MemGANs: Memory Management for Energy-Efficient Acceleration of Complex Computations in Hardware Architectures for Generative Adversarial Networks

Muhammad Abdullah Hanif[1,*], Muhammad Zuhaib Akbar[2,*], Rehan Ahmed[2], Semeen Rehman[1],
Axel Jantsch[1], Muhammad Shafique[1]
[1]*Technische Universität Wien (TU Wien), Vienna, Austria*
[2]*National University of Sciences and Technology (NUST), Islamabad, Pakistan*
{muhammad.hanif, semeen.rehman, axel.jantsch, muhammad.shafique}@tuwien.ac.at
{makbar.msee16seecs, rehan.ahmed}@seecs.edu.pk

***Abstract*—Generative Adversarial Networks (GANs) have gained importance because of their tremendous unsupervised learning capability and enormous applications in data generation, for example, text to image synthesis, synthetic medical data generation, video generation, and artwork generation. Hardware acceleration for GANs become challenging due to the intrinsic complex computational phases, which require efficient data management during the training and inference. In this work, we propose a distributed on-chip memory architecture, which aims at efficiently handling the data for complex computations involved in GANs, such as *strided convolution* or *transposed convolution*. We also propose a controller that improves the computational efficiency by pre-arranging the data from either the off-chip memory or the computational units before storing it in the on-chip memory. Our architectural enhancement supports to achieve 3.65x performance improvement in state-of-the-art.**

***Index Terms*—Generative Adversarial Networks, Hardware Accelerator, DNN, GAN, DCGAN, Memory Architecture**

I. Introduction

Training and inference are two primary stages of any Deep Neural Network (DNN), where the training stage tunes the network parameters, which are then utilized at the inference stage to infer the information from the test/actual data. Traditionally, *supervised learning* is used to train DNNs [1]. However, due to the requirement of extensive labeled data for supervised learning of neural networks, *semi-supervised* and *unsupervised learning* have gained a lot of interest [2]–[5]. One of the particular frameworks towards this is a **Generative Adversarial Networks (GANs)** [6]; see Figure 1.

In GANs, a *generative model* competes against a *discriminative model* (an adversary) which determines whether a sample generated by the *generator* belongs to the data distribution of the training samples or not [2]. These GANs are capable of generating highly meaningful data from the latent space and are the state-of-the-art for many applications, such as text to image synthesis [7], image classifications [8], mobile robots [9] and video prediction [10].

In contrast to the traditional DNNs, GANs involve relatively more complex computations such as *strided convolution*, *transposed convolution*, and *four-dimensional convolutions* [11]. Although traditional DNN hardware accelerators can be used to perform these computations, the computational patterns like *zero-inserting* (or zero-skipping) in transposed (or strided convolution) will result in reduced efficiency for these existing accelerators [11].

Specialized Hardware Accelerator for GANs: Recently, Song et al. [11] proposed a hardware accelerator which uses a time-multiplexed design and an efficient dataflow to perform all the different types of computational phases involved in GANs training and inference. The accelerator is composed of two different microarchitectures: (1) *Zero Free and Output STationary (ZFOST)*, which is responsible for forward data pass and backward error pass; and (2) *Zero Free and Weight STationary (ZFWST)*, which is responsible for weight update. The dataflow supported by these microarchitectures is quite similar. Therefore, for different computational phases, both architectures require data to be loaded in the similar fashion.

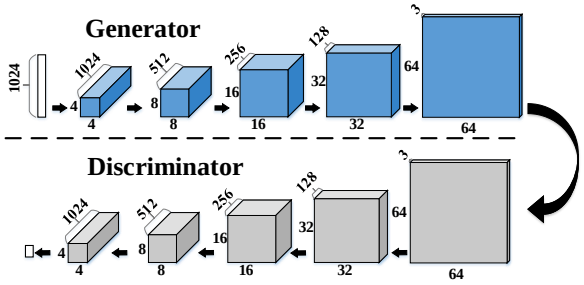

Fig. 1: Deep Convolutional Generative Adversarial Network

In this work, we focus on analyzing and accelerating the processing in the *ZFOST* microarchitecture [11] (see Figure 2 in Section II) by efficient data management, *the ZFOST is heavily used in most of the computing phases related to GAN training and all the phases related to inference.* [1]

The Memory Challenge in "ZFOST" of GAN Hardware: The *ZFOST* microarchitecture is mainly capable of accelerating two types of non-traditional convolutions, namely *strided-convolution* and *transposed-convolution*. To efficiently perform these convolutions, the weights are fed to the *ZFOST* microarchitecture in a *type-oriented format*, instead of a sequential form. Here, type-oriented refers to the alignment of data based on its row and column indexes being even or odd in its data class. That is, pixels or weights belonging to even rows and even columns are considered a part of *even-even* type and similarly others are placed in *even-odd*, *odd-even*, and *odd-odd* categories. This type-orientation format of weights requires input to the PEs in the ZFOST to be loaded in the same format. Moreover, the effective dataflow defined in [11] requires multiple data pixels (up to 24 pixels) to be loaded simultaneously (in one clock cycle). Since the off-chip memory contains input feature maps in linear form as illustrated in Figure 4(c), even with the architecture of [11], fulfilling the requirement of multiple pixels to be fetched from on-chip memory in one clock transition and the fetched data to be type-oriented is a challenge in terms of the design of the on-chip memory and data management. *Thus, a specialized on-chip memory architecture and a control unit are required for GAN Hardware*, which can arrange data according to its type (i.e. even-even, even-odd, odd-even and odd-odd) and can

*Muhammad Abdullah Hanif and Muhammad Zuhaib Akbar both are lead authors, and have equal contributions.

[1]Note: our proposed optimizations can also be used for accelerating the *ZFWST* microarchitecture, if required.

provide multiple data pixels to the local register whenever required to operate it at the full throttle.

Our Novel Contributions: To address the above challenges, we make the following novel contributions.

1) **MemGANs: A Distributed Memory Architecture for GANs:** It is composed of multiple single-port scratchpad memories (SPRAM)[2]. The memory space is divided so that it can store elements of input feature map based on the type of the pixel, i.e., *even-even*, *even-odd*, *odd-even* or *odd-odd*, and is capable of providing multiple data points in a single clock transition, as per the requirement of *ZFOST* for the *strided* and *transposed convolution*.
2) **Re-Packaging Controller for Efficient Data Arrangement:** It computes the required pixels' placement location in on-chip memory and thereby arranges the data in the proposed distributed on-chip memory architecture in a type-oriented format as per the requirements of the ZFOST microarchitecture for strided/transposed convolution.

Evaluation and Comparison with State-of-the-Art: We perform an extensive evaluation for the complete *ZFOST* microarchitecture comprising our proposed architectural enhancements. Compared to the state-of-the-art system in [11], our design achieves an improvement of (on average) 3.65x in the processing time for images of sizes 16x16 to 1024x1024.

II. Background and State-of-the-Art

A. Overview of Deep Convolutional GANs

One of the widely used types of GANs is Deep Convolutional Generative Adversarial Networks (DCGANs) [8]. An example DCGAN is shown in Figure 1. The DCGANs are composed of two networks: (1) Generator, and (2) Discriminator. During the training phase, both the networks are trained as a *two-player game* where both play with the objective to outperform the other. *The objective of the generator* is to generate samples from the latent space which cannot be detected by the discriminator. *The objective of the discriminator* is to accurately distinguish between the generated and the real data. One of the key differences in DCGANs is that the max-pooling layer in conventional CNNs is replaced with *strided convolution*, which is used in the forward-computation phases of the discriminator. The *strided convolution* operation skips output pixel computation as per a defined stride size (zero-skipping) which thereby corresponds to downsampling. Similarly, another key operation in DCGANs is *transposed convolution* (de-convolution), which is used in the forward-computation phases of the generator. In *transposed convolution*, the convolution operation needs to insert zeros in the input feature map (zero-inserting) and thereby corresponds to upsampling.

B. Hardware Accelerator for DCGAN [11]

An efficient dataflow and a hardware accelerator to handle complex computation in DCGAN are proposed in [11] using a **zero-free output stationary (ZFOST)** microarchitecture, as illustrated in Figure 2. The microarchitecture is composed of a 4x4 PE array and an input register array. The PE array is responsible for processing the output, and the input register array is for feeding the inputs neurons (i.e., the pixels of an input feature map) to the PEs. The registers shown with gray color in the register array are directly linked with corresponding PEs in the PE array, while the additional registers are to allow input data reuse of the fetched input data by shifting the content of the registers. The weights are spatially shared by the PEs and are fed one at a time. The size of the PE array is kept to be 4x4 to match the size

[2]Note: single-ported memories incur less area compared to the multi-ported designs, and being distributed allows for parallel access with the specialized design and access control

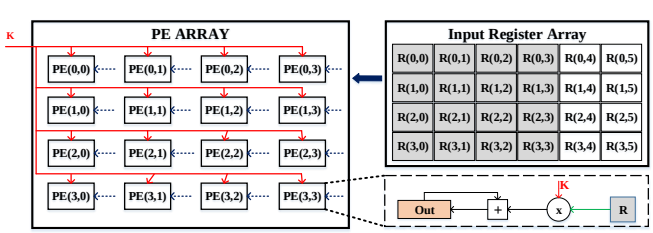

Fig. 2: Architecture of zero-free output stationary(ZFOST) Architecture

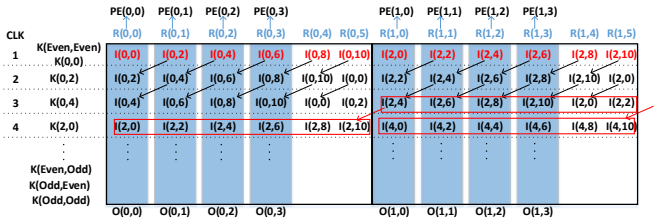

Fig. 3: Dataflow of strided convolution using ZFOST Architecture

of the minimum output feature map in DCGANs. For computations in this microarchitecture, the output neurons (i.e., the pixels of an output feature map) are unrolled, and the spatially neighboring neurons are mapped on the PE array where one output neuron is mapped to one PE and is kept there throughout its computation. The kernel weights are broadcasted one at a time and are spatially shared by the PEs in the PE array.

To avoid bubbles in the computation flow of the *strided* and *transposed convolution* performed by *ZFOST*, an efficient data flow has also been proposed in [11]. In this, the kernel/filter weights, instead of being fetched in sequential order, are loaded into *ZFOST* through input register array in a **type-oriented format** (i.e. *even-even*, *even-odd*, *odd-even* and *odd-odd*). As illustrated by Figure 3, first *even-even* weights (i.e., weights with even row and even column indexes) are processed followed by *even-odd* kernel weights then *odd-even* kernel weights, and at the end *odd-odd* kernel weights are processed by *ZFOST*.

Since each PE is linked to an output $O_{(ox,oy)}$, its required input for kernel weight $K_{(kx,ky)}$ can be computed as $I_{(kx+2ox,ky+2oy)}$. Initially, all the inputs marked with "Red" in Figure 3 are loaded into the input register array. In the next clock cycles data is shifted in input register array for temporal reuse. As an example, when the kernel weight $K_{(0,0)}$ is provided to all PEs at the first clock cycle for processing, Figure 4(a) highlights the required input pixels to be loaded into the input register array. The data is then shifted within the input register array for the next two clock transitions to perform the processing on kernel weights $K_{(0,2)}$ and $K_{(0,4)}$, as defined in the dataflow of Figure 3. For the fourth clock transition when the next *even-even* kernel weight $K_{(2,0)}$ is broadcasted to all PEs, six new data pixels are required in the last row of the input register array, as illustrated in Figure 4(b).

This requirement of dataflow, where multiple data points (up to 24 data points) need to be loaded in the input register array in a type-oriented format, brings challenges in fetching data from the on-chip memory. The key scientific challenges to achieve a highly optimized memory design and management are discussed next.

III. Scientific Challenges

1) **The dataflow requirement makes traditional on-chip buffer inefficient:** Data in the traditional on-chip memory is stored in a linear format which limits the design to provide multiple data points to *ZFOST* in one clock cycle. Figure 4c represents the structure of an on-chip memory which has the data width of 64 bits. Therefore, for a 16-*bit* fixed-point system, each location of the on-chip memory will store 4 data point in a cascaded form. Due to the linear arrangement

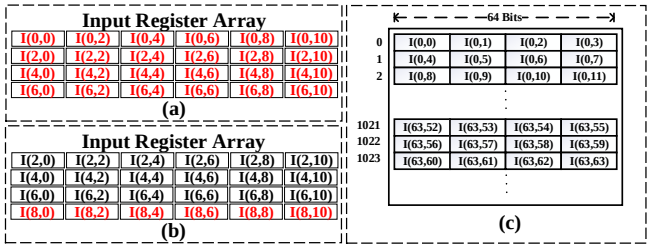

Fig. 4: Data to be loaded in local register of ZFOST a) during processing of K(0,0), b) during processing of K(2,0) and c) Linear data arrangement in on-chip/off-chip memory.

of pixels stored in on-chip memory, in order to load required 24 data point in the input register array of the *ZFOST* during the first clock iteration that have to be multiplied with the kernel weight $K_{(0,0)}$, we need 12 read cycles to extract the relevant data from the on-chip memory. Moreover, since only *even-even* data is required during this computation, half of the read data will be wasted. Therefore, to implement the data flow of Figure 3, *a specialized memory architecture is needed, which can provide multiple data points of input feature map from the on-chip memory in one clock cycle without any data wastage as per the requirements of strided and transposed convolution*.

2) **Requirement of type-orientation of input feature map:** Since the kernel weights are loaded in the *ZFOST* in *type-oriented form*, the corresponding input data must also be fetched based on their type as illustrated in Figure 4(a) and Figure 4(b). To support the specialized memory architecture that can provide multiple data points of input feature map stored in the on-chip memory in one clock cycle, *a controller is also needed in order to arrange the data prior to storing it in the on-chip memory on the basis of the type of the pixels*.

IV. MemGANs: Specialized Memory Architecture and Management for GANs Accelerators

Figure 5 provides an overview of the complete system consisting of our energy-efficient memory architecture for performing efficient processing of *strided and transposed convolutions* using the *ZFOST* microarchitecture. The proposed memory design can provide multiple required data points to the ZFOST microarchitecture in a single clock cycle. The memory architecture is supported with re-packaging units which arrange the data in a type-oriented format before storing it in the on-chip memory. This data can either be the inputs fetched from the off-chip memory (DRAM) or the outputs coming out of the *ZFOST* microarchitecture. The type-orientation removes the unuseful data-fetches by providing only a particular type of data required for processing, as per the dataflow described in Section II-B.

The on-chip memory is designed to have two IO-buffers such that, one is used to hold the inputs while, in the meantime, the other can be used to store the outputs. Once the processing of a layer is complete, the functionality of IO-buffers is switched for the cases where the output of one layer is input of the next. The main controller, shown in Figure 5, controls the overall processing. Initially, it gives a command to read the input feature map from the off-chip memory then it provides a command to the dataflow controller to start performing the convolution.

In the following subsections, details of our memory architecture, re-packaging unit and the other blocks shown in Figure 5 are presented. For the ease of readability, the variables used in the equations and the pseudo-codes in these subsections are described in Table I.

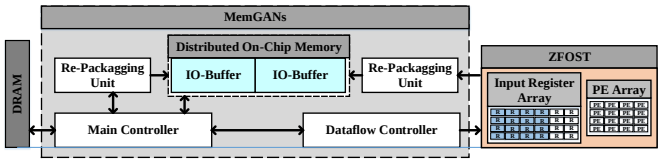

Fig. 5: Integration of our MemGANs architecture in GAN Hardware

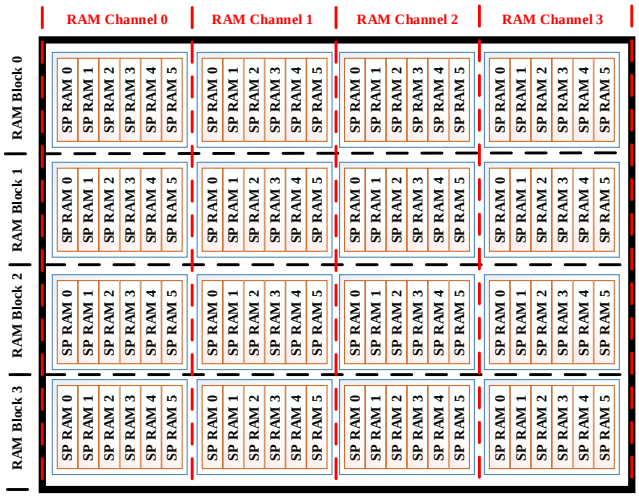

Fig. 6: Design of distributed on-chip IO-Buffer

A. IO-Buffer Design: Distributed On-chip Memory Architecture

The proposed on-chip memory design is composed of multiple small sized SRAMs linked together in a grid formation. The grid is divided based on two major axes named, *RAM-Block*, and *RAM-Channel* where each block of the grid is composed of six SPRAMs, as illustrated in Figure 6. Each memory slice stores data following the requirement of the dataflow for processing the *transposed/strided convolution*. There are four *RAM-Blocks* in the design, based on the number of types in which input data is divided. For example, input pixels with *even-even* indexes (i.e., $I_{(even,even)}$) are stored in *RAM-Block* 0 and, similarly, pixels with *even-odd*, *odd-even*, and *odd-odd* types are stored in *RAM-Blocks* 1, 2 and 3, respectively. Each *RAM-Block* is divided into four *RAM-Channels*, where each channel stores data required by each row of the input register array in the *ZFOST* micro-architecture. Each *RAM-Channel* is divided into six SPRAMs based on the number of columns in the input register array. Therefore, each SPRAM in a *RAM-Block* is responsible for feeding data to one register of the input register array when all the registers have to be loaded simultaneously. For example, for the first computation with $K_{(0,0)}$ kernel weight, 24 points have to be loaded to the input register array, as illustrated in Figure 4(a). Six data points for the first row of the input register array are available in address 0 of the six SPRAMs at *RAM-Channel* 0 and *RAM-Block* 0, and the data points for the last row of the input register array are available in address 0 of the six SPRAMs at *RAM-Channel* 3 and *RAM-Block* 0.

The required size of the SPRAM depends upon the maximum input/output feature map size of the network and can be computed using Eq. 1.

$$Size_{SPRAM} = \frac{Dim_{image} \times Dim_{image} \times N}{N_{Rams} \times N_{Channels} \times N_{Blocks}} \quad (1)$$

B. Data Re-Packaging Unit

It computes the address in IO-buffer for each pixel. The data of the first input feature map and the results of the output feature maps are

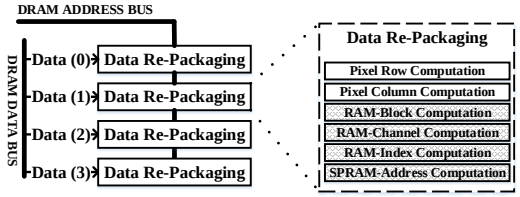

Fig. 7: Re-Packaging Unit

in a linear format as illustrated in Figure 4(c). However, to implement the dataflow illustrated in Figure 3 effectively, we need to re-arrange the data before storing it in the on-chip memory. The proposed re-packaging unit computes the placement location of each pixel in the distributed on-chip memory in terms of *RAM-Block*, *RAM-Channel*, *RAM-Index* (index of an SPRAM inside a block of the grid), and address of the SPRAM.

Figure 7 illustrates that four data points, each of which is 16-bits, are extracted from a single location of the DRAM (64-bits) and are provided to four data re-packaging units. Each unit computes the placement information of the provided pixel. The data re-packaging unit is composed of six computational stages, as illustrated in Figure 7, each of which computes a piece of specific information required to store a pixel into the on-chip memory. The following subsections provide details of each stage.

1) Image Pixel Row Computation: The first stage of the re-packaging unit computes the row index of the pixel in the input feature map. This computation is performed using the DRAM address from which the data is fetched along with the information of the number of pixels that can be stored in one location of the DRAM, the dimension of the input image, and the index of the pixel in the DRAM location to compute the row address, as given in Eq. 2.

$$P_{row} = \frac{(Add_{DRAM} \times NP_{DRAM})}{Dim_{image}} + \frac{dp_x}{Dim_{image}} \quad (2)$$

TABLE I: Notation used in the explanation of Re-Packaging phases

Symbols	Description
Size_{SPRAM}	Size of a SPRAM in distributed memory architecture
N_{RAMs}	Number of SPRAMs in one channel of distributed memory architecture
$\text{N}_{Channels}$	Number of Channels in one block of distributed memory architecture
N_{Blocks}	Number of Blocks in distributed memory architecture
N	Number of bits of one input pixel
P_{row}	Row index of input pixel
P_{col}	Column index of input pixel
Add_{DRAM}	Address of DRAM access
NP_{DRAM}	Number of data point in one DRAM location
dp_x	Data point index out of total number of data point from one DRAM location
Dim_{image}	Feature map dimension (rows/columns where number of rows and columns are same)
Blobk_{Index}	Block-Index of IO-Buffer
ISODD(x)	Results 1 if x is odd number
ISEVEN(x)	Results 1 if x is even number
Channel_{Index}	Channel-Index of IO-Buffer
$\text{Temp}_{RAMIndex}$	Local temporary variable used in computation of RAM-Index of IO-Buffer
X (mod Y)	Remainder when X is divided by Y
X [B1:B2]	Bit wise selection of X with B1 as MSB and B2 as LSB
X [B1]	Bit wise selection of bit B1 of X
RAM_{Index}	RAM-Index
$\text{RAM}_{Address}$	SPRAM Address

2) Image Pixel Column Computation: Pixel column index is computed using the information of pixel row index (as computed in the previous stage) along with the information of the DRAM address, the total number of data points in one DRAM location, input image dimension and index of the data point in the DRAM location. Eq. 3 represents the computation of column index of a pixel.

$$P_{col} = (Add_{DRAM} \times NP_{DRAM}) - (P_{row} \times Dim_{image}) + dp_x \quad (3)$$

3) RAM-Block Index Computation: After computing the input row and column indexes of a data point, we compute the *RAM-Block* index based on the type of the pixel (i.e., *even-even*, *even-odd*, *odd-even*, and *odd-odd*). The index can be computed using Eq. 4. However, it is implemented using the least significant bits of the row and column indexes computed in the previous stages.

$$\text{Block}_{Index} = ISODD(P_{col}) + 2 \times ISODD(P_{row}) \quad (4)$$

4) RAM-Channel Index Computation: As mentioned in Section IV-A, the data points which have to be loaded in the same row of the input register array are placed in the same *RAM-Channel*. Algorithm 1 uses the type of pixel and the least significant three bits of the row index of the data point to compute the *RAM-Channel* index.

Algorithm 1 On-Chip Buffer's Channel Index Computation

```
if P_row[0] XOR Block_Index[1] then
    Channel_Index ← 3
else
    Channel_Index ← P_row[2 : 1]
end if
```

Example: Consider that the first row of the input register array in Figure 4(a) contains starting six *even-even* pixels of the very first row (row index 0) of an input feature map. Since the type of pixels is *even-even*, *Block-Index* computed for these pixels is 0. Also, the least significant three bits of the row index being $(000)_2$ will result in the *RAM-Channel* index to output 0 as well. On the other hand, row 3 of the input register array in Figure 4(a) contains starting six *even-even* pixels of the seventh row (row index 6) of the input feature map and the least significant three bits of the row index of the pixels being $(110)_2$ will result in the *RAM-Channel* index to be 3.

5) RAM-Index Computation: The *RAM-Index* represents the index of the SPRAM inside a *RAM-Channel* in which a particular data point has to be stored. Algorithm 2 presents the pseudocode for computing the *RAM-Index* for an input pixel/data point where the range of the *RAM-Index* is between 0 to 5 based on the number of SPRAMs in a channel.

Algorithm 2 On-Chip Buffer's RAM-Index Computation

```
Temp_RamIndex ← P_col[3 : 1] + 1
if Temp_RamIndex > 6 then
    if Temp_RamIndex(mod 6) == 0 then
        Ram_Index ← 5
    else
        Ram_Index ← (Temp_RamIndex(mod 6)) − 1
    end if
else
    Ram_Index ← Temp_RamIndex − 1
end if
```

Example: Consider the input pixel $I(0, 6)$ in row 0 and column 3 in the input register array; see Figure 4(a). Bits [3:1] in $(0110)_2$ give $(011)_2 = 3$. Thus, the input pixel $I(0, 6)$ will be stored in *RAM-Index* 3.

6) SPRAM Address Computation: Given the size of input feature map and column index of the input pixel, Algorithm 3 is used to compute the address of a particular SPRAM selected by $Block_{Add}$, $Channel_{Add}$ and RAM_{Index}. Two address offsets are computed before the computation of the RAM address. For a single row of input feature map there exist only two types of pixels which reduces the size of a row to be stored in a *RAM-Channel* to half. Furthermore, each *RAM-Channel* consists of six SPRAMs and the complete row is folded accordingly to be stored in six RAMs of a *RAM-Channel*. The starting address (base address) of the next row of input feature map within a *RAM-Channel* is determined by $Address - Offset - 1$. For example, if the input feature map is of dimension 32x32 then one row of pixels

will be stored in 3 locations of a SPRAM (from address 0 to address 2). Therefore, when a particular SPRAM is selected again to store a data value, its address must start from address 3. If the input feature map is greater than 12 pixels, the same SPRAM will contain multiple data values of the sample row. Therefore, $Address - Offset - 2$ is the address computed on the basis of the column index of the input pixels.

Algorithm 3 On-Chip Buffer's SPRAM Address Computation

1: $Ram_{Address\text{-}Offset\text{-}1} \leftarrow (Dim_{image}/(2*6)) + 1$
2: $Ram_{Address\text{-}Offset\text{-}2} \leftarrow (P_{col}/(2*6))$
3: $Ram_{Address} \leftarrow Ram_{Address\text{-}Offset\text{-}1} + Ram_{Address\text{-}Offset\text{-}2}$

C. Data Flow Controller

Since each PE of the *ZFOST* microarchitecture is linked with output $O_{(ox,oy)}$, its required input pixels for kernel weight $K_{(x,y)}$ can be computed using $I_{(kx,2ox,ky+2oy)}$ for *strided convolution* and using $I_{(kx,ox,ky+oy)}$ for *transposed convolution*. Data flow controller loads the kernel weight in a *type-oriented format*, as illustrated in Figure 3, and generates addresses for data to be fetched from the on-chip memory in the form of *RAM-Block*, *RAM-Channel*, *RAM-Index* and *SPRAM-Address* using the same operations defined in Section IV-B. The main controller reads the data from distributed on-chip memory and provides it to the dataflow controller which loads the data into local registers of the *ZFOST* microarchitecture.

D. Main Controller

Initially, the main controller fetches data from the off-chip memory and provides it to the data re-packaging units which compute the destination address of each pixel in the on-chip memory. Once the first layer is fetched, the main controller provides a signal to the dataflow controller to start performing the desired operation on the available input feature map. The main controller receives addresses from data flow controller in the form of *RAM-Block*, *RAM-Channel*, *RAM-Index* and *SPRAM-Address* for the required data and reads the data from the on-chip memory and provides it to the dataflow controller.

V. Results and Discussion

A. Experimental Setup

Our MemGANs architecture (as shown in Figure 5) is designed in Verilog HDL. For functional validation, we synthesized the complete design using Xilinx Vivado Design Tool 2017.4, for the traget device Xilinx Kintex-7 "xc7k325tffg900-2" FPGA with the clock speed of 200MHz. The size of an SPRAM in distributed memory architecture is set to 32KB which fulfills the memory requirement for an image dimension up to 1024x1024 pixel for a 16-bit fixed-point system. To verify the functionality of our MemGANs architecture, we used a trained model of DCGAN [8] and extracted input feature maps and kernel maps using Matlab which are then used to evaluate MemGANs for transposed convolution. These weights and input features are preloaded in a DRAM as stimuli in linear format as illustrated by Figure 4(c). Since the operation of *strided/transposed convolution* is performed on multiple layers of a DCGAN which can have different configurations/dimensions, as illustrated in Figure 1, we evaluated our design using different image sizes with the kernel of 4x4 size. We computed performance, resource utilization and the number of read-write accesses using Vivado HDL Design flow. Moreover, to emulate a real-world ASIC-based implementation, the memory configuration used in Vivado Design Tool is also provided to CACTI-p tool by HP to compute the read/write accesses energy consumption of memory.

To compare the performance of MemGANs comprising our proposed distributed on-chip memory design, we also implemented a state-of-the-art-based design [11] (in a reproducible fashion) containing a

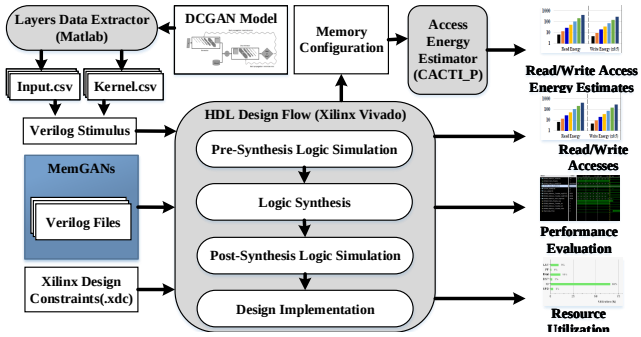

Fig. 8: Tool Flow for evaluation of MemGANs

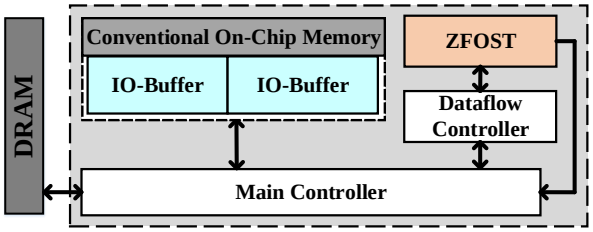

Fig. 9: Implemented Architecture for evaluation of conventional on-chip memory

conventional on-chip buffer that stores data in a linear format. The data bus of this memory is kept 64-bits. Therefore, for a 16-bit fixed-point system, it can store 4 data points in one location. Size of this conventional memory is set to 512KB to fulfill the requirement of storing input image of size up to 1024x1024 pixels. Figure 9 illustrates the overall hardware design using the conventional on-chip memory. The dataflow controller in this design also arranges the data in the *type-oriented format* before loading it into the *ZFOST* to fulfill the dataflow requirement as described in Figure 3.

B. Comparison with State-of-the-Art

First, we demonstrate the benefits of our design in terms of its performance, including the reduction in the number of accesses from the on-chip memory which further improved the overall processing time and reduced the read/write energy consumption of the on-chip memory. Next, we provide a comparison of the resource utilization of our proposed MemGANs architecture with the architecture illustrated in Figure 9 composed of conventional on-chip memory design.

Performance Evaluation: We compute the number of read/write accesses of the on-chip memory, energy consumption of the associated read/write accesses, and the overall time to process with input feature maps of different sizes. Figure 10 illustrates that our MemGANs reduces the number of read accesses and write accesses by 85% and 75%, respectively, when compared to the architecture shown in Figure 9 (with 64-bits wide SRAM). Figure. 11 shows that, on average, over different image sizes, MemGANs reduces the energy consumption of read and write accesses by 65% and 58%, respectively, when compared to the state-of-the-art (with 64-bits wide SRAM). Figure 10 and Figure 11 also show the comparison of the proposed with the design of Figure 9 when the bit-width of the SRAM is reduced from 64 to 16 bits. As shown in the figures, in this case, the proposed MemGANs reduces the read and write accesses by 92% and 93%, respectively, and the energy consumption associated with the read and write accesses by 82% and 89%, respectively. As the results of the conventional design with 16-bit wide SRAM are worse than with 64-bit wide SRAM, for all rest of the following results we only considered the 64-bit wide SRAM in the conventional design.

Since the feature map for the first layer needs to be loaded from the off-chip memory, the loading time for both MemGANs and the

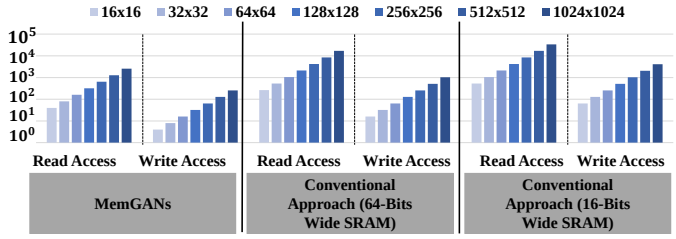

Fig. 10: Comparison between number of accesses

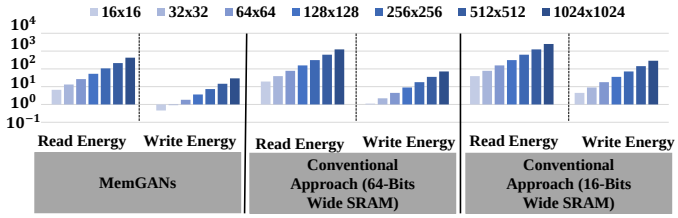

Fig. 11: Comparison between accesses Energy

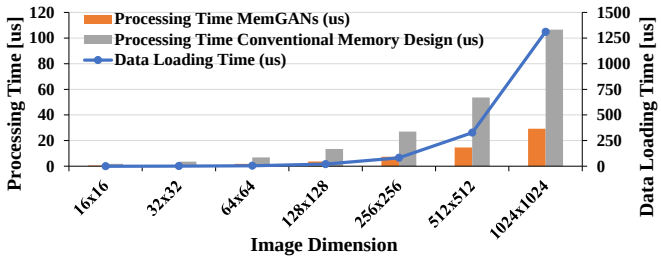

Fig. 12: Performance evaluation between MemGAN and the design of [11] for transposed convolution

design of Figure 9 is the same. Figure 12 illustrates the data loading time from off-chip memory along with the processing time by both the hardware designs. Our MemGANs outperforms the design of Figure 9 in processing time of an input feature map. Over different image sizes, MemGANs supports the ZFOST to achieve 3.65x faster processing time than the baseline. In terms of the overall power consumption the proposed MemGANs consume 5% more power compared to the design of Figure 9 due to the additional re-packaging units used in the design. The overall power consumption as well as the power breakdown of both the designs is shown in Table III.

Resource Utilization: To compare the utilization of hardware resources, we implemented both, MemGANs (Figure 5) and the design of [11] (Figure 9), using Xilinx Vivado design tool for the Xilinx Kintex 7 device (xc7k325tffg900-2 FPGA). As shown in Table II, MemGANs utilizes more resources in comparison to the design of Figure 9, thereby providing trade-off for obtaining improved performance and energy efficiency, as discussed in the previous section. Table II illustrates that MemGANs utilizes 6x more BRAMs, whereas the utilization of look-up-tables, flip flops and the DSP blocks on FPGA is increased by 1.16x, 1.14x, and 1.14x, respectively, when compared to the design of Figure 9.

TABLE II: Comparison of Resource Utilization

	Available	Utilized	
		Mem-GAN	**Conventional Memory Design**
LUT	203800	19138	16439
FF	407600	984	863
RAM	445	48	8
DSP	840	16	14

TABLE III: Power breakdown comparison of the proposed design with the conventional design

Module	**Power Consumption Break Down [W]**	
	MemGANs	**Conventional Design**
Re-Packaging Unit (DRAM to IO-Buffer)	0.08	-
Re-Packaging Unit (ZFOST to IO-Buffer)	0.50	-
Main Controller	0.04	0.18
Dataflow Controller	0.12	0.39
ZFOST	0.10	0.10
IOBUFFER (BRAM Blocks in FPGA)	1.74	1.79
Total Power	**2.59**	**2.46**

VI. Conclusion

In this paper, we proposed a novel on-chip memory architecture (MemGANs) with an optimized data re-organization unit for improving the performance and energy efficiency of GANs training using the *Zero-Free Output STationary (ZFOST)* microarchitecture. Our architecture fulfills the requirement of multi-pixel load to the input register array of the *ZFOST*. The re-packaging unit provides data organization support, which helps to organize the data in the required *type-oriented format*.

When compared to the state-of-the-art design of [11], our MemGANs reduces (1) the number of memory read and write accesses by 85% and 75%, respectively; and (2) the energy consumption during the read and write accesses by 65% and 58%, respectively. Moreover, MemGANs supports ZFOST to achieve 3.65x faster processing time as compared to state-of-the-art [11].

Acknowledgment

This work was partially supported by the Erasmus+ International Credit Mobility (KA107).

References

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

[2] X. Zhu, Z. Ghahramani, and J. D. Lafferty, "Semi-supervised learning using gaussian fields and harmonic functions," in *Proceedings of the 20th International conference on Machine learning (ICML-03)*, 2003, pp. 912–919.

[3] M. Noroozi and P. Favaro, "Unsupervised learning of visual representations by solving jigsaw puzzles," in *European Conference on Computer Vision*. Springer, 2016, pp. 69–84.

[4] E. L. Denton, S. Gross, and R. Fergus, "Semi-supervised learning with context-conditional generative adversarial networks," *CoRR*, vol. abs/1611.06430, 2016. [Online]. Available: http://arxiv.org/abs/1611.06430

[5] M. Song, K. Zhong, J. Zhang, Y. Hu, D. Liu, W. Zhang, J. Wang, and T. Li, "In-situ ai: Towards autonomous and incremental deep learning for iot systems," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 92–103.

[6] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in neural information processing systems*, 2014, pp. 2672–2680.

[7] 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.

[8] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," *CoRR*, vol. abs/1511.06434, 2015. [Online]. Available: http://arxiv.org/abs/1511.06434

[9] W. Lawson, E. Bekele, and K. Sullivan, "Finding anomalies with generative adversarial networks for a patrolbot," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2017, pp. 12–13.

[10] A. Ghosh, B. Bhattacharya, and S. B. R. Chowdhury, "SAD-GAN: synthetic autonomous driving using generative adversarial networks," *CoRR*, vol. abs/1611.08788, 2016. [Online]. Available: http://arxiv.org/abs/1611.08788

[11] M. Song, J. Zhang, H. Chen, and T. Li, "Towards efficient microarchitectural design for accelerating unsupervised gan-based deep learning," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 66–77.