernel that connects the if^{th} input feature map and the of^{th} output feature map (in *S-CONV,T-CONV*).
$K^{of}_{(kx,ky)}$	The weight located at *(kx,ky)* of the of^{th} kernel (in *W-CONV*).
N_{if}/N_{of}	Total number of input/output feature maps.
N_{ix}/N_{iy}	Total number of rows/columns in one input feature map.
N_{ox}/N_{oy}	Total number of rows/columns in one output feature map.
N_{kx}/N_{ky}	Total number of rows/columns of weights in one kernel.
P_{if}/P_{of}	Total number of input/output feature maps that are unrolled to run in parallel.
P_{ix}/P_{iy}	Total number of rows/columns of input neurons that are unrolled to run in parallel.
P_{ox}/P_{oy}	Total number of rows/columns of output neurons that are unrolled to run in parallel.
P_{kx}/P_{ky}	Total number of rows/columns of weights that are unrolled to run in parallel.
$PE^{l}_{x,y}$	Processing element located at *(x, y)* of the l^{th} PE channel. Each PE is responsible for a MAC operation.
Spatial sharing	The data is broadcast to many PEs in the one cycle.
Temporal sharing	The same data is reused by PEs during adjacent cycles.

this kind of architecture remains the same kernel weights in PEs (Processing Elements) while receiving new input neurons, it is named as *WST* (Weight STationary). [20], [21] exploit variants of *WST* in their works. In *OST* (Output STationary) [22], [23], shown in Fig. 5(c), multiple adjacent output neurons are mapped to the array of PEs by unrolling Loop-2, and these output neurons are fixed to certain PEs to reduce partial sum read/write. Thanks to the paths between the neighbor PEs, the input neurons can be temporally reused across different PEs. Besides, all the PEs in *OST* spatially share the kernel weight (K).

Although *S-CONV*, *T-CONV* and *W-CONV* in GAN share the nested-loop with the traditional convolutional operation in Fig. 4, there are some differences that prevent above traditional architectures from being efficiently applied to GAN.

1) Four-dimension outputs in W-CONV eliminate P_{if} unrolling in NLR

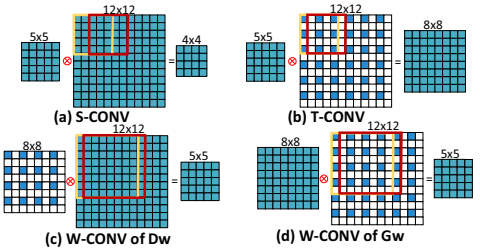


Figure 6. The Core Operations in GAN (zeros are white)

In traditional CNNs, each output feature map is calculated by accumulating all the weighted input feature maps. Therefore, in *NLR* (Fig. 5(a)), there is an adder tree that sums the results of P_{if} multipliers. However, as shown in Fig. 3, *W-CONV* has a four-dimension output, meaning only one input feature map is needed to compute each output neuron and only one multiplier is involved in the calculation of each output feature map, so the adder tree in *NLR* is of no use. Since *NLR* calculates P_{of} output neurons in one cycle, only P_{of} multipliers are working and $P_{of}\times(P_{if}-1)$ multipliers are idle when *W-CONV* is processed by *NLR*. Therefore, *NLR* is not suitable to perform *W-CONV*.

2) Down-sampling in convolutional operations increases PE idleness in WST

In *WST*, $P_{ky}\times P_{kx}$ PEs (*nPEs*) share a single input neuron in each cycle, but not all the input neurons are required to perform multiplication with each kernel weight. Take input $I_{(0,0)}$ as an example. Only the PE with $K_{(0,0)}$ operand produces useful result and other PEs do not contribute to any output neuron. In this case, a lot of PEs will be idle. Here, we derive a formula to quantify this idleness.

Since one input neuron is fed into *WST* in each cycle and the computation of a layer involves all the input neurons, the number of cycles needed to process a whole convolutional layer (*nCycles*) equals to the number of input neurons ($N_{of}\times N_{if}\times N_{iy}\times N_{ix}$). Based on Fig. 4, the number of MAC operations (*nMACs*) in a convolutional layer is $N_{of}\times N_{if}\times N_{ky}\times N_{kx}\times N_{oy}\times N_{ox}$. *WST* consumes *nCycles*×*nPEs* PE's operations to complete one layer of convolution, but the effective number of PE's operations is equal to *nMACs*. Thus, we can define the utilization of PEs in *WST* by dividing the number of effective PE's operations to the total number of operations.

$$\text{Util. PEs} = \frac{\text{nMACs}}{\text{nCycles}\times\text{nPEs}} = \frac{\text{Noy}\times\text{Nox}}{\text{Niy}\times\text{Nix}}, \qquad (5)$$

where $N_{oy}\times N_{ox}$ and $N_{iy}\times N_{ix}$ are the sizes of output and input feature map respectively. Therefore, *WST* achieves low resource utilization for convolutional layers where the input size is bigger than its output size, such as *S-CONV* and *W-CONV*.

3) Strided-convolution eliminates data sharing and Zero-inserting increases computational burden in OST

Although *OST* could efficiently process traditional convolutional layers (Fig. 7(a)), there are still some problems when it handles the core operations in GAN shown in Fig. 6. We depict dataflows of *S-CONV* in Fig. 7(b). Each output neuron resides in one PE's output register and the kernel weight is imported in a sequential order and shared by all PEs. The input neurons required by the PEs are shown in the

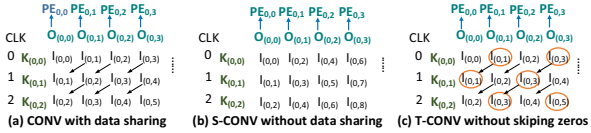


Figure 7. Dataflows of *OST*

corresponding columns. In this dataflow, PEs have totally different input neurons among the adjacent cycles, such as cycle 0 and cycle 1. Obviously, the temporal sharing of input neurons in *OST* disappears when exploited by *S-CONV*.

Due to the lack of pooling layers in DCGAN, to implement up-sampling, *T-CONV* needs to insert many zeros into its input feature maps. Fig. 6(b) illustrates zero-inserting in the input feature maps of *T-CONV*. Specifically, one zero needs to be inserted between every two input neurons. Since the input feature maps of *W-CONV* in backward $\overleftarrow{G_w}$ are from its forward computation $\vec{G}$ (*T-CONV*), there are also a lot of zeros in its input feature maps shown in Fig. 6(d). For *W-CONV* in $\overleftarrow{D_w}$, instead of zero-inserting in the input feature maps, we need to insert one zero between every two kernel weights before performing convolutional operations, as shown in Fig. 6(c).

Zero-inserting not only consumes extra memory but also significantly increases the computational burden. These inserted zeros result in a huge amount of zero-operand multiplications. Since the zero-operand multiplications do not contribute to the results, we name them ineffectual operations. These ineffectual operations account for about 64% and 75% of total multiplications in $\vec{G}$ / $\overleftarrow{G_w}$ and $\overleftarrow{D_w}$ respectively. However, *OST* cannot skip over inserted zeros in the input neurons. As shown in Fig. 7(c), although there still exists the temporal sharing, the inserted zeros (marked in colored circles) in the input feature maps are not skipped.

To achieve high processing efficiency, we should enable GAN accelerator to eliminate zero-operand multiplications in both input and kernel data.

IV. Our Proposed Design

In this section, we overcome three above-mentioned challenges with an algorithm and architecture co-design approach. First, we reduce the memory consumption by delaying the synchronization to the end of mini-batch processing. We then propose a time-multiplexed design to maximize resource utilization. Next, we introduce our optimized microarchitecture and provide dataflow for each computing phase.

A. Deferred Synchronization that Reduces Memory Consumption

As discussed in Section III-A, the synchronization operation exists in loss calculation. Since the loss function is used to calculate the error in the output layer (i.e., L^{th} layer) and the error of L^{th} layer (δ^L) is equal to the loss function's partial derivatives to the output data in L^{th} layer, we can calculate the error of L^{th} layer for real data x_i ($\delta^L_{x_i}$) using (6).

$$\delta^{\mathrm{L}}_{\mathrm{x_i}} = \frac{\partial \mathrm{loss_{dis.}}}{\partial \mathrm{D(x_i)}} = -\frac{1}{\mathrm{m}}\frac{\partial(\sum_{\mathrm{j=1}}^{\mathrm{m}}\mathrm{D(x_j)})}{\partial \mathrm{D(x_i)}} = -\frac{1}{\mathrm{m}}\frac{\partial \mathrm{D(x_i)}}{\partial \mathrm{D(x_i)}}, \qquad (6)$$

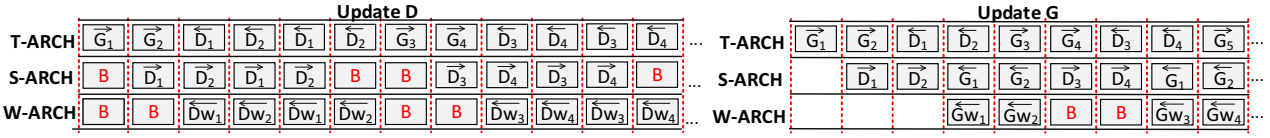


Figure 9. Pipeline based Design (B: Pipeline Bubble)

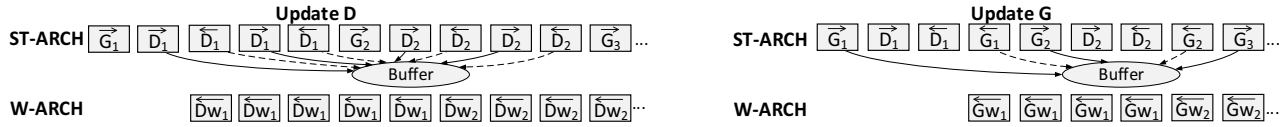


Figure 10. Time Multiplexed Design

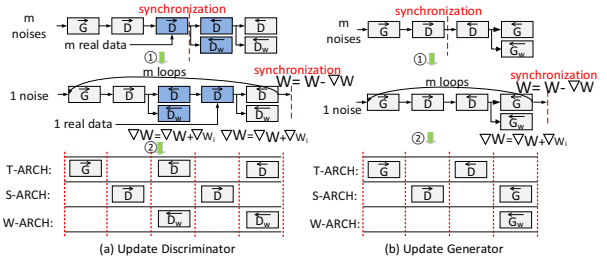


Figure 8. The Redesigned Training by Leveraging Deferred Synchronization (①) and Time-multiplexing (②)

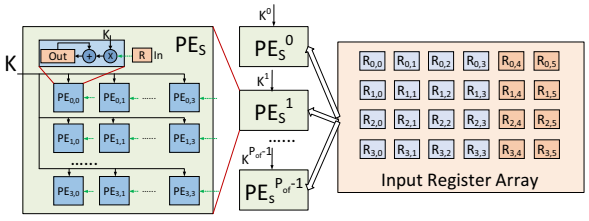


Figure 11. Zero Free and Output STationary Microarchitecture for $\vec{D}$, $\overleftarrow{D}$, $\overleftarrow{G}$ and $\vec{G}$ (*ZFOST*)

where the loss function ($loss_{dis.}$) is derived from (1). Thanks to the linear averaging in the calculation of loss function, for a real data x_i, its error of L^{th} layer only relates to its output ($D(x_i)$), as shown in (6). Similarly, the error of generated data ($\tilde{x}_i$) in the output layer is only related to its output ($D(\tilde{x}_i)$). Therefore, the error calculation does not need to wait for all the samples to finish their forward computations and the error back-propagation can start as soon as a sample finishes its forward computation. Due to this independence (the error of one input is only related to its own output), the mini-batch processing in Fig. 8 can be divided into m independent loops (batch size is m). During each loop, only one generated data and one real data calculates its own ∇w_i and the ∇w_i accumulates in ∇W. At the end of the mini-batch processing, we obtain the final ∇W and use it to update the parameters of Discriminator. Note that our optimization does not remove the synchronization operation in the weights updating. Instead, we delay the synchronization operation (averaging all the loss functions) to the end of batch processing by averaging all ∇w_i. Therefore, our optimization has the same effect as the original training algorithm.

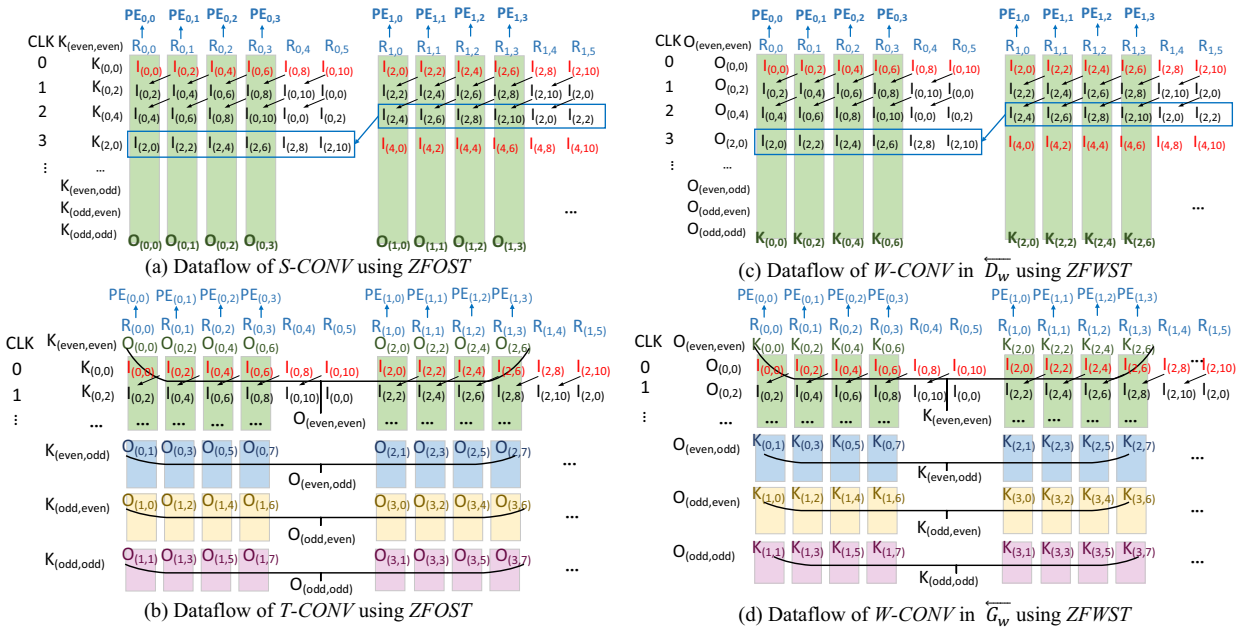
Based on the discussion above, once one data (real data or generated data) completes its forward processing, its backward computation can be performed immediately. In this way, the number of buffered intermediate data decreases to 1 and can be easily stored in the on-chip buffer. Similarly, we can also apply the deferred synchronization to the procedure of Generator update in Fig. 8(b).

B. Time Multiplexed Design that Fits All Computing Phases without Idleness

As discussed in Section III-B, time multiplexing of a uniform computing architecture is an effective method to reduce idleness in computing resource. In this section, we discuss how to efficiently map different computation phases to the shared architecture.

As shown in Fig. 1, Generator has an inverse architecture of Discriminator. Thus, backward $\overleftarrow{D}$ has the similar computation pattern as forward $\vec{G}$ and they can share a single computing architecture. Since the convolutional pattern for $\vec{G}$ and $\overleftarrow{D}$ is *T-CONV*, we name the architecture *T-ARCH*. Similarly, forward $\vec{D}$ and backward $\overleftarrow{G}$, whose core operation is *S-CONV*, can utilize the same computing architecture *S-ARCH*. For backward $\overleftarrow{D_w}$ and $\overleftarrow{G_w}$, where there is no accumulation operation (their convolutional pattern is *W-CONV*), a specific computing architecture *W-ARCH* is designed for them. As shown in Fig. 8, we can implement all the computing phases with these three computing architectures: *T-ARCH*, *S-ARCH* and *W-ARCH*. We can further improve utilization of computing architectures with pipeline-based design outlined in Fig. 9. However, there will be many pipeline bubbles (*B*) in this simple design. First, as shown in Fig. 8(a), because *S-ARCH* runs less frequently than *T-ARCH* when updating Discriminator, there would be bubbles in *S-ARCH*. Second, the utilization of *W-ARCH* is low (66.7% when updating Discriminator and 50% when updating Generator) as the calculation of ∇w_i is less frequently performed.

To eliminate the pipeline bubbles, a more uniform computing architecture should be proposed. Compared with four-dimension convolution *W-CONV*, *S-CONV* and *T-CONV* share more common features and it is possible to find a uniform computing architecture for them. Therefore, we combine *S-ARCH* and *T-ARCH* as *ST-ARCH* and have it time-multiplexed by *S-CONV* and *T-CONV*. In this way, the first two rows in Fig. 9 become one row in Fig. 10 and bubbles disappear. To improve utilization of *W-ARCH*, we can slow it down by decreasing its computing resource. To do so, not only do we buffer the intermediate data from the forward



(a) Dataflow of *S-CONV* using *ZFOST*

(c) Dataflow of *W-CONV* in $\overleftarrow{D_w}$ using *ZFWST*

(b) Dataflow of *T-CONV* using *ZFOST*

(d) Dataflow of *W-CONV* in $\overleftarrow{G_w}$ using *ZFWST*

Figure 12. Dataflows of *ZFOST* (PEs in ZFOST have fixed output neurons and share one kernel weight in each cycle) and *ZFWST* (PEs in ZFWST have fixed kernel weights and contribute to one output neuron in each cycle)

computation (solid line in Fig. 10), but also we store the errors generated in the backward computation (dashed line in Fig. 10) so that *W-ARCH* does not necessarily finish its processing within one pipeline cycle. By carefully choosing the buffer size, we can ensure that *W-ARCH* is fully utilized as shown in Fig. 10. We will elaborate the detailed implementation in Section V.

C. Computing Architectures and Dataflows that Maximize Data Reuse while Skipping Zeros

1) Zero Free and Output STationary design (ZFOST) for ST-ARCH

First, we propose our microarchitecture design *ZFOST* for *ST-ARCH* in Fig. 11. It can skip zeros in input without sacrifice of data reuse.

Our *ZFOST* mainly takes the output stationary optimization (unrolls output neurons and fix them to certain PEs) to have the input neurons temporally reused among adjacent PEs. Specifically, we utilize an input register array to store the input neurons so that the temporal reuse can be realized by just shifting the content in the registers. Since the minimum size of output feature map in GAN is smaller than that of traditional CNN, such as 4×4 in Fig. 1, only unrolling output neurons in one output feature map (Loop-2) cannot provide sufficient parallelism. Thus, we further unroll P_{of} output feature maps to increase the throughput. This unrolling strategy can also benefit spatial sharing of input neurons by broadcasting the input register array to different PE channels. In Fig. 11, all the unrolled PEs share a register array. The blue registers are directly connected to the corresponding PEs and serve as input operands for the PEs. For example, $R_{0,0}$ and $R_{0,1}$ are connected to $PE_{0,0}$ and $PE_{0,1}$ respectively.

To skip the zero-operand multiplications without sacrifice of data reuse, we develop following dataflows in Fig. 12. Fig. 12(a) indicates the dataflow when *S-CONV* is processed by *ZFOST*. In every cycle, only one kernel weight (K) is fed and spatially shared by all PEs. Each PE contributes to one fixed output neuron in $N_{kx} \times N_{ky}$ cycles. For example, $PE_{0,0}$ is dedicated to the calculation of the output $O_{(0,0)}$ and the result is accumulated in $O_{(0,0)}$.

Traditionally, kernel weights are fed in the sequential order: $K_{(0,0)}$, $K_{(0,1)}$, $K_{(0,2)}$ and so on. Unfortunately, as discussed in Section III-C3, this dataflow does not support temporal sharing of input neurons among PEs for *S-CONV*. Therefore, we reorder the sequence that kernel weights enter the PEs. That is, kernel weights with even horizontal index and even vertical index ($K_{(even,even)}$) enter first, then follow those with even horizontal index and odd vertical index ($K_{(even,odd)}$) and finally come $K_{(odd,even)}$ and $K_{(odd,odd)}$. Since each PE has its own output neuron ($O_{(ox,oy)}$), we can derive its demanded input neuron ($I_{(kx+2ox,ky+2oy)}$). Based on this relationship, we can figure out all the demanded input neurons in Fig. 12(a). Initially, the input neurons (marked as red color) are loaded into the register array from on-chip buffer. Then the input neurons can be temporally reused among the adjacent PEs by just circularly shifting the registers (as shown by the arrows).

Next, we discuss the dataflow of *T-CONV* using *ZFOST* in Fig. 12(b). To reduce the overhead of reordering kernel weights, we still utilize the same sequence of kernel weights. For *T-CONV*, only input neurons with even horizontal index and even vertical index ($I_{(even,even)}$) are not zeros (shown in Fig. 6(b)). To skip the zero-operand multiplications in *T-CONV*, we only need to load $I_{(even,even)}$ into the input register array. Based on the loop body of *T-CONV* in Fig. 4, the index of input $I_{(kx+ox,ky+oy)}$ is the sum of its output index $O_{(ox,oy)}$ and kernel index $K_{(kx,ky)}$. Since the index of non-zero input neurons are all even, the sum of its output index and kernel index should also be even. That means when the kernel weight is

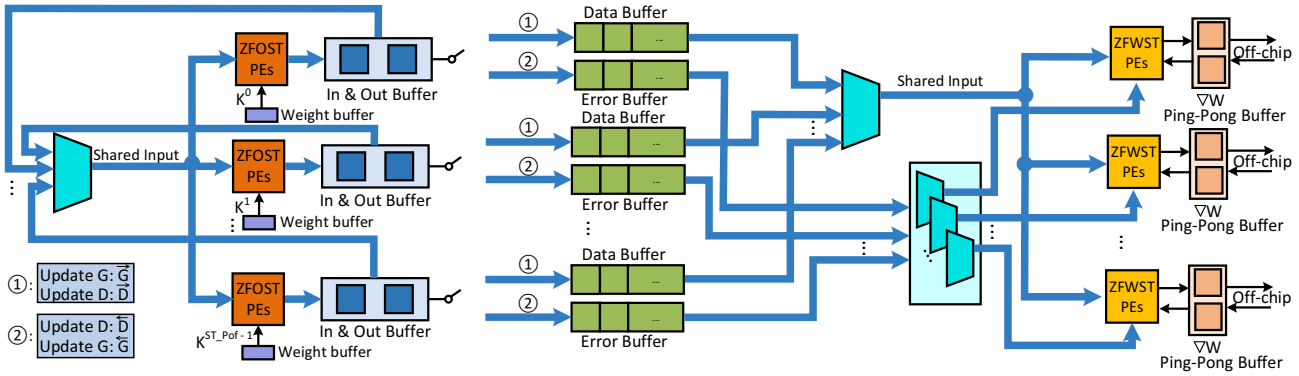


Figure 14. GAN Accelerator Design

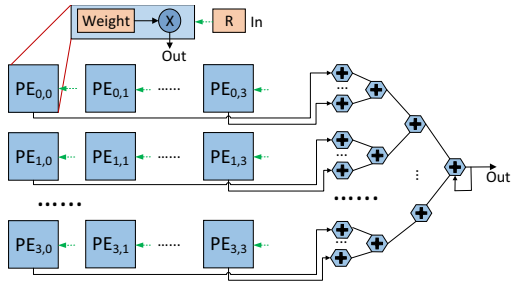


Figure 13. Zero Free and Weight STationary Microarchitecture for $\overleftarrow{D_w}$ and $\overleftarrow{G_w}$ (*ZFWST*)

$K_{(even,even)}$, only the operations calculating $O_{(even,even)}$ are effective. Similarly, when kernel weights are $K_{(even,odd)}$, $K_{(odd,even)}$ and $K_{(odd,odd)}$ respectively, their corresponding output neurons are $O_{(even,odd)}$, $O_{(odd,even)}$ and $O_{(odd,odd)}$, shown in Fig. 12(b). After pruning the ineffectual computations, we can calculate *4X* output neurons within the same time. Besides, we remain output neurons stationary in the PEs and the input neuron is still temporally reused by adjacent PEs.

2) Zero Free and Weight STationary design (ZFWST) for W-ARCH

W-ARCH is responsible for the processing in $\overleftarrow{D_w}$ and $\overleftarrow{G_w}$. In Fig. 6(c) and Fig. 6(d), zeros are inserted into the kernel of $\overleftarrow{D_w}$ and input data of $\overleftarrow{G_w}$. Thus, not only should *W-ARCH* skip the zeros in input data, but also in kernel weights.

Fig. 13 illustrates our microarchitecture design (*ZFWST*) for *W-ARCH* by unrolling Loop-3 (the kernel weights), where all the PEs contribute to one output neuron using the adder tree. Each PE has its own stationary kernel weight. This is the reason why we named our architecture as *ZFWST*. Note that PEs in *WST* (Fig. 5(b)) have the same input neurons, while PEs in *ZFWST* have various input neurons.

To temporally reuse input neurons among adjacent PEs, *ZFWST* takes the similar input register array as *ZFOST*. Thus, *ZFWST* also has the similar dataflows as *ZFOST* (same input neurons). Based on the facts that: (1) output neurons of *ZFOST* are the kernel weights of *ZFWST* and vice versa; (2) PEs in *ZFOST* have fixed output neurons and share one kernel weight in each cycle while PEs in *ZFWST* have fixed kernel weights and contribute to one output neuron in each cycle, we can conclude that *ZFWST* and *ZFOST* are somehow asymmetric in terms of kernel weights and output neurons. Therefore, we can easily derive the dataflows of *ZFWST* in Figs. 12(c) and (d) by switching kernel weights and output neurons in the dataflows of *ZFOST*.

When processing *W-CONV* in $\overleftarrow{D_w}$, the dataflow is similar to that of *S-CONV*. To skip the zeros in *W-CONV*'s kernel weights, we only allocate non-zero kernel weights to PEs. For *W-CONV* in $\overleftarrow{G_w}$, where zeros are inserted into the input neurons, we only load non-zero input neurons into the input register array. Since all the input neurons and kernel weights needed by $\overleftarrow{D_w}$ and $\overleftarrow{G_w}$ have been stored into the on-chip buffer by *ST-ARCH*, we can easily compute their addresses and load the corresponding non-zero data from the on-chip buffer. To increase parallelism, we further unroll its output feature maps as we did in *ZFOST*. And this unrolling can also support the spatial sharing of input register array among different PE channels.

V. Implementation

The proposed GAN accelerator is demonstrated by implementing DCGAN in Fig. 14. We utilize Xilinx VCU118 Evaluation board that includes two sets of five 512MB DDR4 SDRAM and a Xilinx UltraScale+ XCVU9P FPGA, which consists of 2586K Logic Cells, 6840 DSP Slices, and 75.9 Mb Block RAMs.

A. The Number of PEs in ZFWST and ZFOST

In Fig. 14, *ZFWST* is responsible for computing phases $\overleftarrow{D_w}$ and $\overleftarrow{G_w}$. All the other computing phases are processed by *ZFOST*. As shown in Fig. 11, each PE in *ZFOST* has one dedicated output neuron. Therefore, the number of PEs in each PE channel is equal to the number of unrolled output neurons. To achieve the maximum resource utilization, we set the number of PEs in *ZFOST* as the minimum output feature map of DCGAN (4×4). Similarly, for *ZFWST* in Fig. 13, as each PE has its own kernel weight and the minimum kernel size in $\overleftarrow{D_w}$ and $\overleftarrow{G_w}$ is 4×4, *ZFWST* also has 4×4 PEs.

B. On-chip Buffer Design

As shown in Fig. 14, four kinds of buffers are constructed in our design. Next, we discuss these buffers in detail.

1) In&Out buffer

Initially, the first layer's input data is loaded from off-chip memory into one of the *In&Out* buffers next to *ZFOST* PEs

TABLE III. RESOURCE UTILIZATION

Resource type	Total number of usage	Total resource on board
Logic utilization (in LUTS)	254523	1182240
Flip-Flops	79668	2364480
Block RAM	2008	2160
DSP	1694	6840

shown in Fig. 14. Then *ZFOST* starts to perform the processing and stores the results to the other *In&Out* buffer. After completing one layer's processing, the input and output buffers are switched and data in the previous output buffer becomes the input of the next layer. Then, *ZFOST* continues to process the next layer. To store the output of each layer, the size of buffers in *In&Out* should be equal to the maximum size of outputs among all the layers.

2) Data&Error buffer

Since $\overleftarrow{D_w}$ and $\overleftarrow{G_w}$ need the intermediate data from its forward computation (data d) and backward computation (error δ) to calculate the loss function's partial derivatives to weights (weight update ∇W), we create Data buffer and Error buffer to store these intermediate data respectively. When updating Discriminator, the intermediate data of $\vec{D}$ is stored into Data buffer, while $\overleftarrow{D}$ buffers its data into Error buffer. Similarly, $\vec{G}$ and $\overleftarrow{G}$ store their intermediate data to Data and Error buffers respectively when training Generator.

3) ∇W &Weight buffer

As discussed in Section IV-C2, we take *ZFWST* to perform the calculation of ∇W. To maximize data reuse, the kernel weights keep stationary in our *ZFWST* until its related output neurons have been calculated. However, when kernel size (N_{kx}×N_{ky}) is bigger than the size of unrolled kernel weights (P_{kx}×P_{ky}), *ZFWST* will generate partial results for ∇W. To store these partial results, we design ∇W buffer shown in Fig. 14. Note that the buffer is a ping-pong buffer so that *ZFWST* always keeps busy.

For *ZFOST*, its weights need to be loaded from off-chip memory and these weights will be reused multiple times after some cycles. To get rid of loading the same weight from off-chip memory repeatedly, we design Weight buffer and these weights will be stored in Weight buffer until their related output neurons have been calculated. In this way, for each weight, only one off-chip data access is demanded, which greatly reduces the requirement of off-chip bandwidth.

C. Unrolling Strategy (ST_P_{of} and W_P_{of})

To improve the processing throughput, we further unroll Loop-1 to process the different output feature maps in parallel. For *ZFOST*, the parallelism is ST_P_{of}, while the parallelism of *ZFWST* is W_P_{of}. Therefore, there are ST_P_{of} *ZFOSTs* and W_P_{of} *ZFWSTs* in Fig. 14.

1) Off-chip bandwidth determines W-P_{of}

Since not all the data can be stored on-chip, off-chip accesses are needed and the bandwidth of off-chip memory is a key factor that determines how fast our design can perform.

There are three different data accesses to/from off-chip memory in our design. The first one is in the initial phase where input data is loaded from off-chip memory. Since only the first layer needs to load the input, the loading frequency is very low and the bandwidth of this kind of off-chip access is not the bottleneck. The second one results from weights loading of *ZFOST*. Thanks to the Weight buffer in Fig. 14, the kernel weights can be fully reused before being removed from on-chip. Therefore, its off-chip bandwidth requirement is moderate. The last one is in ∇W calculating of *ZFWST* and has the maximum off-chip bandwidth requirement. Since *ZFWST* performs one ∇W calculating (which needs one read and one write) during $(N_{kx} \times N_{ky})/(P_{kx} \times P_{ky})$ cycles, its demanded off-chip bandwidth is $2 \times Frequency \times W\_P_{of}/((N_{kx} \times N_{ky})/(P_{kx} \times P_{ky}))$. The highest bandwidth requirement for *ZFWST* is from the layer with the minimum kernel size ($P_{kx} \times P_{ky}$) and it is equal to ($2 \times Frequency \times W\_P_{of}$). Therefore, we can derive the parallelism of *ZFWST* (W_P_{of}) based on the maximum off-chip bandwidth and the clock frequency of PE:

$$W\_P_{of} = \frac{Off\text{-}chip\ bandwidth}{2 \times Frequency \times bits\ per\ data} \quad (7)$$

2) The relation between speeds of W-ARCH and ST-ARCH determines ST_P_{of}

In Fig. 8, as Discriminator updating consists of two processes running on *W-ARCH* and five processes executed by *ST-ARCH*, the speed of *W-ARCH* (*ZFWST*) should be 2/5 of *ST-ARCH* (*ZFOST*) so that *ZFWST* keeps busy all the time. For Generator updating, which is comprised of one processing in *W-ARCH* and four processings in *ST-ARCH*, the speed of *ZFWST* is 1/4 of *ZFOST*. To minimize the buffer size, *ZFWST* should choose the faster speed (2/5 speed of *ZFOST*). As *ZFOST* and *ZFWST* have the same number of PEs and all the computing phases have the equivalent amount of computations, ST_P_{of} is 2.5X than W_P_{of} so that the processing speed of *ZFWST* is 2/5 of *ZFOST*.

$$ST\_P_{of} = 2.5 \times W\_P_{of} \quad (8)$$

For example, the width of data is 16 in our system and the bandwidth of off-chip memory is 192Gbps. The clock frequency of PE is 200MHz. Therefore, W_P_{of} is 30 and ST_P_{of} is 75. The total resource utilization of our design is shown in TABLE III.

TABLE IV. PARAMETERS OF GANS

GAN	Input	Kernel	Stride	Output
MNIST-GAN	1×28×28	5×5	2×2	64×14×14
	64×14×14	5×5	2×2	128×7×7
cGAN	3×64×64	4×4	2×2	64×32×32
	64×32×32	4×4	2×2	128×16×16
	128×16×16	4×4	2×2	256×8×8
	256×8×8	4×4	2×2	512×4×4

VI. EVALUATION

In this section, we first evaluate our microarchitecture design on four different computing phases by comparing it with the traditional computing architectures. Next, we illustrate the overall benefits of our design, including the improvements of deferred synchronization mechanism and time multiplexed design. We also demonstrate the performance variation with increasing PEs. All the designs

TABLE V. UNROLLING STRATEGY

Arch.	*ST-ARCH* ($\vec{G}/\overleftarrow{D}$, $\vec{D}/\overleftarrow{G}$) (*nPEs*=1200)	*W-ARCH* ($\overleftarrow{D_w}$, $\overleftarrow{G_w}$) (*nPEs*=480)
NLR	P_{if}=16, P_{of}=75	P_{if}= 16, P_{of}=30
WST	P_{kx}=5,P_{ky}=5, P_{of}=48	P_{kx}=4,P_{ky}=4, P_{of}=30
OST	P_{ox}=4,P_{oy}=4,P_{of}=75	P_{ox}=5,P_{oy}=5,P_{of}=19
ZFOST	P_{ox}=4,P_{oy}=4,P_{of}=75	$\overleftarrow{D_w}$:P_{ox}=5,P_{oy}=5,P_{of}=19 $\overleftarrow{G_w}$: P_{ox}=3,P_{oy}=3,P_{of}=53
ZFWST	$\vec{G}/\overleftarrow{G}$:$P_{kx}$=5,$P_{ky}$=5, P_{of}=48 $\vec{G}/\overleftarrow{D}$:P_{kx}=3,P_{ky}=3, P_{of}=133	P_{kx}=4,P_{ky}=4, P_{of}=30

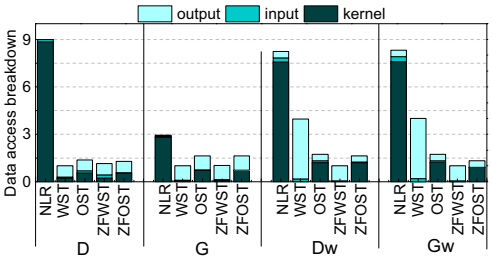


Figure 16. On-chip Data Accesses Breakdown for DCGAN

are implemented on Xilinx VCU118 board. Finally, we compare our accelerator with CPU and GPU based GAN implementations. The baseline CPU we used is a 6-core Intel i7-6850K. The GPUs we used are NVIDIA K20 and NVIDIA Titan X.

Besides the DCGAN network in Fig. 1, we further include GAN trained using MNIST dataset (MNIST-GAN) [24] and cGAN used in Context Encoders [7] and Image editing [25] to perform the comprehensive evaluation. Since Generator has an inverse architecture of Discriminator, TABLE IV only lists the parameters of Discriminator for these two GAN networks.

A. Comparison of Different Computing Phases

As discussed in Section IV-B, GAN has four computing phases: $\vec{D}/\vec{G}$, $\vec{G}/\overleftarrow{D}$, $\overleftarrow{D_w}$ and $\overleftarrow{G_w}$. In this section, we compare our microarchitectures (*ZFOST* and *ZFWST*) with the traditional computing architectures on these four phases. When demonstrating the differences between these, we strive to avoid any unfair advantage for our architectures. First, we unroll the output feature maps (P_{of}) in *NLR*, *WST* and *OST* to ensure these architectures have the same number of PEs. Second, we apply different unrolling strategies to different architectures to guarantee the lowest idleness when performing each computing phase, shown in TABLE V. Third, we optimize the dataflow of *NLR* so that it can skip over zeros in its input data and kernel weights. Finally, to reduce the off-chip data accesses, all the architectures have the same on-chip buffer design.

Fig. 15 demonstrates the performance (processing throughput) comparison on the above four computing phases and all the performance results are normalized to the improved *NLR*. To select the best one from architectures with the similar performance, we also compare the number of on-chip data accesses in Fig. 16, which includes loading kernel weights and input neurons and reading/writing output neurons.

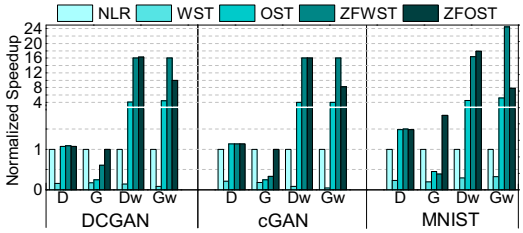


Figure 15. Performance Comparison on four Computing Phases (D: $\vec{D}/\overleftarrow{G}$, G: $\vec{G}/\overleftarrow{D}$, Dw: $\overleftarrow{D_w}$ and Gw: $\overleftarrow{G_w}$)

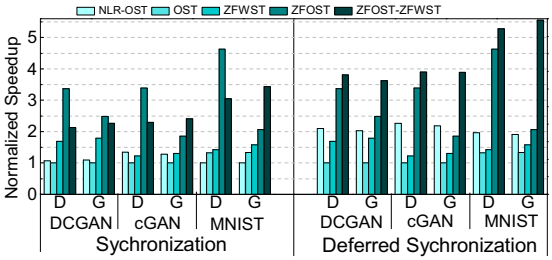


Figure 17. Overall Performance Comparison among various Computing Architectures (D: Discriminator , G: Generator)

Thanks to the zero-free design, *ZFOST* and *ZFWST* yield the optimal performance among all the phases. In Fig. 15, *ZFOST* has a better performance on $\vec{G}$, while $\overleftarrow{G_w}$ runs faster on *ZFWST*. For $\vec{G}$, though *NLR* and *ZFOST* achieve the same speedup, *ZFOST* has the lower on-chip data accesses shown in Fig. 16. For $\overleftarrow{G_w}$, *ZFWST* achieves the maximum speedup with the minimum on-chip data access. Therefore, *ZFOST* has an advantage to perform computing phases $\vec{G}/\overleftarrow{D}$ and $\vec{D}/\overleftarrow{G}$, while *ZFWST* is good at the computations of $\overleftarrow{D_w}$ and $\overleftarrow{G_w}$, which validates our designs for *ST-ARCH* and *W-ARCH*.

B. Overall Benefits of Our Design (ZFOST-ZFWST) with Deferred Synchronization and Time-multiplexed Architecture

In this section, we demonstrate the overall benefit of our design when performing the update of Discriminator and Generator. Our design consists of *ZFOST* and *ZFWST*, where *ZFOST* is responsible for $\vec{G}/\overleftarrow{D}/\vec{D}/\overleftarrow{G}$ and *ZFWST* performs $\overleftarrow{G_w}/\overleftarrow{D_w}$. To demonstrate the advantage of the combinational design, we first select three top architectures (*OST*, *ZFWST* and *ZFOST*) in Fig. 15 to perform the whole computing phases. We also choose the best combination (*NLR-OST*) from the traditional architectures to perform $\vec{G}/\overleftarrow{D}/\vec{D}/\overleftarrow{G}$ and $\overleftarrow{G_w}/\overleftarrow{D_w}$ respectively. All the designs have the same number of PEs (1680).

Under the synchronization in Fig. 17, the unique architecture *ZFOST* outperforms our combinational architecture (*ZFOST-ZFWST*) since the synchronization leads to only one architecture (*ZFOST* or *ZFWST*) in our combinational architecture working at each time. With the optimization of deferred synchronization, two architectures in the combinational design can execute in parallel and the performance of combinational architecture (*NLR-OST* and

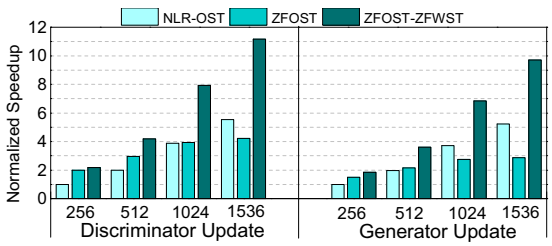


Figure 18. Performance Variation with Various PEs

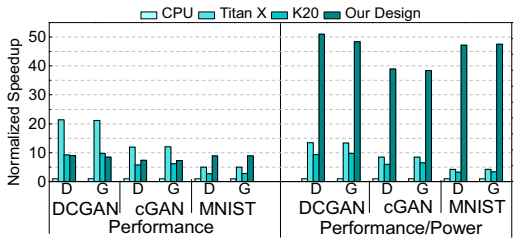


Figure 19. Comparison with CPU and GPU

ZFOST-ZFWST) improves significantly, while the performance of unique architecture remains the same. Among the combinational architectures, *ZFOST-ZFWST* outperforms *NLR-OST* due to its zero-free optimization in all the computing phases. Therefore, *ZFOTS-ZFWST* architecture achieves the best speedup (average 4.3X) in Fig. 17.

We select the top three implementations: *NLR-OST*, *ZFOST* and *ZFOST-ZFWST* in Fig. 17 to further explore the performance variation when PE numbers vary. In Fig. 18, our *ZFOST-ZFWST* always has the best performance across the designs with various PEs. Moreover, by leveraging 512 PEs, *ZFOST-ZFWST* achieves similar performance as *NLR-OST* and *ZFOST* with 1024 PEs.

C. Comparison with CPU and GPU implementations

We conduct a comparison with CPU and GPU based GAN implementations. We leverage Caffe [26] to implement CPU and GPU acceleration by crafting the GAN networks with convolutional layers and deconvolutional layers. To compare CPU/GPU (using floating point) and FPGA (using fixed point), we use Giga operations per second (GOPS) as the performance metric. We also measure the power consumption using WattsUp [27]. In Fig. 19, our accelerator has an average of 8.3X speedup and 45.2X energy-efficiency over CPU. It also achieves an average of 5.2X energy-efficiency over Titan X and 7.1X energy-efficiency over K20.

VII. Related Work

Traditionally, neural networks are accelerated by GPUs [28], [29]. Although GPUs can flexibly adapt to various workloads, the flexibility is achieved at a large fraction of transistors, significantly affecting the energy-efficiency and cost of executing specific workloads such as CNNs and GANs. To achieve scalability and efficiency in all dimensions, abundant application-specific accelerators have been proposed for various neural networks, with implementations on either FPGAs [19]–[21], [23], [30] or ASICs [16]–[18], [22], [31]–[33]. DianNao [17] and DaDianNao [18] utilize a large global buffer as a shared storage to reduce DRAM access energy consumption. However, there is no local data reuse, which requires a high bandwidth between the on-chip buffer and PEs and further results in low energy efficiency. Farabet et al. [20] propose a systolic architecture called NeuFlow architecture where kernel weight remains stationary in the register to maximize kernel reuse. However, there are a lot of idle PEs in NeuFlow when it is applied to *S-CONV* and *W-CONV*. Moreover, NeuFlow could not efficiently handle the zero-inserting in kernels. By eliminating all DRAM accesses for weight and exploiting a specific data access pattern (kernel weight broadcast and output neurons spatially reuse across the PE array), ShiDianNao [22] can map a CNN within an SRAM to greatly reduce the energy consumption. However, ShiDianNao can only accelerate a limited number of neural networks with small size and could not handle GAN with a large number of kernel weights. To achieve high energy efficiency, Eyeriss [32] proposes a row stationary dataflow by maximally reusing data locally. It can also gate zero input neuron computations to further save power. Instead of powering off the zero neuron computations, Cnvlutin [33] directly skips over the zero inputs. However, both of them could not handle the zero-inserting in the kernel for *W-CONV*. EIE [34] compresses the deep neural network by leveraging the sparsity of the input neurons and kernels. However, this compression only focuses on fully-connected layers, while DCGAN mainly consists of convolutional layers. PipeLayer [15] proposes a pipeline based accelerator for CNN training. This pipeline based design will result in pipeline bubbles in GAN training. Moreover, its memristor based processor could not eliminate the zero-operand multiplications in GAN.

VIII. Conclusion

This work presents the first holistic solution for accelerating the unsupervised deep learning. We demonstrate the challenges in GAN accelerator design: synchronization mechanism, a mass of computing phases and non-traditional convolutions by analyzing the procedure of GAN training. Then we overcome the challenges with an algorithm and architecture co-design approach. First, we defer the synchronization operation to the end of batch processing. We then propose a novel time-multiplexed design to efficiently map the abundant computing phases to our customized microarchitectures (*ZFOST* and *ZFWST*). Compared with traditional deep learning accelerators, our proposed design achieves the best performance (average 4.3X) under the same computing resource. Our design also has an average of 8.3X speedup over CPU and 6.2X energy-efficiency over NVIDIA GPU.

Acknowledgment

This work is supported in part by NSF grants 1527535, 1423090, 1320100,1117261, 0937869, 0916384, 0845721 (CAREER), 0834288, 0811611, 0720476, by SRC grants 2008-HJ-1798, 2007-RJ-1651G, by Microsoft Research Trustworthy Computing, Safe and Scalable Multicore Computing Awards, and by three IBM Faculty Awards.

REFERENCES

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In Advances in Neural Information Processing Systems (NIPS), 2012.

[2] Carl Doersch, Abhinav Gupta, and Alexei a Efros. Unsupervised Visual Representation Learning by Context Prediction. In IEEE International Conference on Computer Vision (ICCV), 2015.

[3] Mehdi Noroozi and Paolo Favaro. Unsupervised Learning of Visual Representations by Solving Jigsaw Puzzles. In European Conference on Computer Vision (ECCV), 2016.

[4] Mingcong Song, Kan Zhong, Jiaqi Zhang, Yang Hu, Duo Liu, Weigong Zhang, Jing Wang, and Tao Li. In-situ AI: Towards Autonomous and Incremental Deep Learning for IoT Systems. In 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA), 2018.

[5] Ian J Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative Adversarial Networks. In Advances in Neural Information Processing Systems (NIPS), 2014.

[6] Emily Denton, Sam Gross, and Rob Fergus. Semi-Supervised Learning with Context-Conditional Generative Adversarial Networks, arXiv Prepr. arXiv1611.06430, Nov. 2016.

[7] Pathak Deepak, Philipp Krahenbuhl, Jeff Donahue, Trevor Darrell, and Alexei A Efros. Context Encoders : Feature Learning by Inpainting. In Computer Vision and Pattern Recognition (CVPR), 2016.

[8] Xiaolong Wang and Abhinav Gupta. Unsupervised Learning of Visual Representations using Videos. In IEEE International Conference on Computer Vision (ICCV), 2015.

[9] Rise of the machines: Google AI experiment may lead to robots that can learn WITHOUT human input: http://www.dailymail.co.uk/sciencetech/article-4420804/Experiment-lead-machine-s-learning-without-humans.html.

[10] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. In International Conference on Learning Representations (ICLR), 2016.

[11] Michael Mathieu, Camille Couprie, and Yann LeCun. Deep multi-scale video prediction beyond mean square error. In International Conference on Learning Representations (ICLR), 2016.

[12] Andrzej Pronobis and Rajesh PN Rao. Learning Deep Generative Spatial Models for Mobile Robots, arXiv Prepr. arXiv1610.02627, Oct. 2016.

[13] Arna Ghosh, Biswarup Bhattacharya, and Somnath Basu Roy Chowdhury. SAD-GAN: Synthetic Autonomous Driving using Generative Adversarial Networks, arXiv Prepr. arXiv1611.08788, Nov. 2016.

[14] Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein GAN, arXiv Prepr. arXiv1701.07875, Jan. 2017.

[15] Linghao Song, Xuehai Qian, Hai Li, and Yiran Chen. PipeLayer : A Pipelined ReRAM-Based Accelerator for Deep Learning Basics of Deep Neural Network. In 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA), 2017.

[16] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R Stanley Williams, and Vivek Srikumar. ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars. In ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), 2016.

[17] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. DianNao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning. In 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2014.

[18] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. DaDianNao: A Machine-Learning Supercomputer. In ACM/IEEE 47th Annual International Symposium on Microarchitecture (MICRO), 2014.

[19] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), 2015.

[20] Clément Farabet, Berin Martini, Benoit Corda, Polina Akselrod, Eugenio Culurciello, and Yann LeCun. NeuFlow : A Runtime Reconfigurable Dataflow Processor for Vision. In IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops, 2011.

[21] Cyril Poulet and Yann Lecun. An FPGA-Based Stream Processor for Embedded Real-Time Vision with Convolutional Networks. In IEEE 12th International Conference on Computer Vision Workshops, 2009.

[22] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. ShiDianNao: Shifting Vision Processing Closer to the Sensor. In Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA), 2015.

[23] Yufei Ma, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. Optimizing Loop Operation and Dataflow in FPGA Acceleration of Deep Convolutional Neural Networks. In ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), 2017.

[24] MNIST-GAN: https://github.com/Newmu/dcgan_code/blob/master/mnist/train_cond_dcgan.py.

[25] Guim Perarnau, Joost Van De Weijer, Bogdan Raducanu, Jose M Álvarez, and Data Csiro. Invertible Conditional GANs for image editing. In Conference on Neural Information Processing Systems (NIPS), 2016.

[26] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe : Convolutional Architecture for Fast Feature Embedding. In 22nd ACM International Conference on Multimedia, 2014.

[27] Watts Up: https://www.wattsupmeters.com/.

[28] Mingcong Song, Yang Hu, Yunlong Xu, Chao Li, Huixiang Chen, Jingling Yuan, and Tao Li. Bridging the Semantic Gaps of GPU Acceleration for Scale-out CNN-based Big Data Processing: Think Big, See Small. In The 25th International Conference on Parallel Architectures and Compilation Techniques（PACT）, 2016.

[29] M Song, Y Hu, H Chen, and T Li. Towards Pervasive and User Satisfactory CNN across GPU Microarchitectures. In 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA), 2017.

[30] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. Fused-Layer CNN Accelerators. In ACM/IEEE 49th Annual International Symposium on Microarchitecture (MICRO), 2016.

[31] Wenyan Lu, Guihai Yan, Jiajun Li, Shijun Gong, Yinhe Han, and Xiaowei Li. FlexFlow : A Flexible Dataflow Accelerator Architecture for Convolutional Neural Networks. In 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA), 2017.

[32] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), 2016.

[33] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. Cnvlutin: Ineffectual-Neuron-Free Deep Convolutional Neural Network Computing. In ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), 2016.

[34] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark Horowitz, and Bill Dally. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), 2016.