A Survey of Hardware Architectures for Generative Adversarial Networks

Nivedita Shrivastava[1], Muhammad Abdullah Hanif[2], Sparsh Mittal[3],
Smruti Ranjan Sarangi[1] and Muhammad Shafique[4]
[1]IIT Delhi, India, [2]Technische Universität Wien (TU Wien), Austria
[3]IIT Roorkee, India, [4]New York University Abu Dhabi, UAE.
nivedita.shrivastava@ee.iitd.ac.in, muhammad.hanif@tuwien.ac.at, sparshfec@iitr.ac.in,
srsarangi@cse.iitd.ac.in, muhammad.shafique@nyu.edu

Abstract

Recent years have witnessed a significant interest in the "generative adversarial networks" (GANs) due to their ability to generate high-fidelity data. Many models of GANs have been proposed for a diverse range of domains ranging from natural language processing to image processing. GANs have a high compute and memory requirements. Also, since they involve both convolution and deconvolution operation, they do not map well to the conventional accelerators designed for convolution operations. Evidently, there is a need of customized accelerators for achieving high efficiency with GANs. In this work, we present a survey of techniques and architectures for accelerating GANs. We organize the works on key parameters to bring out their differences and similarities. Finally, we present research challenges that are worthy of attention in near future. More than summarizing the state-of-art, this survey seeks to spark further research in the field of GAN accelerators.

Index Terms

Review, deep neural networks, generative adversarial network, transposed convolution, dilated convolution, GPU, FPGA.

1 Introduction

Artificial intelligence has now pervaded all fields of human endeavor. Most of the artificial intelligence models are based on supervised machine learning models. However, supervised models require a large dataset with millions of labels. This requirement prevents the deployment of supervised models in domains where labels are costly or difficult to collect. A promising solution to this issue is the automatic generation of the dataset by unsupervised and semi-supervised learning models. "Generative adversarial networks" or GANs [1] is one of the most popular and effective unsupervised generative models for generating realistic-looking fake data. The key advantage of using GAN is that it can easily learn patterns or regularities of a complicated input dataset. Once trained, the model can be used for the generation of new artificial realistic samples which share similar features as the original data. GANs have been successfully used in various applications, such as autonomous driving [2], video prediction [3], image super-resolution [4], image synthesis, text-to-image conversion [5], style transfer [6], providing robots that can learn automatically from the surrounding without any human intervention, and so forth.

A crucial challenge in the use of GANs, however, is that their network structure is quite complex. In GAN training, two competing neural networks, viz., generator and discriminator, are trained alternatively. In GAN, there is a symmetry between the generator and the discriminator, and they are trained alternatively. The training of the generator aims at producing fake samples, whereas the discriminator's training is done using both real and fake data samples. Due to this, the execution latency of a GAN is much higher than that of a conventional CNN. Facilitating the collaboration between them requires transferring a huge amount of data between them. These factors underscore the need for the hardware acceleration of GANs.

Hardware acceleration of GANs presents challenges of its own. The existing hardware accelerators are aimed at discriminative CNNs. However, the generator network uses an "up-sampling fractionally-strided CNN" which is significantly different. A GAN accelerator needs to be generic enough to be able to accelerate both the generative and discriminative networks. For example, it needs to optimize the data-reuse in both a regular CONV and a DeCONV. Further, GAN computations involve irregular data dependencies, which lead to a high amount of bandwidth pressure [7]. Further, the DeCONV operation performs up-sampling of the input feature map using either zero-insertion or nearest-neighbor-based approach. However, the zero-insertion strategy leads to a large number of ineffective computations. With

Sparsh is the corresponding author. Sparsh, Abdullah and Nivedita are co-first authors. *"Support for this work was provided by Semiconductor Research Corporation."*

4×4 input and a stride of 2, nearly 87% computations are redundant, and with 16×16 input and a stride of 32, 99.8% computations become redundant [8]. The nearest-neighbor-based upsampling involves repeated multiplications. Evidently, the design of GAN accelerators presents unique challenges beyond those posed in the design of CNN accelerators. Several recent works have sought to address these challenges.

Contributions: In this paper, we present a survey of hardware architectures and optimization techniques for GANs. Figure 1 presents an overview of this paper. Section 2 provides a background on important concepts. It also classifies the works on key parameters. Section 3 reviews the research works in terms of their architectures and Section 4 looks at them in terms of the optimization techniques used by them. In these sections, we review a technique in a single section only, even though many of the research works belong to multiple categories. Since different works use different experimental platforms and workloads, we only discuss their key ideas and do not present their quantitative results. Section 5 discusses future research challenges that are worthy of attention from researchers. Finally, Section 6 concludes this paper.

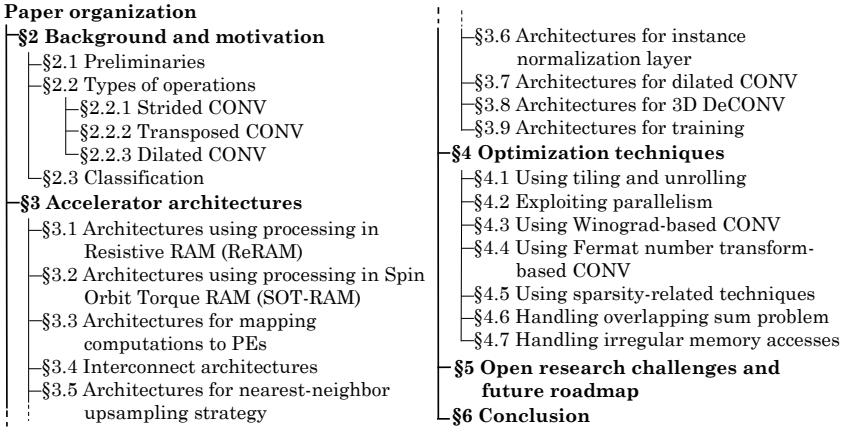

Fig. 1. Organization of the paper

Following acronyms are used frequently in this paper: 3D data wire connection unit (3DDC), backward/forward propagation phase (BWP/FWP), convolution (CONV), convolutional/deep neural network (CNN/DNN), deconvolution (DeCONV), fast Fourier transform (FFT), Fermat number transform (FNT), input/output feature map (ifmap/ofmap), long stort term memory (LSTM), look-up table (LUT), matrix multiplication (MM), matrix vector multiplication (MVM), memristive crossbar array (MCA), multi-level cell (MLC), multiple/single instruction multiple data (MIMD/SIMD), multiply-accumulate (MAC), processing element (PE), processing-in-memory (PIM), "resistive RAM" (ReRAM), "spin orbit torque RAM" (SOT-RAM), sub-crossbar tensor (SCBT), transposed convolution (TrCONV), vector-matrix multiplication engine (VMME)

2 Background and Motivation

In this section, we first present some preliminary ideas about GANs (Section 2.1). This is followed by an introduction to some operations performed by a GAN (Section 2.2). Finally, we present a classification of research works (Section 2.3).

2.1 Preliminaries

A GAN has two competitive neural networks: A "generator" and a "discriminator". These neural networks compete with each other to learn a high-dimensional data distribution. The generator comprises a "deconvolutional neural network" (DeCNN) whereas a discriminator comprises a CNN. The former is used for artificial data generation, and the latter is used for distinguishing the synthetic data from the actual data. The aim of the training procedure of GANs is to acquire a generator that can generate nearly identical artificial data and a discriminator that can effectively extract features and distinguish between the real and artificially generated data.

Figure 2 presents a high-level representation of a GAN design. The error calculation block calculates the error between the artificial data and the real data. This error is propagated back to both the neural networks, and weights are updated accordingly. In the training phase, the discriminator and generator are trained alternately, which leads to a high requirement of memory and computational resources.

2.2 Types of Operations

We now discuss and distinguish different types of operations performed by a GAN.

2.2.1 Strided CONV

Strided CONV (no zero-insertion): Figure 3(a) presents a strided convolution operation which is a basic convolution operation with a stride of 2. As the kernel is skipped by the stride of 2, half of the CONV operations from both the vertical and horizontal directions are skipped. This process generates a quarter-sized ofmap.

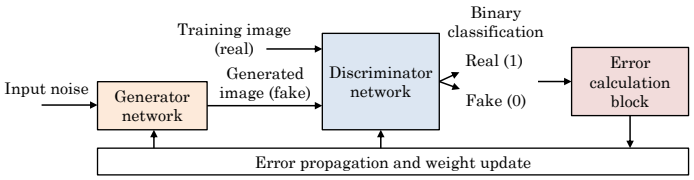

Fig. 2. Deep convolutional generative adversarial network. Generator is used for the generation of the artificial data and the discriminator learns to differentiate between the real data and the artifically generated data. [9]

2.2.2 *Transposed CONV*

Transposed convolutions (TrCONV) are also known as "fractionally strided convolutions" or "up convolutions". A TrCONV does not ensure recovering the input since, mathematically, it is not the inverse of the CONV. Instead, it just outputs a fmap having the same dimension as the input fmap. It performs the regular CONV, but it reverts its spatial transformation. It gracefully combines the process of up-scaling an image and convolution, which proves to be very helpful in the encoder-decoder-based architectures. For performing upsampling, zeros are inserted between the non-zero inputs as well as around the image border. Figure 3(b) presents the TrCONV process.

It is crucial to highlight that the DeCONV is mathematically defined as the inverse of a convolution. Hence, a transposed CONV is not exactly the same as a DeCONV. Still, most papers use them interchangeably. Following these works, we use these terms interchangeably.

2.2.3 *Dilated CONV*

The dilated convolution is also known as an atrous convolution. In dilation convolution, zeros are inserted only in the kernels, which leads to dilated kernels with an enlarged receptive field [10]. The dilated CONV uses a parameter called dilation rate, which shows the spacing between the weights in the kernel. A 3×3 kernel with a "dilation rate" of 2 has similar "field of view" as a 5×5 kernel. Figure 3(c) presents the Dilated CONV process.

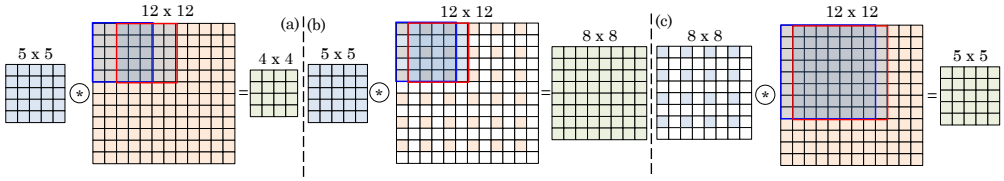

Fig. 3. Different types of operations: (a) Traditional (direct) CONV with a stride of 2, called strided-CONV (no zero-insertion) (b) Transposed CONV (zero-insertion in the input) (c) Dilated CONV (Zero-insertion in kernel)

Table 1 distinguishes transposed CONV from dilated CONV [11–14].

TABLE 1
Difference between transposed CONV and dilated CONV

	Zero-insertion in	
	Input	Kernel
Transposed CONV (also called fractionally strided CONV, upconvolution or deconvolution)	Yes	No
Dilated CONV (also called atrous CONV)	No	Yes

2.3 Classification

Table 2 classifies the research works based on the machine-learning phase they target, optimization metric, their workload and datasets.

3 Accelerator Architectures

We now discuss the architectures of GAN accelerators. Section 3.1 reviews processing in ReRAM and Section 3.2 reviews processing in SOT-RAM. Section 3.3 discusses strategies for matching computation-pattern to the accelerator architecture and Section 3.4 reviews interconnect architectures. An architecture for handling nearest-neighbor upsampling strategy is reviewed in Section 3.5 and one for handling "instance normalization layers" is discussed in Section 3.6. Architectures for dilated CONV and 3D DeCONV are discussed in Sections 3.7 and 3.8, respectively. Finally, architectures for training a GAN are discussed in Section 3.9.

Table 3 classifies the works based on their architectures and computing platforms.

TABLE 2
A classification of research works

Machine learning phase	training [9, 15–20], inference [10, 15, 17, 21–26]
Optimization metric	Energy [8, 16, 17, 19–21, 21, 22, 24, 26–41], performance or accuracy (nearly all)
Benchmarks	DCGAN [8, 10, 19, 20, 22–24, 26–28, 31–34, 37, 40–44], ArtGAN [19, 27, 28, 32, 37, 41, 42, 44], DiscoGAN [19, 28, 32, 33, 37, 41, 42] , GP-GAN [19, 22, 28, 32, 41, 42, 44], MAGAN [19, 28, 41] EB-GAN [42], 3D-GAN [19, 22, 27, 28, 41, 42], 3D-ED-GAN [42], V-Net [22] UP-GAN [26], U-Net [29, 36, 37, 39, 43], E-Net [30, 31], C-GAN [10, 20, 26], DN-GAN [26], iGAN [27] SNGAN [8, 24, 44] cGAN [19], fully-convolutional network [8, 24, 30, 35, 37], FSRCNN [25, 37, 43, 45], super-resolution network [35], MDE [44], FST [44] style-transferGAN [21, 35], CycleGAN [21], DeblurGAN [21], StarGAN [33], WGAN-CP [18, 34], WGAN-GP [18, 34], DeepLabv3+ [30]
Dataset	MNIST [9, 17, 19, 20, 23, 34], CIFAR-10 [8, 17–19, 24, 31, 34, 44], CIFAR-100 [10], CelebA [30, 34, 43, 44], STL-10 [8, 18, 34], Cityscapes [24, 31, 36], KITTI [24], PASCAL VOC [8, 24], Fashion-MNIST [34], Set5, Set14, and B100 [45], transient attributes [44], FSP [30], RH [30], LSUN [8, 17, 24]

TABLE 3
A classification of algorithms, architectures and computing platforms

Algorithm type	
DeCONV (transposed CONV)	[10, 16, 17, 19–43, 45–47]
Dilated CONV	[10, 29–31]
Hybrid architecture	
CONV-DeCONV	[20, 21, 24, 25, 31, 33, 35–37, 39, 41, 43, 46, 47]
DeCONV-dilated CONV	[10, 30, 31, 37]
DeCONV-CONV-dilated CONV	[31]
Method for accelerating DeCONV operation	
Zero skipping/removal	[17, 19, 20, 24, 28, 30, 31, 40, 41]
TDC (transforming DeCONV into CONV)	[10, 23, 25, 27, 29, 32, 43, 44]
Reverse Looping	[48]
FNT	[33]
Kernel based Operations	Conversion [35, 37, 39, 47], Decomposition [10, 23, 25, 27, 29, 32, 43, 44]
Hardware architecture	
FPGA	[20–23, 25, 26, 28, 29, 32, 36–38, 40, 42, 43, 45–48]
ASIC	[9, 18, 24, 33, 39, 41]
Edge TPU	[44]
PIM-based	ReRAM [8, 9, 16–19, 27, 34], SOT-RAM [18]
Comparison with	CPU [10, 20, 22, 26, 27], GPU [10, 16–20, 22, 26–29, 34, 37], FPGA [17, 19, 23, 25, 32, 33, 36–38, 40, 43, 46], ASIC [9, 18, 24, 27, 33–35, 39, 41], PIM-based [19, 27]

3.1 Architectures using Processing in Resistive RAM (ReRAM)

TrCONV involves augmenting a low-dimension ifmap to a high-dimension fmap. In the first step, multiple zeros are inserted in the input data, and in the second step, CONV is performed on the expanded fmap. These zeros are responsible for upwards of 60% computations, and hence, the execution of GANs on traditional DNN architectures is highly inefficient.

Chen et al. [27] propose a PIM-based ReRAM architecture for accelerating GAN computations. Consider the example of TrCONV between 4×4 input and 5×5 kernel shown in Figure 4(a). They note that the CONV can have only four different valid patterns, which are shown by various colors. The elements in odd and even rows of output are computed from the odd and even (respectively) rows of the weight matrix. Based on this, they separate the weight matrix into several subsets and perform CONV only among suitable subsets of the weight kernel and the original input. This is shown in Figure 4(b). This approach is used in both FWP and BWP phases.

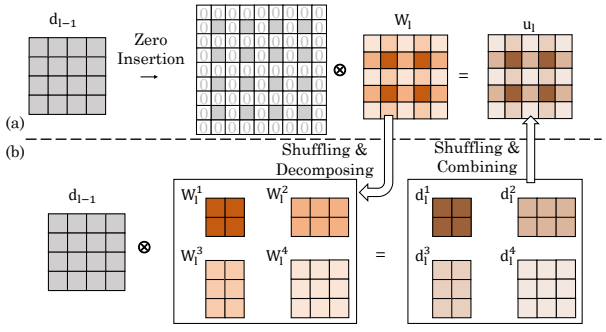

Fig. 4. (a) 2D TrCONV for a 4×4 input and a 5×5 kernel. (b) Their proposed dataflow for executing 2D TrCONV [27]

Let P_w and P_h be the padding on width and height, respectively, and S_w and S_h be the stride on width and height, respectively. In general, $S_w > (P_w + 1)$ and $S_h > (P_h + 1)$. They partition the kernels in $S_w \times S_h$ subgroups. In the FWP

phase, the CONV between expanded ifmap and kernel can be obtained by CONV between original ifmap and multiple kernel subgroups. By properly arranging these ofmaps, the final ofmap can be obtained. In the error propagation phase, the error for layer $L-1$ is obtained simply by convolving the error matrix with the suitable subgroup of weight matrix since an element in ofmap of layer L is computed only from selected subgroups of the kernel. During CONV, ineffectual computations due to zero operand are skipped by the scheduler.

As for the mapping of matrix multiplication to memristive crossbar array (MCA), they note that a coarse-grain mapping leads to different computation latency for different kernel subgroups. For instance, Figure 5(a) presents the mapping of kernel matrix of layers L1 and L2 on an MCA. Due to the differences in the ofmaps of these layers, the computation latencies of these layers are different. This precludes the use of pipelining and also leads to an unbalanced load. To avoid this, they perform fine-grain mapping where multiple (say Q) copies of small-size MM are mapped on the "vector-matrix multiplication engine" (VMME). Figure 5(b) shows this with an example of Q=2 and Q=3, respectively. This leads to balanced computation latency and also enables pipelining of GAN training.

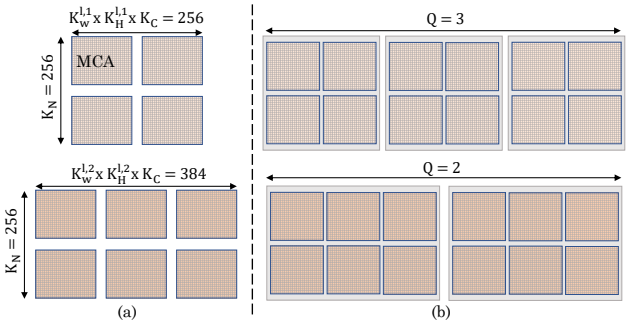

Fig. 5. Mapping matrix multiplication (MM) onto "vector matix multiplication engines" (VMMEs) using (a) coarse-grain approach (b) fine-grain approach [27]

Their overall architecture is shown in Figure 6. Their compute engine has ReRAM memory for storing inputs and outputs, and multiple PEs connected through the on-chip mesh, as shown in Figure 6(a). Every PE has multiple VMMEs (Figure 6(b)), a ReRAM buffer for caching intermediate output, activation units and output registers, etc. Every VMME has multiple ReRAM crossbar arrays (Figure 6(c)), of which only one array is activated at a time for performing processing in memory. For example, one PE has 4 VMMEs, each of which has 8 ReRAM crossbar arrays. To enable mapping of large-matrices, a shift-and-add unit is also used, which enables the accumulation of partial sums. They also use other units for performing various computations of GAN. To reduce the overhead of "digital-to-analog convertor" (DAC), its functionality is achieved with an inverter. Also, to reduce the overhead of "analog-to-digital convertor" (ADC), one ADC is shared between 8 ReRAM crossbar arrays.

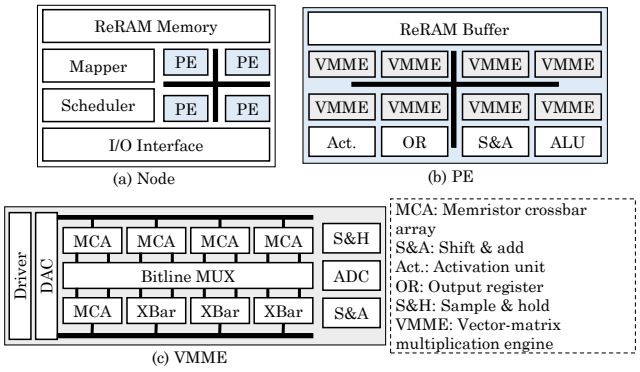

Fig. 6. (a) In the technique of Chen et al. [27], every node has ReRAM memory for storage, a mapper for mapping the dataflow to the PEs, and a scheduler for controlling the computation flow. Multiple PEs are connected through an on-chip mesh. (b) Every PE has many VMMEs, a buffer for caching temporal data, and a register for aggregating the output. (c) Every VMME has a few MCAs that share the ADC, a few 1b DACs, etc. MCAs in the VMME have a common driver. Only one MCA can be active at a time. A bitline MUX is used for selecting the results.

Overall, their technique benefits from achieving PIM execution [49, 50] and pipelining and avoiding computations on zero operands. Their technique achieves 146× speedup over an E5-2630 v3 CPU and 7.6× speedup over a Geforce GTX 1080 GPU. Also, it achieves higher performance and energy efficiency than both ReRAM [16] and CMOS-based [24, 41] DNN accelerators, for example, the speedup and energy efficiency improvement over ReGAN [16] are 1.15× and 1.5×, respectively. Further, by virtue of using 3D ReRAM, it achieves a lifetime of 3.4 years under continuous training, whereas ReGAN has a lifetime of 5 months only.

Fan et al. [8] note that the non-padding DeCONV strategy requires additional circuitry in ReRAM-based accelerators for performing addition and cropping operations. Further, while the output of zero-padding DeCONV is produced in

M columns (M=number of output channels), the non-padding DeCONV needs $M \times K_H \times K_W$ columns on a crossbar. Since the bitline/wordline driving power rises quadratically with the number of columns, the non-padding DeCONV consumes much higher power than zero-padding DeCONV. Figure 7(a) and 7(b) show padded and padding-free DeCONV operations, respectively.

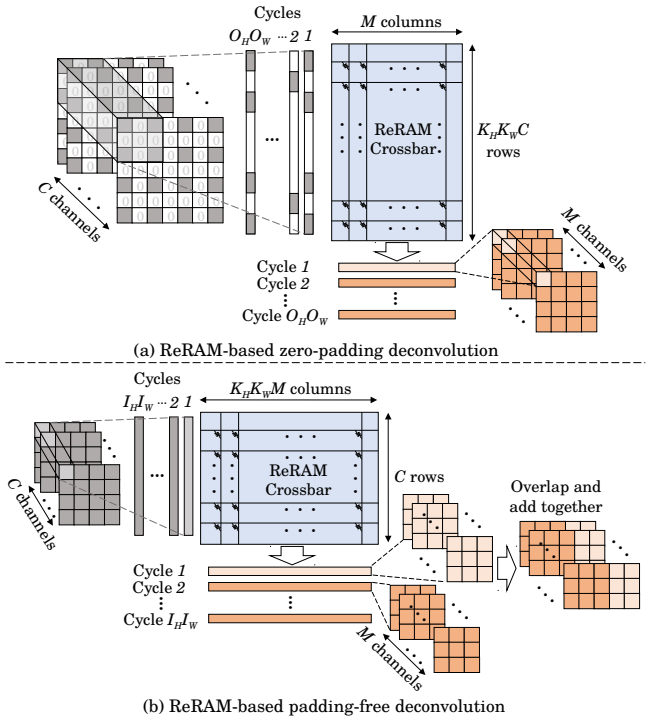

Fig. 7. ReRAM-based DeCONV which (a) requires padding and (b) does not require padding [8]

They propose architectural techniques for mitigating this inefficiency. Their overall architecture is shown in Figure 8(a). Their design uses $K_H \times K_W$ sub-crossbars of size $C \times M$, and thus, a sub-crossbar has C inputs and M outputs. To mitigate the redundancy due to zero-padding, they use pixel-level mapping, shown in Figure 8(b). The combination of sub-crossbars produces a "sub-crossbar tensor" (SCBT) of dimension $C \times M \times K_H \times K_W$. In pixel-level mapping, $SCBT[c, m, i * K_W + j] = W[i, j, c, m]$, where $[i, j]$ show the position of the weight in every filter, c shows the channel-index and m shows the index of weight filter.

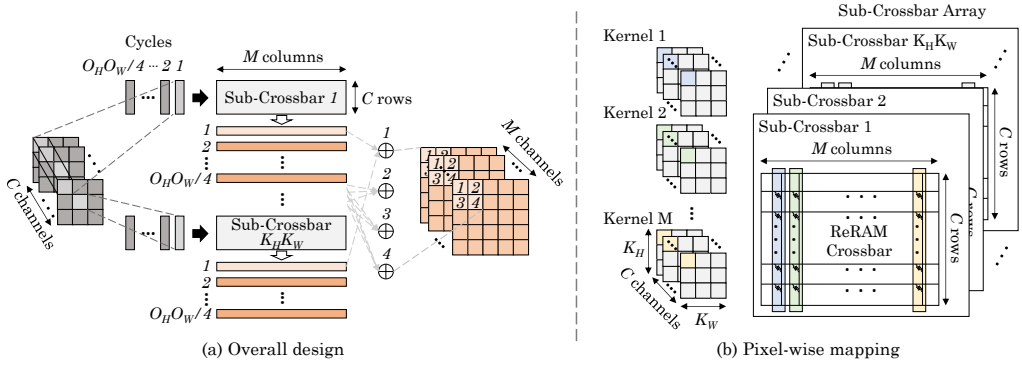

Fig. 8. (a) In the technique of Fan et al. [8], DeCONV is run by $K_H \times K_W$ sub-crossbars, each of size $C \times M$. Only non-zero pixels are taken for forming the input vector. The partial outputs from the subcrossbars are accumulated to obtain the output pixels. For every ofmap, multiple pixels are generated in parallel. (b) pixel-level mapping. A kernel of size $K_H \times K_W \times C \times M$ is mapped to $K_H \times K_W$ sub-crossbars, each of size $C \times M$.

In a CONV operation with a 3x3 kernel and a stride of 2, due to the high sparsity of expanded input, the CONV operation has only four distinct computation patterns, which are shown in Figure 9(a)-(d). Also, the weights involved in these patterns are also exclusive. A kernel of size $K_H \times K_W \times C \times M$ is mapped to $K_H \times K_W$ sub-crossbars (SCB). On completion of one computation cycle in SCBs, the output from the corresponding SCBs is added for obtaining the final DeCONV result. The ReRAM accelerator design already enables efficient vertical summation, and hence, additional circuits are not required for realizing the add operations.

They further present a dataflow that avoids computations on zero pixels. It is shown in Figure 9(e). For a 3×3 kernel, they use 9 SCBs. The output vectors which are generated from the SCBs which are located on the same row are summed

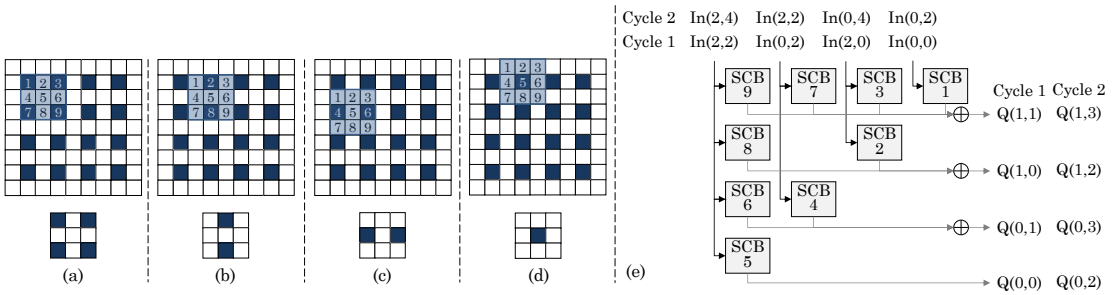

Fig. 9. (a)-(d) Four compute patterns in DeCONV for a kernel size of 3×3 and stride of 2 [8] (e) zero-skipping data flow (SCB =sub-crossbar, Q = output)

up together, and the same input is provided to the SCBs on the same column. Let $In(x, y)$ be the input vector. Since their dataflow skips padded zeros, they have even numbers as x and y index on the padded image.

In cycle 1, the inputs go as follows: In(0; 0) to SCB1, In(2; 0) to SCB2 and SCB3, In(0; 2) to SCB4 and SCB7, and In(2; 2) to SCB5, SCB6, SCB8 and SCB9. All 9 SCBs work simultaneously and their outputs are combined. In cycle 2, the next group of non-zero pixels are used, viz., In(0; 2), In(0; 4), In(2; 2) and In(2; 4). As compared to the zero-padding DeCONV, this dataflow improves parallelism by 4×.

The acceleration brought by their technique increases with the number of compute-patterns, which equals the square of the stride. However, with increasing stride, the number of SCBs required is also increased, which leads to area penalty. Based on this tradeoff, they decide the number of SCBs. Also, they add zeros to the input vector for reducing the number of SCBs to half of its original value. In comparison to the zero-padding architecture, their architecture attains a 3.7×-31.1× speedup for different benchmarks by virtue of lowering the array and periphery latency. The energy savings are between 8% to 88%. In comparison to the zero-padding design, their technique lowers the array latency since it uses output vectors of smaller size.

Liu et al. [17] present a ReRAM-based accelerator for GAN. The accelerator has blocks for discriminator, generator, difference and control. The discriminator and generator blocks perform the functionality of discriminator and generator, respectively. They have multiple ReRAM-based CNN and DeCNN units, respectively. The difference block is designed with two ReRAM-based LUTs, adders and memory units, and it calculates the gradients of generator and discriminator blocks. The size of the memory-unit depends on the batch-size. The control unit orchestrates the dataflow. Figure 10 shows the working of their design. (1): The real data y is supplied to the discriminator for computing D(y). (2): Fake data x is supplied to the generator for obtaining artificial samples G(z). (3): G(x) is sent to the discriminator for computing D(G(x)). (4): D(y) and D(G(x)) are sent to the difference block. The weights of discriminator (5), and generator (6) are updated based on the gradient computed by the difference block. Steps (1)-(4) are part of the FWP phase and (5)-(6) are part of the BWP phase.

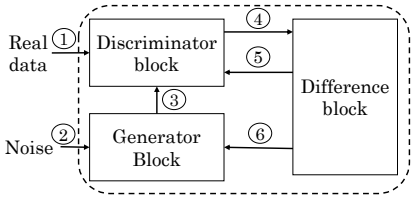

Fig. 10. In the technique of Liu et al. [17], the discriminator block computes D(x) and D(G(z)) for stochastic gradient descent computation. The generator block computes G(z) for generating the artificial samples for calculating the stochastic gradient in the update of generator weight. The difference block calculates the gradients of the discriminator and generator blocks.

They find that updating the generator and discriminator takes most of the time. Also, the generator and discriminator can work independently except in steps (2) and (3) and when weight update has not finished. Based on these, they develop a pipelined execution schematic which allows the generator and discriminator to work in parallel as much as possible. For example, steps (1) and (2) are executed in parallel. Since step (2) takes more time than step (1), the result of step (1) is stored in a memory block in the difference unit. Similarly, steps (5) and (6) run in parallel. Step (5) finishes before step (6), which allows the next training iteration of the discriminator to begin asynchronously without the need of extra memory in the discriminator block. The training of generator and discriminator is synchronized again before step (3) is executed in the next iteration. Their pipelining technique improves resource utilization and reduces latency.

They propose an architecture that has units for updating weights, computing errors and performing CONV/DeCONV operations. It is based on the ReRAM crossbar and has a parallel FWP phase and a memory-free BWP phase. It can act as a discriminator or a generator using suitable initial weights programmed on the DeCONV/CONV units. The initial or updated weights are programmed to the crossbar of the CONV (or DeCONV) computing engine, and the results are supplied to the difference block.

During the BWP phase, the weights are updated according to the gradient from the difference block. The transpose of weights are loaded from the operation units in FWP and applied to the crossbar of the "error computation units". The output of every layer in FWP is mapped to the crossbar of the "weight updating units". Their design does not need memory for storing the updated weights or inter-layer signals due to the PIM capability. The updated weights are programmed to the CNN computing units and inter-layer signals are mapped to the "weight updating units". The LUTs in difference block perform function $\nabla \log D(y)$ and $\nabla \log(1 - D(G(x)))$. From these, the value of error for discriminator and generator are obtained. For Lsun/bedroom dataset, their technique provides 6.1× energy reduction and 2.8× improvement in performance over a Geforce GTX 1080 GPU. Also, it provides 1.4× energy reduction and 5.5× improvement in performance over the FPGA implementation of the technique of Song et al. [20].

3.2 Architectures using Processing in Spin Orbit Torque RAM (SOT-RAM)

Rakin et al. [18] propose an efficient GAN training algorithm using ternary weights (-1,0,1). In the proposed method of GAN training, initially, statistical weight training and the weight ternalization (i.e. -1,0,1) are done. The loss-function ternary weight-based inference is deduced, which is backpropagated to update the full-precision weights. This reduces the computational complexity. The digital processing unit ternarizes the weights, as shown in Figure 11. These weights are then mapped to the sub-array, supporting flexible addition/subtract operations.

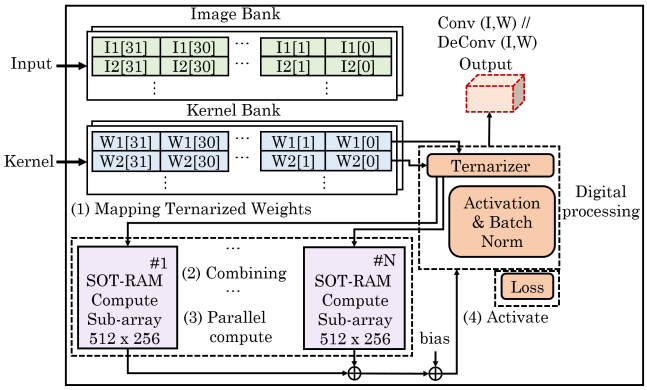

Fig. 11. The accelerator proposed by Rakin et al. [18] consists of image and kernel banks, a digital processing unit which includes four supporting units (loss function, batch normalization, activation function, ternarizer) and a SOT-MRAM based computational sub-array.

Due to ternarization, computationally intensive CONV and DeCONV are converted to subtract/addition operations. These operations are realized using a spin-orbit torque RAM-based processing-in-memory approach. Their overall approach is shown in the Figure 11. In step (1) digital processing unit maps the ternarized weights to the computational sub-array. In (2) combining is performed in the sub-array. In the next step (3) parallel computation is performed. Finally, in step (4) activated sub-arrays produces final ofmap. Their technique performs the two important steps: combination and parallel computation. For performing combination operation, the input's sign-bit is changed according to the kernel as shown in the Figure 12(a). They map this aggregate batch to the sub-arrays. In Figure 12(b), the 16 sub-arrays are divided into four mats. The generated aggregate batch of four channels is mapped to these four mats. These sub-arrays operate in parallel and use the addition/subtraction operations to produce the ofmaps. Their technique achieves 22× speedup and 25.6× better energy efficiency than the GPU platform.

3.3 Architectures for Mapping Computations to PEs

Yan et al. [35] propose an architecture that can support the CONV layer, DeCONV layer, and residual blocks. In the CONV operation with 7×7 input, 3×3 kernel, the output size is 5×5, and computing each output requires 9 multiplications. Based on this, for the CONV operation, they use the 'output-oriented mapping', whereby one PE is responsible for computing one output element. For a DeCONV operation with 3×3 input, 3×3 kernel, the output size is 3×3, but different outputs need different numbers of multiplications due to the zero-padding. On using one PE for computing a single output, different PEs have a different number of computations. They note that, in the DeCONV operation, the number of operations performed on each input is the same. Hence, for DeCONV operation, they use 'input-oriented mapping', which maps each input processing task to a single PE.

They further note that the residual block consists of the elementwise additions and the CONV layers. However, elementwise additions have a much smaller operation density than the CONV. This leads to high memory requirements and a poor PE utilization. To deal with this issue, they design a cross-layer dataflow for the residual block. They select the size of each tile such that there is no overflow in the input buffer. This tile is stored in the cold buffer. Then, two consecutive CONVs are performed on this data. The first CONV gives a part of the output, and then this part is sent to the second CONV. Finally, element-wise addition is performed in the accumulator between the cold buffer data and the convolved data. This process does not require any off-chip memory access.

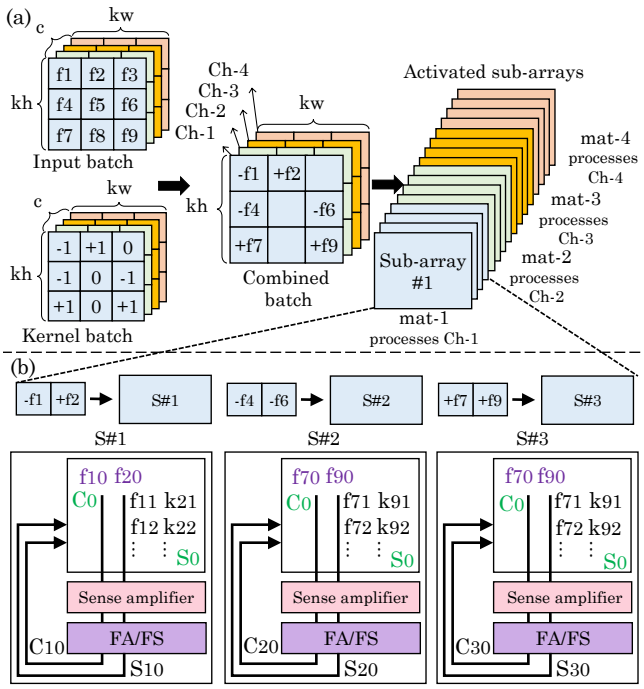

Fig. 12. (a) Combining operation: for combination, the sign-bit of the input is changed according to the kernel's value. (b) parallel computation: the aggregate batch computed in parallel on the sub-arrays [18] (FA/FS = full adder/subtractor, S#1 = sub-array 1, etc)

Their overall architecture is shown in the Figure 13. The computation core consists of four parallel CONV engines, each with 8×8 PEs. For the intra-PE processing, precision adaptive PEs are designed, which can support flexible bitwidth. Moreover, buffer bandwidth can reconfigure to either 256 or 128 bits, according to the computation mode of the PEs. Here, the computation mode refers to the use of different bitwidths such as 8 bit, 16 bit, etc. First, the kernel and the activation values are loaded into the CONV engine. During computations, the CONV engine receives data from the same rows but from the different channels. The partial sums are reused till the accumulation of the tiled output maps completes. Finally, the ofmaps are saved in the output buffer. A cold-buffer is used to store the overlaps, tiling results, and input data for the element-wise addition in the residual blocks. The coordinator module computes the coordinates of the output and arranges them in the ofmap. Their technique achieves 61 % higher PE usage than other traditional GAN accelerators. Proposed scheme achieves 2.05 TOPS/W energy efficiency.

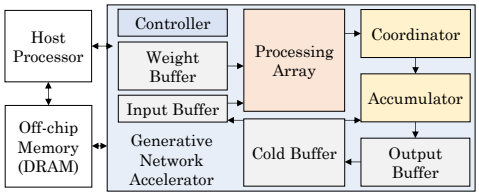

Fig. 13. The accelerator proposed by Yan et al. [35]. The convolution core performs CONV/DeCONV with four convolution engines which consists of (8×8) PE array. The cold buffer stores overlapping results, tiling results, and the input data for performing elementwise addition.

3.4 Interconnect Architectures

Complex dataflow in GAN and zero-insertion in the training phase degrades the performance of the GAN accelerator. Mao et al. [19] present a technique, termed zero-skipping data organization (ZSDO), which has two schemes: one for TrCONV operations and another for Weight-CONV of strided-CONV. Weight-CONV is different from both TrCONV and strided-CONV.

(1) **First scheme:** To convert CONV into matrix-vector multiplication (MVM), they reorganize kernel weights into vectors by extracting weights that are multiplied with the non-zero inputs. There are 512 kernels of size $W_h = W_w = 5$ and $W_h = 1024$. Four kernel weights that multiply with the non-zero input element are extracted to create $4 \times 1024 = 4096$ elements. All the 512 weight kernels are reorganized into a similar way to create a 512×4096 matrix. They are then multiplied with the correcponding 4096 inputs, which gives 512 results. Then, the kernel weight is slided by the stride of one which changes the weights used for multiplication. When kernels slide on the edge of ifmap or inside the ifmap, reshaped weight matrices get reused. They store 25 types of reorganized weight matrices.

(2) **Second scheme:** This scheme bears similarity with the first scheme. For the weight-CONV, zeros are removed from $\nabla output$. Then, it is reshaped as weight and CONV is performed on the input map for obtaining $\nabla weight$. Both the schemes have three types of reorganizations: corner, edge and inside. The reuse of weights is higher in the "inside" than in the "edge", whereas there is no reuse in the "corner". Due to this, the execution and transmission latency of inside-reorganization is higher than that of corner-reorganization. To balance these latencies, they create multiple copies of edge- and inside-reorganizations.

They further note that GAN accelerators have inefficient I/O interconnection. Previous CNN accelerators use H-tree routing for handling high number of memory accesses. However, since GANs have a more complex dataflow, training a GAN by implementing the phases to H-tree connection leads to many long routings. Further, strategies for simplifying the interconnection harms the performance of GAN. To mitigate this issue, they present a 3D PIM based efficient dataflow. They note that although multi-level cell (MLC) has high density, it has high write latency since it requires multiple rounds of program-and-verify operations. Due to this, weight update process becomes inefficient. They propose an approximate writing scheme, whereby the SET operations are avoided whenever the value to be written need not be precise but can stay within a certain range.

In their architecture, a 3D connected PIM is designed to adapt the dataflows. It can be better illustrated using a binary tree. Wires are added between two nodes which are having different parent nodes in the same layer. Then, three banks are piled up and wires are added between vertically-adjacent nodes. The nodes in middle-bank can connect with nodes in both upper and lower banks. Thus, a node is connected to its parent, horizontal neighbor and upper and/or lower neighbor. This technique removes the bottleneck of long data movement.

Each node also has an adder which may be bypassed. This 3D data wire connection unit (3DDC) can work in either computing mode or memory mode. In computing mode, the connections are configured dynamically based on the dataflows, whereas in the memory mode, the connections are statically composed in the H-tree style. Two 3DDCs can be connected using horizontal connections. They map generator to one or more 3DDCs and the discriminator to the 3DCUs connected to the generator. To achieve this, they split kernel weights and store them in multiple nodes. To further increase the parallelism, they duplicate weights in multiple nodes of a bank. This, however, leads to storage overhead.

Based on these techniques, a ReRAM based 3D connected accelerator is designed. Their overall approach is shown in Figure 14. A layer-by-layer network description is done in the program phase. The interface realizes zero-skipping data organization. The compiler maps the reshaped data. Then, the compiler's information is recorded by the memory controller. ReRAM based PIM communicates with the memory controller. Their technique provides higher performance than GPU and FPGA-based GAN implementations and a ReRAM-based neural network accelerator. Authors used NVIDIA Titan X GPU and Xilinx VCU118 board for the implementaion. It acheives 21.42×, 47.2× and 7.46× speedup as compared to GPU platform, FPGA-based accelerator, and ReRAM- based neural network. It also acheives 10.75×, 1.34× and 13.65× energy saving as compared to PRIME, FPGA-based and GPU-based accelerators, respectively.

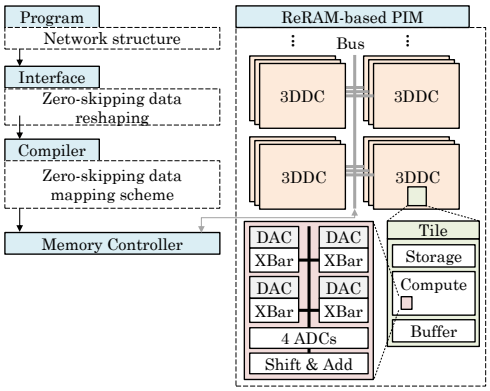

Fig. 14. The overall approach of Mao et al. [19]. The design is comprised of five main parts: Program, interface, compiler, memory controller and ReRAM-based PIM.

3.5 Architectures for Nearest-Neighbor Upsampling Strategy

Figure 15(a) shows the up-sampling approach. After performing upsampling, CONV is performed between the upsampled ifmap and the kernel to obtain the ofmap. A common approach for up-sampling is zero-insertion. Among other methods, the nearest-neighbor based upsampling is widely used. The zero-insertion strategy leads to checkerboard artifacts in the generated images, whereas the nearest-neighbor strategy does not have this issue. Figure 15(b) presents both zero-insertion based and nearest-neighbor based up-sampling methods.

Yu et al. [37] propose an accelerator that can perform both traditional CONV and TrCONV with two up-sampling strategies (zero-insertion and nearest-neighbor based). They note that the upsampling need not be actually performed in the hardware since the upsampled ifmap can be deduced from the original ifmap. At the software-level, their technique eliminates the ineffectual operations and unifies the computational pattern of different CONVs. Authors also generate a

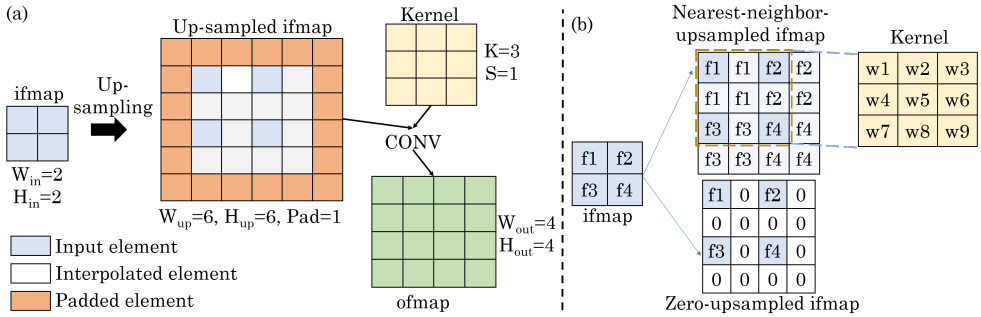

Fig. 15. (a) TrCONV process with padding (b) two upsampling strategies: zero-insertion and nearest-neighbor based [37]

compilation flow, which analyses the inefficiency in any TrCONV layer, and generates an efficient acceleration schedule based on the throughput optimization, computation reformulation, and scheduling. Based on the result of architecture parser, computation reformation decides the type of the CONV. The layers of the standard CONV are directly sent to the address constraint extraction. For TrCONV, up-sampling is done, but for nearest-neighbor strategy, up-sampling is preceded by kernel conversion. In the nearest-neighbor strategy, applying the kernel on a single ifmap window requires 9 multiplications and 8 additions. To reduce the number of multiplications, in the offline stage, they add up the kernel weights getting multiplied with the same ifmap value. For example, in Figure 15(b) (upper-part), $f1$ is getting multiplied with $w1$, $w2$, $w4$ and $w5$. Hence, instead of separately multiplying $f1$ with four values, a simple rearrangment can be done to reduce the number of operations. $f1$ can multiplied with $(w1 + w2 + w4 + w5)$, which is the new kernel weight $w1'$. Similar rearrangements can be done for obtaining the other weight values as well. This is referred to as kernel-conversion. Address constraint extraction transforms CONVs into the same computation pattern. The final reformulated network is sent to the scheduling optimizer, which fits the current network into the hardware architecture. It is also responsible for scheduling and network slicing required for the hardware mapping.

The overview of the accelerator is shown in the Figure 16. At the hardware level, it deals with channel level parallelism. It seeks to fit each layer in the hardware processor to improve the throughput, while accounting for the limitations of block-RAM. To achieve this, in each layer, they perform both channel slicing and feature map slicing. Each layer is sliced into blocks. They evaluate their technique on Xilinx Zynq FPGA. For the nearest neighbor up-sampling, their technique achieves 1.63× latency reduction and 12.43× better power efficiency compared with Titan Xp GPU. For zero-TCONV, their technique achieves 1.90× latency reduction and 15.04× better power efficiency compared with Titan Xp GPU.

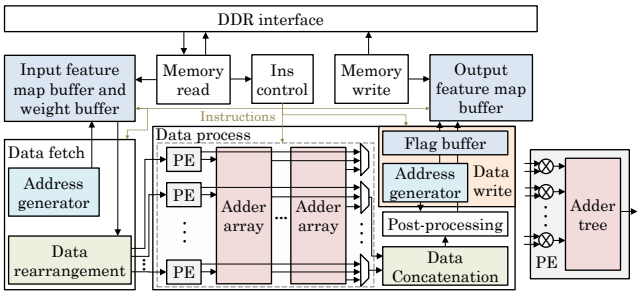

Fig. 16. The accelerator proposed by Yu et al. [37]. Ifmap buffer and kernel weight buffer uses ping-pong memory structure. Data process consists of a post-process block and a computational engine, data is fetuched and rearranged using the data fetch unit. Ins control sends instructions to all the blocks.

3.6 Architectures for Instance Normalization Layer

Apart from CONV and DeCONV, GANs also have "instance normalization" layers, which focus on instance-specific information. They are computed after CONV/DeCONV layers. Xu et al. [21] present a hardware accelerator that supports various operations of generative networks such as CONV, DeCONV, and especially instance normalization. Instance normalization comprises various complicated, time-consuming operations, such as computing the variance, which is square of standard deviation. Let the dataset be of size HW where W and H represent the weight and height of the ofmap. A naive computation of variance requires $3HW$ operations. They note that the standard deviation can be computed simply by computing the mean value, and the "root mean square" value of the output values. This simplification reduces the operation-count to just $HW - 1$, thus reducing the operation-count and memory latency to just one-third. They decompose the addition of output neurons into several MAC operations, which allows the reuse of the output of CONV and DeCONV operations and of the filter weights.

Since the computation of standard deviation and mean depends only on the output neurons, they use output stationary dataflow for computation of instance normalization. Here, the output neurons are pinned to specific PEs, and they are immediately used by the subsequent instance normalization layers. The output stationary dataflow can be used with either a single ofmap channel or multiple ofmap channels, depending on how many ofmap planes are processed at a time. Of these, they choose a single ofmap channel since it leverages the data-reuse between CONV/DeCONV and instance normalization. It keeps the data of one ofmap channel on-chip and exploits its reuse. By contrast, processing multiple ofmap channels would lead to exceeding the capacity of the on-chip buffer.

The architecture of the accelerator is divided into three domains: storage, computing, and control. The control domain comprises the controller, while the storage domain deals with DMA and global buffer. The computing domain comprises a 2D PE-array, adder tree, vector processing unit, scaler processing unit, and register file. The vector processing unit and the scaler processing unit are responsible for computing instance normalization layers by calculating shift-scale and mean-variance, respectively.

The "instance normalization layer" works in three steps: addition, calculation of the normalized parameters, and normalization. The addition includes element-wise squaring using a PE. It also includes the addition of the output neurons and the addition of the square of the output neurons. Calculation of the normalized parameters is composed of various operations such as division, square root operations, and subtraction. Normalization is performed parallely using a vector processing unit.

The CONV operations is shown in Figure 17(a). The dataflow for CONV operations is shown in Figure 17(b). In the first cycle, the weight is broadcast to PEs, computations are performed, and the results are stored in the PE register. In the next cycle, the neurons are shifted between neighboring PEs for data reuse. After 3×3 cycles, all the output neurons are computed.

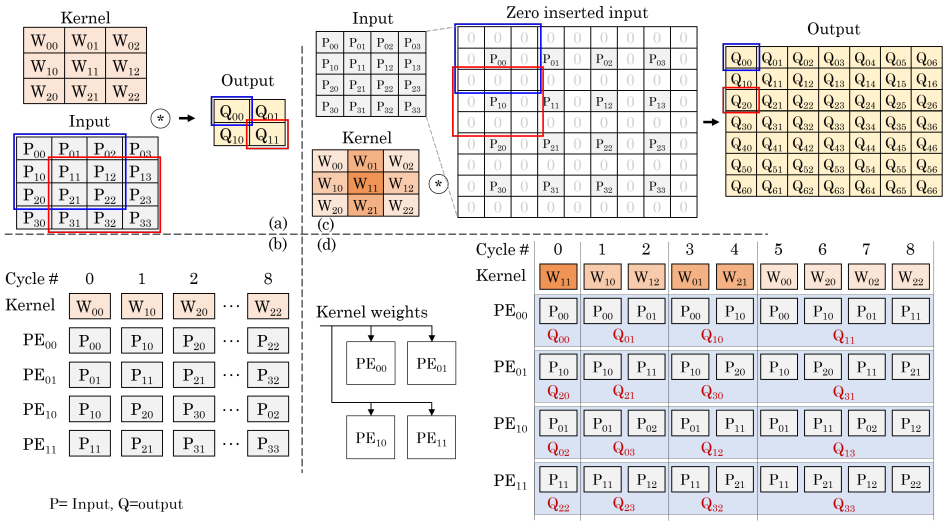

Fig. 17. (a) CONV operation (b) Dataflow for CONV operation (c) DeCONV operations and (d) Dataflow for DeCONV operation [21]

For performing the DeCONV operation, the weights are decomposed into four weight patterns, as shown in Figure 17(c). In the first cycle, the first pattern is broadcast, which leads to the computation of four output neurons. In the next two cycles, the next weight patterns are sent to PE serially, and the other output neurons are computed, as shown in Figure 17(d). The authors move the instance normalization layer out of the critical path resulting in increased performance. With the proposed methodology, all these operations except the scale and shift can be performed simultaneously with CONV/DeCONV operation. Thus, a two-stage pipeline between CONV/DeCONV and scale-shift is created. Their technique improves the speedup and power efficiency by 4.56 $\times$ and 29$\times$ respectively, as compared to prior baseline GAN accelerators.

3.7 Architectures for Dilated CONV

Im et. al [30] propose an accelerator for transposed and dilated CONVs. They deploy their accelerator for the ENet network [51], which is used for the image segmentation based on region-of-interest. They note that the segmentation accuracy is degraded by the "region-of-interest-based segmentation" as the pre-trained "dilation rates" are optimized for the original image resolution. To solve this, the authors propose an algorithm for "dilation rate adjustment" that regulates the "dilation rate" based on the resolution of the region-of-interest. The dilation rate controller implements this algorithm.

Their architecture is shown in the Figure 18. The aggregation core gathers all the partial results from the output memory and thus accumulates and manages the CONV results. The delay cells are used to skip the predicted virtual zeros. The top-controller uses a flip-flop-based frequency divider to create four frequency modes. The frequency of the main clock is divided by two, four, and eight to create different frequency modes. The best operating frequency mode is selected according to the region-of-interest resolution. This helps to lower power consumption. For example, the design requires

33 ms to segment a 400×400 image with 200 MHz frequency. When the image resolution is 128×128, the speed can be maintained even with a 25 MHz frequency. Thus, by decreasing the operating frequency, an 81.2 % reduction in the power consumption is observed. Thus, the dynamic frequency scaling is performed based on the image resolution.

The previously proposed accelerators skipped zeros but required workload balancing to improve the PE utilization. In contrast, the authors propose a simpler, highly reconfigurable delay cell logic to skip the redundant zeros. Both the dilated CONV and TrCONV insert zeros in the kernels and the ifmap. For the ENet, these zeros are predictable and are skipped using the delay cells, which are located between PEs. These delay cells do not latch the data for a simple CONV operation but only for the dilated CONV and TrCONV. In the case of the dilated CONV, the non-zero elements of the kernel are fetched. An element of the ifmap is fetched in the PE row. These are multiplied, and ifmap value gets latched in the delay cell for a fixed number of cycles before they can reach the next PE row. The number of cycles for the element's stay in the delay cell is decided according to the dilation rate. If the "dilation rate" is S, the element stays in the delay cell for $S - 1$ cycles, and then it is propagated to the next PE. When the first input element propagates all the way down to the last PE, all the partial sums are added up by the adder tree. The outputs are then transferred to the aggregation tree. In the end, all the values of the ofmap are computed, and the result is sent to the memory. The delay cells are also used to skip computation of the zero-padding in case of the dilated CONV. PEs in the same column is responsible for the computation of the 1×3 non-zero kernel elements for a 3×3 kernel. By not fetching the zeros of the ifmap, multiply-operation with the lower and upper padded zeros can be avoided. For the left and right padded zeros, the first, third, and fifth elements of the ifmap are fetched in the first cycle. These are multiplied with the first, second, and third elements of the kernel of size 3×3 respectively. In the next cycle, the element from the next row of ifmap is fetched.

In the case of the TrCONV, ifmap is added with the zeros. The ifmap with non-zero values is saved in the input memory. In this case, as the kernel values are non-zero, so the ifmap needs out to be latched. Here, Output features are latched in the delay cells before they can get added by the adder tree. This enhanced the throughput by 159× and 3.84× in dilated CONV and transposed CONV, respectively. Delay cells latch the input and can support variable CONV layers. Delay cell consists of a multiplexer and SIPO registers. The convolution configuration is decided by the multiplexers state table. Their design incurs 4.66 ms latency and achieves 3.22 TOPS/W throughput.

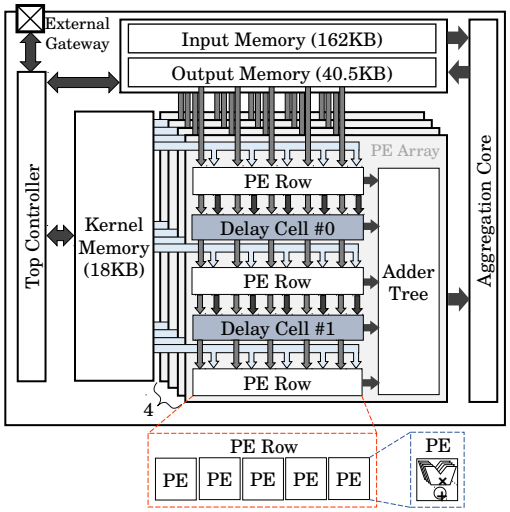

Fig. 18. The architecture proposed by Im et al. [30] consists of four PE arrays, a controller along with a "dilation rate" controller and a aggregation core. A PE array has five columns and three rows along with an adder tree and two delay cells. Aggregation core collects the partial sums and accumulates and manages the PE's computation result.

3.8 Architectures for 3D DeCONV

Wang et al. [22] present a design for accelerating both 3D and 2D DeCONV on FPGAs. The accelerator uses tiling to mitigate the limited on-chip memory issue of FPGA [52, 53]. Separate on-chip buffers are used for storing the output, input, and weight tiles. The compute engine has a T_m array of PEs. In every array, the PEs are arranged in a 3D mesh organization that has $T_n \times T_z$ PEs. All PEs are connected to the input buffer. In a row, only the leftmost PEs are connected to the weight buffer, and they gather the result of PEs which are located in the same row and supply them to the adder trees. These adder trees add the results of different ifmaps. Each PE has two register files to buffer the weight and input activations. It also has 3 overlap FIFOs which store the overlapping values. The non-overlapping values are sent to the local result FIFO. The multiplier results are added with the data in overlapping FIFOs. When the result of all input channels has been accumulated, the final ofmaps are sent to the DRAM memory.

They extend the "input-oriented mapping" of Yan et al. [35] to the 3D case, which is shown in Figure 19. The neighboring activations of the ifmap are mapped to the neighboring PEs. In a PE, every activation is multiplied with a kernel of dimension $K \times K \times K$ and produces a result tile of size $K \times K \times K$. The outputs are summed to the corresponding

position of ofmaps. The overlapped portions of neighboring tiles are sent to the PE, which processes a tile, and then element-wise summation is done. In every tile, the overlapping region has a length of $K - S$. The relation between O_H and I_H is $O_H = (I_H - 1) \times S + K$ and same for O_W, I_W, O_D and I_D.

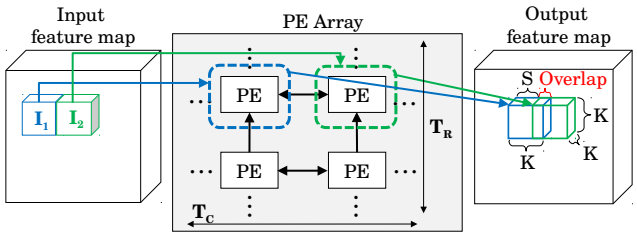

Fig. 19. 3D input-oriented mapping [22]. I1 to I3 are neighboring activations of ifmap. They are mapped to neighboring PEs, where they are multiplied with a 3D kernel and produce a 3D result. These results are aggregated over time. The overlapped results from the PEs that process I2 and I3 are sent to the PEs that process I1, and the pointwise summation is done.

Figure 20 shows the dataflow of a PE with an example. In step 0, the weights Wt(0,0,0,0,0) and activations In(0,0,0,0) to In(2,0,0,0) are loaded to the leftmost PEs for multiplication. The overlaps generated from $PE_{1,0}$-$PE_{2,0}$ are loaded to their overlapping vertical-FIFOs. In step 1, activations In(0,1,0,0) to In(2,1,0,0) are sent in the second column of PEs ($PE_{0,1}$-$PE_{2,1}$). Weight Wt(0,0,0,0,0) is also moved to these PEs and is multiplied with the activations. During this time, the leftmost PEs ($PE_{0,0}$-$PE_{2,0}$) perform multiplication with Wt(0,1,0,0,0). The overlaps generated by $PE_{0,1}$-$PE_{2,1}$ are sent to their horizontal-FIFOs and those generated by $PE_{1,0}$-$PE_{1,2}$ are sent to their vertical-FIFOs.

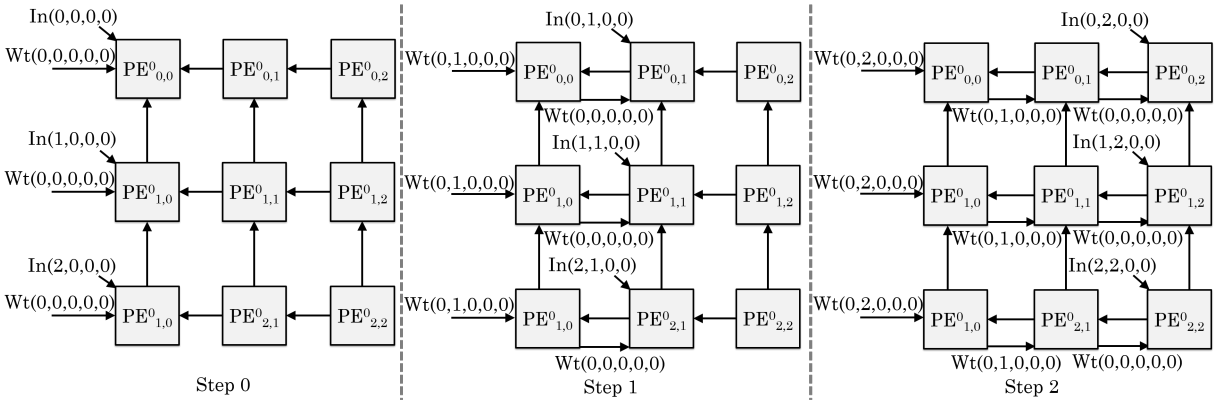

Fig. 20. Dataflow of a PE proposed by Wang et al. [22]

Their accelerator can support both 2D and 3D DeCONV operations. Implementing their technique on a VC709 FPGA platform provides up to 3 TOPS and achieves more than 90% resource utilization. The performance is higher for 3D DeCONV than for 2D DeCONV since 3D DeCONV has higher sparsity and a larger amount of data transfer. Also, their technique provides 22.7×-63.3× speedup over CPU and 3.3×-8.3× higher performance per watt than a GPU-based technique. For 2D DeCONV, GPU outperforms their technique, whereas, for the 3D DeCONV, their technique provides higher performance than the GPU.

3.9 Architectures for Training

Song et al. [20] present a hardware architecture for accelerating the training process of GANs. They highlight three key challenges. (1) The synchronization for loss calculation of a mini-batch restricts optimizations and requires a large amount of memory. The loss function that is commonly used for the discriminator, as well as for the generator, involves averaging over all the samples in a mini-batch. Therefore, the backward pass cannot begin until the loss for all the samples has been computed. Moreover, the intermediate outputs are required for computing the gradients. Therefore, they need to be buffered, which results in high memory requirements. (2) The large number of different computing phases involved in GAN training creates a tradeoff between a generic uniform design and a per-phase customized design. The discriminator update process has four computing phases: generator forward pass (G_f), discriminator forward pass (D_f), discriminator backward error propagation (D_{be}) and discriminator weight update (D_w). The generator update adds two more computing phases: generator backward error propagation (G_{be}) and generator weight update (G_w). With a large number of computing phases, a per-phase design becomes very costly to implement on a single chip due to large compute and memory resource requirements. Therefore, the tradeoff needs to be investigated to find an effective and cost-efficient solution. (3) The unconventional convolution operations involved in GAN training, i.e., strided convolution, transposed convolution, and convolution for weight update, make traditional CNN hardware accelerators inefficient.

To overcome the aforementioned challenges, they propose an algorithmic modification and a specialized hardware architecture. At the algorithm-level, they exploit the linear averaging in the loss function to defer the synchronization to the end of mini-batch processing, i.e., to the weight update stage, instead of the loss computation stage. This enables starting the backward error propagation phase of each sample right after its forward pass is complete, which significantly reduces the memory requirements for larger batch sizes. At the architecture-level, they propose two microarchitectures that are time-multiplexed to perform all the computing phases involved in each iteration of GAN training. Specifically, the zero-skipping output-stationary (ZeSOS) microarchitecture is designed for forward pass computations (i.e., D_f and G_f) and backward error propagation (i.e., D_{be} and G_{be}). Further, the zero-skipping weight-stationary (ZeSWS) microarchitecture is designed for updating the weights (i.e., D_w and G_w). As most of the computing phases are handled by the ZeSOS architecture, the pipeline bubbles are eliminated by adjusting the computing resources in the two microarchitectures.

The ZeSOS microarchitecture is shown in Figure 21(a). It is a unified architecture designed to perform strided and transposed convolutions and thereby handles the complete forward pass and the backward error propagation. It is based on the output-stationary dataflow. ZeSOS is composed of an inpute register file and a PE array. The PE array contains 4×4 PEs, where each PE has a MAC unit for the computation of partial sum and a register to store it for the subsequent MAC operation. The input activations are fed to the PEs through the register file that is composed of 4x6 registers. The registers highlighted in gray in Figure 21(a) are connected to their corresponding PEs in the array, e.g., $R_{1,1}$ is connected to $PE_{1,1}$ and $R_{1,2}$ is connected to $PE_{1,2}$. The additional register columns are used to enable temporal reuse of the fetched activations. The weights are broadcasted sequentially in a "type-oriented manner" to the PE array, where they are shared spatially by all the PEs. The type-orientation here refers to the arrangement of values on the basis of their column and row indexes being odd or even in the corresponding tensor. The values that belong to even rows and even columns are placed in the even-even category, while others are arranged in "odd-even", "odd-odd" and "even-odd" categories, based on their respective row and column indexes. The weights belonging to the even-even category are fed to the PE array first, followed by the ones belonging to odd-even, odd-odd, and even-odd categories for computing the outputs. This type-orientation-based scheduling helps avoid multiplication with zero operations by skipping the categories that have only zeros and thus, enables efficient implementation of strided and transposed convolutions.

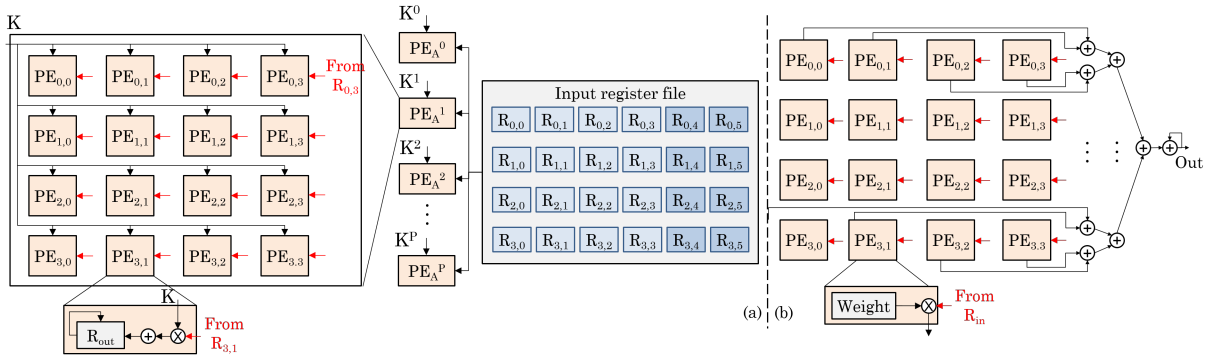

Fig. 21. (a) Zero-skipping output-stationary (ZeSOS) microarchitecture. (b) Zero-skipping weight-stationary (ZeSWS) microarchitecture. [20]

The ZeSWS microarchitecture is shown in Figure 21(b). It is designed to handle the convolutions for weight update in the discriminator as well as in the generator, i.e., D_w and G_w. The architecture is based on weight stationary dataflow. For computations using ZeSWS, the loops related to kernel weights are unrolled and mapped on the PEs, where one weight is mapped on one PE at a time. All the PEs perform computations related to a particular output at a time, and the results are accumulated using the adder tree. To exploit temporal reuse of the input values, ZeSWS has a similar input register file as ZeSOS architecture. Therefore, it supports similar dataflows as ZeSOS, which helps in skipping zeros inserted in the kernels of D_w and in the input data of G_w.

Figure 22 shows the complete accelerator design. Note that the architecture uses multiple instances of ZeSOS and ZeSWS microarchitectures to speedup the computations. For ZeSOS, the loop related to output feature maps is unrolled so that the same register file can be shared by all the ZeSOS PE arrays. Similarly, for ZeSWS, the loop related to the number of filters is unrolled. Their accelerator contains four types of buffers. The input&output buffer is responsible for storing the inputs and outputs of the ZeSOS architecture. The Data and Error buffers are responsible for storing all the intermediate outputs and errors that are required for computing the weight update. The weight buffer is required to store partial ΔW values for the ZeSWS microarchitecture and weights for the ZeSOS microarchitecture. Note that the number of PEs in a single block of both the microarchitectures is defined to be 4×4, based on the size of the smallest ofmap and the minimum kernel size in DCGAN [54]. Moreover, to avoid bubbles in the pipeline, the number of ZeSOS blocks is set to be $2.5\times$ the number of ZeSWS blocks.

As for conventional architecture, they find that using output stationary for D_w and G_w and no-local-reuse for the remaining computations provides the best performance. Their ZeSOS-ZeSWS architecture provides an even higher speedup than this. They implement their design on a VCU118 board which has an Ultrascale+ XCVU9P FPGA. Compared to i7-6850K, their design has an $8.3\times$ speedup and $45\times$ better energy-efficiency. Also, it achieves $5.2\times$ energy-efficiency

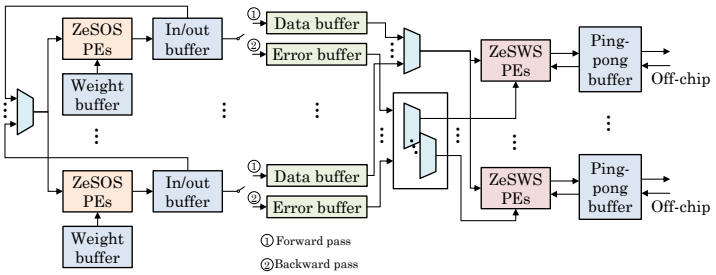

Fig. 22. Overall design of Song et al. [20]. ZeSWS computes D_w and G_w. ZeSOS does all the other computations. Both ZeSWS and ZeSOS have 4×4 PEs. The following four types of buffers are used. The first layer's data are loaded into one of the in/out buffers. The output of ZeSOS is stored in another in/out buffer, which becomes the input buffer for the next layer. D_w and G_w require data from the FWP and error from BWP, and they are stored in the data buffer and error buffer, respectively. The computation of ∇W is done by ZeSWS. When the kernel size is greater than the size of unrolled kernel weights, partial results are generated for ∇W. They are stored in ∇W buffer.

improvement over K20 GPU.

Hanif et al. [40] propose an on-chip memory architecture for efficient utilization of ZeSOS microarchitecture proposed by Song et al. [20]. The ZeSOS microarchitecture is based on output stationary dataflow. It is responsible for performing strided and transposed convolution operations, which are the two key operations involved in the GAN training process. For processing using ZeSOS, data is fed to the PE array in a type-oriented format, i.e., computations related to the even-even category are performed first, followed by computations related to even-odd, odd-even, and finally odd-odd category. This type-orientation-based processing allows to easily skip multiplication with zero operations and thereby improves the computational efficiency.

To operate the PE array at its full potential, it requires a maximum of twenty-four data points to be stored in the register file in a type-oriented format in a single clock cycle from the on-chip memory. The conventional on-chip memory stores data in a linear format and, therefore, cannot provide multiple data points in a type-oriented format without fetching unnecessary data. To overcome this challenge, Hanif et al. [40] propose a distributed memory architecture with scratchpad memories, each having a single port. Figure 23 shows the distributed memory architecture where the scratchpad memories are arranged in a 2D grid architecture. The architecture is divided into four blocks and four channels depending on the number of type-orientations and the number of rows in the register file, respectively. Each block-channel tile contains six scratchpad memories depending on the column-count of the register file. The even-even input data is stored in Block 0 of the memory, where the six-element rows are distributed in a raster scan order along the channels. Similarly, the "even-odd", "odd-even", and "odd-odd" data are placed in blocks one, two, and three, respectively. Each scratchpad memory is capable of providing a single data element per clock cycle. Therefore, 24 data elements can be fetched from all the scratchpad memories in a single Block.

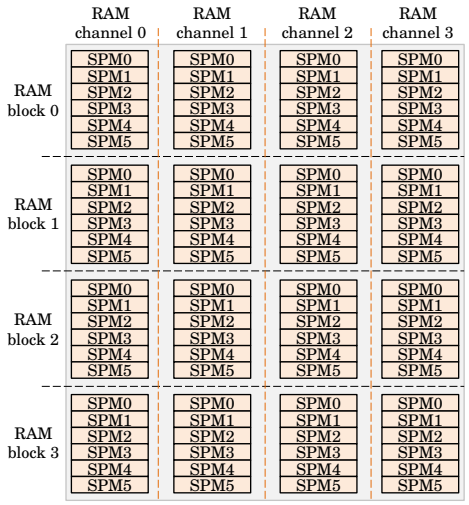

Fig. 23. Architecture of distributed on-chip memory in the technique of Hanif et al. [40]

They also propose a data arrangement controller for arranging the data in the required format inside the on-chip memory while fetching it from the DRAM or while receiving it from the PE array. The controller is responsible for computing the on-chip memory addresses for the data. For this, it computes input pixel row and column indexes using the DRAM address of the pixel, the input image dimensions, the number of pixels stored per DRAM location, and the pixel-index in the DRAM. The available information is then used to compute block and channel indexes, scratchpad memory index inside the tile, and finally, the scratchpad memory address where the data should be stored. They implement their

architecture on a Kintex-7 FPGA. Compared to a previous work [20], their technique provides 3.6× higher performance and about 85% decrement in the read accesses and 75% decrement in the write accesses.

Roohi et al. [34] propose an optimized training algorithm for GAN using binary weights. In the training phase, the first statistical weight scaling and binarization are done. In binarization, sign bits of the full precision weight are considered, and then, based on the statistical distribution, the scaling factor is computed. This leads to a vanishing gradient problem. To avoid this and to achieve a good binarization, binarized representation entropy regularization [55] is used. This is followed by binary weight-based inference for the loss computation. Finally, backpropagation is done for updating the full precision weights. Also, to minimize the loss, static gradient descent is considered. The same steps are followed in the next iterations. Layers that have a higher degree of redundancy are binarized.

The authors design a reconfigurable addition method that uses both precise and approximate addition operations. Figure 24(a) shows the binary CONV in their technique. In this first step, four channeled input is convolved with binary weights, and the output batch is mapped to a sub-array. After this, addition/subtraction is done using a inexact computing unit [56], which consists of an inexact adder and an exact adder. To reduce error and increase accuracy, MSB bits are computed using the exact adder, while the LSB bits are computed using the inexact adder. This is shown in Figure 24(b).

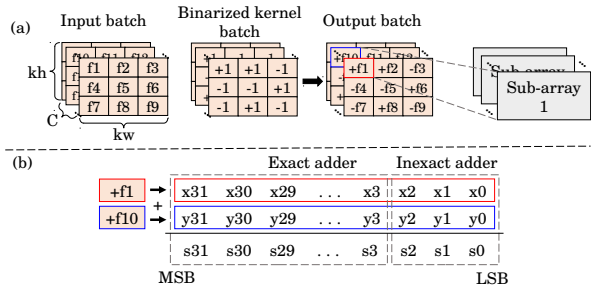

Fig. 24. (a) Binary CONV operation [34]. c channels of size $kh \times kw$ from the input batch generates the combined batch with respect to the kernel batch. (b) partial inexact computing, which computes 3 LSBs using an imprecise adder and remaining bits of the 32b ofmap using a precise adder.

Their in-memory accelerator is based on a memristive computational sub-array, and it is presented in Figure 25. The original ifmaps and weights are stored in kernel and image sub-arrays, respectively. These are distributed across memory-banks. The external processing unit comprises five computational sub-modules. The memristive computational sub-array can perform bit-width operations which are parallel and flexible. In step (1), the binarizer performs binary kernel processing by altering the sign-bit of input with respect to the kernel data. Thereafter in step (2), the transpose of the channels of the combined batch is computed. Moreover, the transposed batch is mapped to the sub-arrays. In step (3), the computational subarray performs parallel processing, then the batches are processed using EPU's shared components. Finally, in step (4), the final ofmap is generated for the next layer. A parallel in-memory circuit is designed to accelerate the multi-bit operations.

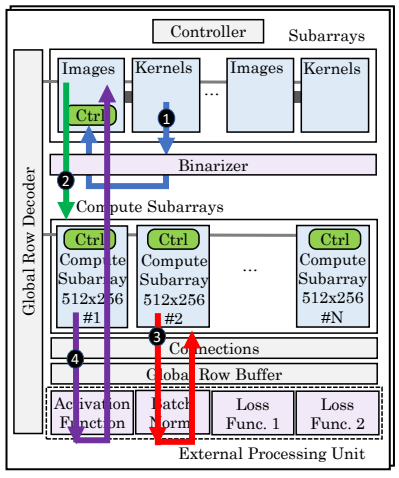

Fig. 25. The in-memory accelerator proposed by Roohi et al. [34].

To further boost the performance, the authors extend the spatial parallelism method in [16] to design a fully-pipelined computation mechanism. During training, the input data is processed in the batches of 8/32/64. In fully-pipelined computation, to achieve pipelining, data is duplicated for the intermediate layers so that the data is readily available. Moreover, the spatial parallelism method [16] shows that there is no dependence among the training phases of the discriminator. Hence, they can be executed in parallel. The two training phases utilize separate sub-arrays. The DeCONV and CONV layers can execute simultaneously. Experiments showed that their technique provides 5.1× speedup and 2.5×

higher energy efficiency than CMOS-ASIC accelerators [57]. However, it degrades the inception score by 11%.

4 Optimization Techniques

Table 4 highlights the optimization techniques used by different works.

TABLE 4
A classification of optimization techniques

Category	References
Algorithm-level	
CONV style/implementation	Winograd [32, 42], Fermat number transform [33], fast FIR scheme [43]
Computing approach	casting CONV into a matrix multiplication operation [46] or a matrix-vector multiplication operation [19], Converting DeCONV to CONV [32, 43, 44], partition every weight filter into 4 smaller filters [29, 42], skipping up-sampling operation by pre-adding the weights [37]
Others	regulating the dilation rate based on RoI resolution [30], delyaing the synchronization to the end of mini-batch [20], use of instance normalization layer [21], layer-fusion [45]
Hardware-level	
Tiling	[19, 21–23, 26, 32, 35, 40, 42, 45, 46, 48]
Loop unrolling	[33, 46, 48]
Improving parallelism	Partitioning the memory into multiple segments for allowing concurrent accesses [42], duplicating weights to improve parallelism [19], duplicating the data of intermediate layers for achieving full-pipelining [34]
Parallelism between	generator and discriminator [17], CONV and DeCONV layer [34]
Dataflow	output stationary [20, 21], weight stationary [20]
Load-balancing	[23, 31, 35, 45]
Lowering compute overhead	approximate adder [34], double MAC scheme for simultaneously computing two narrow multiplications on a single wide DSP [45], reducing multiplications at the cost of extra additions [43], pruning [23, 58]
Power saving	clock gating [33], dynamic frequency scaling [30], reducing ADC overhead by realizing it through an inverter [27], avoiding SET operations in MLC ReRAM [19]
Double-buffering	[29, 33, 37, 39]
Low-precision	Ternary GAN [18], binary GAN [34], different bit-widths in inference and training [15], adaptive precision [35, 36, 48], quantization [15, 22, 28, 36, 37, 45–47]
Pipelining	[16–18, 21, 29, 33, 34, 42, 48]
Avoiding computations on zero-operands	[8, 19, 21, 27, 30–32, 37, 43]
Others	use of spatial parallelism idea [16, 18], Huffman encoding [39], systolic array architecture [29]

We now review the optimization techniques used by different works. Sections 4.1 reviews tiling and unrolling, whereas Section 4.2 discuss use of parallelism. Sections 4.3 and 4.4 review Winograd transform-based and Fermat transform-based CONV operations, respectively. Section 4.5 reviews sparsity-related techniques such as pruning, avoiding ineffectual operations and load-balancing. Finally, Sections 4.6 and 4.7 review techniques for handling overlapping sum problem and irregular memory accesses, respectively.

4.1 Using Tiling and Unrolling

Liu et al. [26] design a hardware accelerator for DeCONV algorithm. They use the DeCONV style proposed by Zhang et al. [48] which works in four phases (i) multiplying one input pixel with the weight matrix (ii) adding the results of phase (i) in regions of overlap (iii) repeating (i) and (ii) for all the input items (iv) cropping. In the proposed accelerator, depending on the values of s and k, a suitable sized register is used for buffering the overlapped data for performing addition. In every cycle, a data-item is supplied to the k multipliers, where it is multiplied with one column of the weight matrix. The outputs are sent to a row of registers and shifted in each cycle. For handling the column overlap, the multiplier results are summed with the data in the last column of the registers before it is fed to the 'partial result buffer'. The outputs generated from the $k - s$ adders are summed with the partial outputs before performing accumulation. Thus, the difference from a CONV accelerator is that their accelerator uses a register array for adding the column overlaps and 'partial result buffers' for adding the row overlaps. The accelerator can be instantiated for different values of s, p, and k, where k, s, and p are the kernel size, stride, and the padding for a given layer, respectively.

In DeCONV, crop operation is required for removing the undesired border pixels and, thus, produce correctly-shaped output. They implement a hardware-level crop-unit that takes the result of the DeCONV kernel as input. Its design is presented in Figure 26(a). The location of the pixel in the output shape is found from two counters, viz., 'line counter' and 'row-counter'. These counters are compared with their respective thresholds for ascertaining whether a pixel is to be retained or cropped (removed). The thresholds can be set at runtime for achieving cropping of arbitrary-shape.

To mitigate the memory-limitations of their accelerator, they use tiling. For this, the ifmaps of dimensions $C \times H \times W$ are split into tiles of size $C \times T_H \times T_W$. Here, C and H/W are the number of channels and the height/width, respectively. In the conventional approach, shown in Figure 26(b), all the tiles of a layer are processed sequentially before moving on to the next layer. Here, the ifmap 'A' is split into 9 tiles, and every tile is accessed sequentially for generating the ofmap 'B'. For the second layer, the ifmap 'B' is split into four tiles since ifmap has fewer channels than 'A'. This tiling scheme is

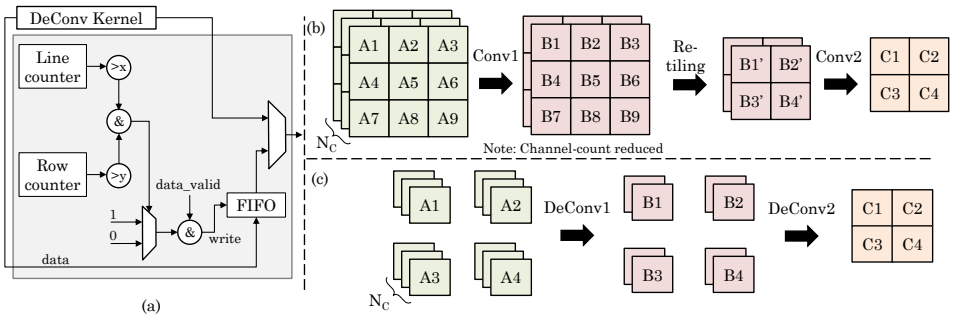

Fig. 26. (a) The crop module [26] (here, x and y are the thresholds) (b) traditional tiling approach for CONV and DeCONV where all the tiles of a layer are processed sequentially, before moving on to the next layer. (c) Proposed tiling approach for DeCONV [26], where every tile is processed till the last layer to obtain its output tile.

necessary in CONV because the result of the second CONV layer depends on the content of multiple tiles. For instance, the result of 'C1' depends on 'B1' and partial data of 'B2' to 'B4'. The conventional tiling approach allows adapting the tile size for different layers but requires moving a large amount of intermediate data to off-chip.

To avoid these issues, they propose an approach where the biggest tile-size is ascertained first. Thereafter, the total number of tiles is decided for each layer, and each tile is processed till the last layer for obtaining its output tile. Figure 26(c) shows their proposed tiling approach. Different tiles are processed serially, and thus, the network can be seen as a group of independent sub-networks. In other words, the removal of dependence between different sub-networks reduces the need for off-chip memory accesses. The interim data is stored on-chip, and off-chip access is required only for the input, weights, and output data. Thus, off-chip accesses are reduced at the cost of a higher requirement of on-chip memory. Their design uses two on-chip buffers, which work in a ping-pong manner for achieving double-buffering. They implement their accelerator on an XC7Z045 FPGA. Compared to the accelerator that uses conventional tiling, their proposed accelerator has a 2.3× speedup. Also, their technique provides 30× to 90× speedup compared to i7-950 CPU and 8× to 108× improvement in energy efficiency over Titan X GPU.

Zhang et al. [48] propose an approach which allows using a CONV accelerator for executing DeCONV operation. They note that summing up the overlapped regions leads to overhead and is especially inefficient on FPGAs. This is because FPGAs require separate hardware blocks to create overhead or to communicate with the host processor. This ultimately increases the latency of the system.

To avoid it, their technique finds which input blocks need to be deconvolved for obtaining different output blocks. Let the size of input be $I_h \times I_w$, and that of output be $O_h \times O_w$. In the baseline implementation, there are loops on I_h and I_w. They recast these loops to those on O_h and O_w. They also examine the impact of different bitwidths on the inference quality of the generative network and, from this, find the optimal bitwidth. Further, they perform roofline analysis to increase the throughput by intelligently choosing the width/height and channel size of the output and input tile. They perform loop unrolling and pipelining to improve concurrency. Finally, they insert registers for improving local memory bandwidth. They implement their design on FPGA, although it can also be implemented on an ASIC. Their technique achieves high-performance density. Also, the visual quality of generated images with 12b is similar to that with 32b.

Bai et al. [46] propose an unified hardware architecture for the CONV and DeCONV operations. To optimize CONV operation, the innermost kernel loop is fully unrolled. Also, to reduce the number of data transfers and partial sums, partial unrolling of the outer loops of the input and output channel is done. This also reduces the number of multipliers used in the hardware implementation. Loop tiling decides the partition of the fmap. It is done on the depth of the ifmap. The partition size ultimately decides the on-chip memory size.

The basic steps of the DeCONV process consist of padding the ifmap, followed by applying CONV on the padded ifmap. In this method, multiplication with zeros leads to ineffectual operations. To overcome this limitation, the authors propose that after padding the ifmap, scanning of the ifmap should be done using a 2×2 sliding window. This is followed by the DeCONV operation for each patch, as presented in Figure 27.

The hardware accelerator comprises a line buffer that reorganizes the input image, performs zero-padding required for the CONV, and converts the CONV into a matrix multiplication operation. Only a part of the ifmap is loaded into the ifmap buffer to utilize the on-chip memory efficiently. This is followed by a PE array, where each PE has an adder tree and a nine-multiplier array. The PE multiplies the input image with the kernel weights. This array structure is reused for both CONV and DeCONV operations. Thereafter, batch-normalization, activation, and max-pooling are done. In the end, the generated ofmap is stored in the ofmap buffer. The authors implement this architecture in ZC706 FPGA and achieve 151.5 and 94.3 GOPS for CONV and DeCONV, respectively.

4.2 Exploiting Parallelism

Hsiao et al. [39] propose an accelerator that supports various operations such as LSTM, CONV, DeCONV, depthwise CONV, point-wise CONV, fully-connected, pooling, and batch normalization. Their design exploits four types of parallelism. The

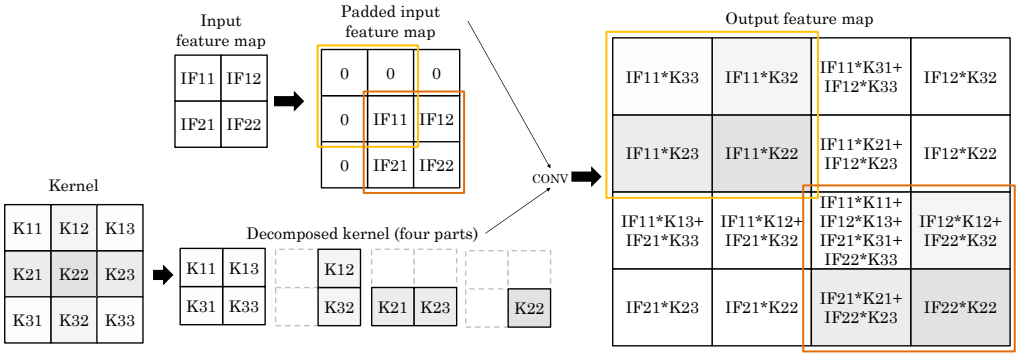

Fig. 27. Optimization of DeCONV [46] for a 2×2 feature map. Most of the redundant multiplication can be avoided.

depthwise CONV exploits output window parallelism and kernel weight parallelism. The first layer of CONV utilizes output channel parallelism and kernel weight parallelism. The CONV layers are other than the first layer leverage output channel parallelism and input channel parallelism.

The architecture is shown in Figure 28(a). To reduce the energy and the data transfer time, 2-symbol Huffman coding is used. The encoder consists of zero detectors, an index buffer that stores the flag of non-zero/zero bit, and 8 FIFOs. The non-zero data is moved to the FIFO, and the non-zero/zero flag is recorded in the index buffer. Each PE is composed of a multiply-adder tree with 8 multipliers followed by the adder. The index buffer in the encoder-decoder unit reduces the data computation power for zero data, which disables zero data multiplication. The extra processing unit is responsible for the computation of non-linear functions required by the LSTM. To implement the streaming process, double buffering is used. Figure 28(b) shows the overall functional block of the architecture. Various blocks are activated according to the type of operation. During LSTM computation, line buffer, temp buffer, batch normalization layer, activation, and pooling layer are bypassed. For the convolution operation, the line buffer is bypassed. Input data and weight are loaded in the SRAM_A and SRAM_B, respectively. In order to reduce the SRAM access time, the input is also stored in the temp buffer. This helps in accessing the overlapped inputs in the case of the 3×3 convolution. For the point-wise convolution, there is no issue of overlapping. Therefore, both the line and the temp buffers are bypassed, and the data is directly fed into the PE. For the deconvolution operation, only the line buffer and LSTM block are bypassed. Moreover, input data from the SRAM_A are fed directly to PE's multiple-adder tree. The output is saved in the temp buffer, which then sends the inputs to the accumulator. For the fully connected layer, line buffer, pooling, LSTM block, and temp buffer are bypassed. Their proposed technique reduces the power consumption of arithmetic computations and memory accesses and also improves performance.

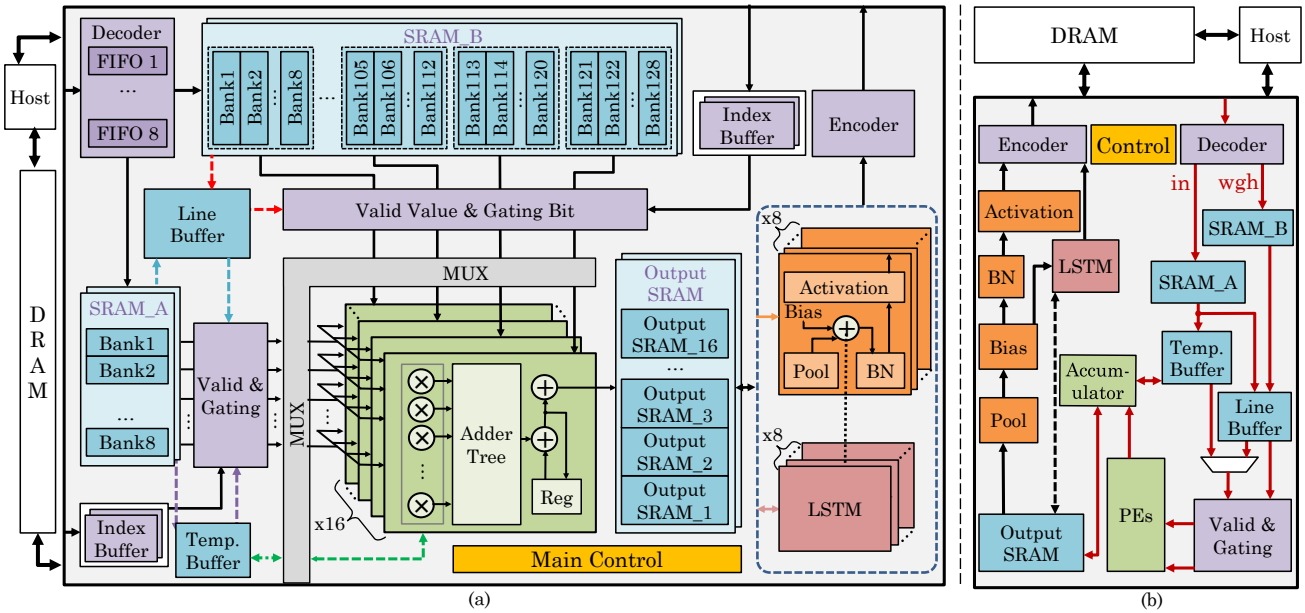

Fig. 28. (a) The overall architecture of Hsiao et al. [39]. The two SRAM supports streaming process with external memory access and overlapped computations. (b) functional block diagram. [39]. Blocks are activated according to type of operation.

Perri et al. [38] propose an efficient hardware accelerator for 2D DeCONV. The accelerator's top-level architecture is presented in Figure 29(a). Figure 29(b) shows the design of their accelerator. It parallely operates on T_N ifmaps and T_M

kernels to produce T_M ofmaps. If the size of the ifmap or kernel exceeds these limits, then the computation happens in multiple iterations. The kernel buffer has a register file for storing the coefficients. In every cycle, it receives the corresponding ifmap values packed in a single word. This reduces the latency of uploading the kernel coefficients processed by the DeCONV engine.

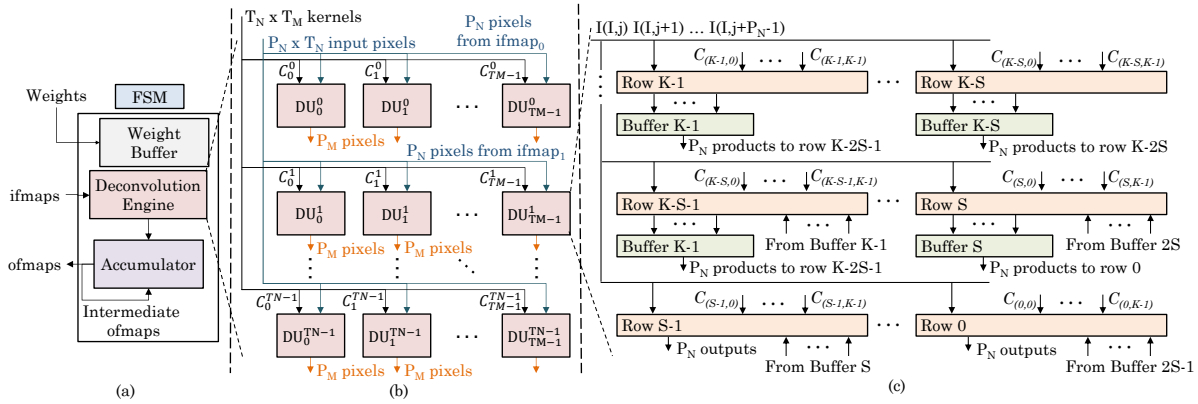

Fig. 29. (a) PE used for DeCONV layer [38] (b) Architecture of DeCONV engine (c) Architecture of DeCONV Unit (DU)

The distribution logic supplies the coefficients to the DeCONV engine. The DeCONV engine has $T_N \times T_M$ DeCONV units working in parallel. Every DeCONV unit deconvolves multiple nearby input pixels with the corresponding kernel coefficients for computing the ofmap. For managing the overlapping rows/columns between the nearby blocks of products, the DUs are architected, as presented in Figure 29(c). The total number of DSP units in a row decides the degree of parallelism. FIFO buffers are used for aligning the overlapping products. They did an FPGA implementation of the proposed work. It reduces the resource and power consumption while enhancing the computational speed compared to state-of-the-art architectures.

Shi et al. [10] propose a generative model engine for performing edge computing. It is used for executing both the transposed and dilated CONVs by decomposing the kernels. The algorithm consists of the initial decomposition of the kernel, followed by untangling the kernels and input matrix multiplication. In the end, the results are dispatched and combined in the output tensor. Their technique reduces both memory access latency and computation overhead. When the input tensor is zero inserted with stride 2, the kernel of a transposed CONV can be decomposed into four possible patterns such that non-zero elements of kernel meet non-zero elements of the tensor, as shown in Figure 30. This helps in learning the relation between the indices of non-zero elements of the fmap and the decomposed kernel. This allows easy removal of zeros. This leads to smaller standard CONVs on the input tensors.

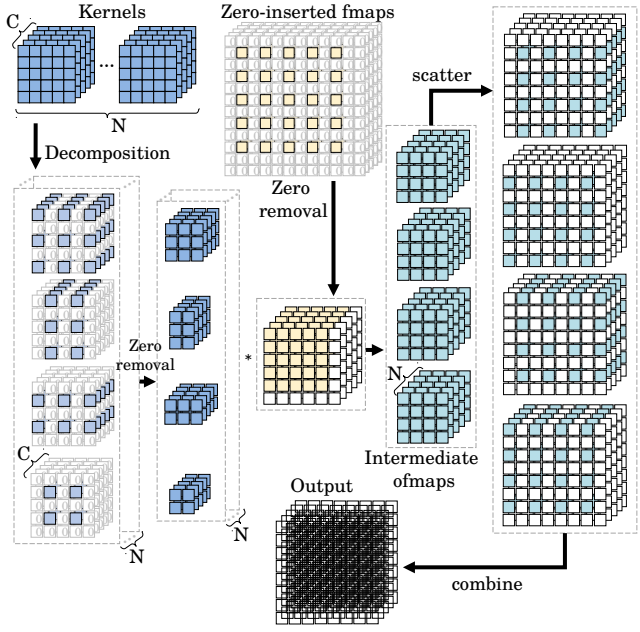

Fig. 30. Kernel decomposition in the technique of Shi et al. [10]

Furthermore, the decomposed TrCONV patterns are untangled into 1×1 CONVs. This improves the arithmetic computations' parallelism. The authors propose a strategy to untangle the decomposed pattern to a set of 1×1 CONV. For

standard CONV, untangling is done, as shown in Figure 31. Let us consider zero removed N decomposed kernels of size $m \times n \times C$. These kernels are regrouped by accumulating N columns along the dimension C from each kernel position. This forms a new matrix of size $N \times C$. Corresponding receptive field of size $(H - m + 1)(W - n + 1) \times C$ is fetched from the input tensor of size $H \times W \times C$. This configuration is considered as a 1×1 CONV with 'N' 1×1 kernels. Then, the results are accumulated and sent to the corresponding positions in the output tensor. Similarly, the dilated CONV is also untangled with a larger sliding step on the input tensor, and the receptive field reduces with the strides' multiple. Moreover, untangling the transpose kernel improves parallelism.

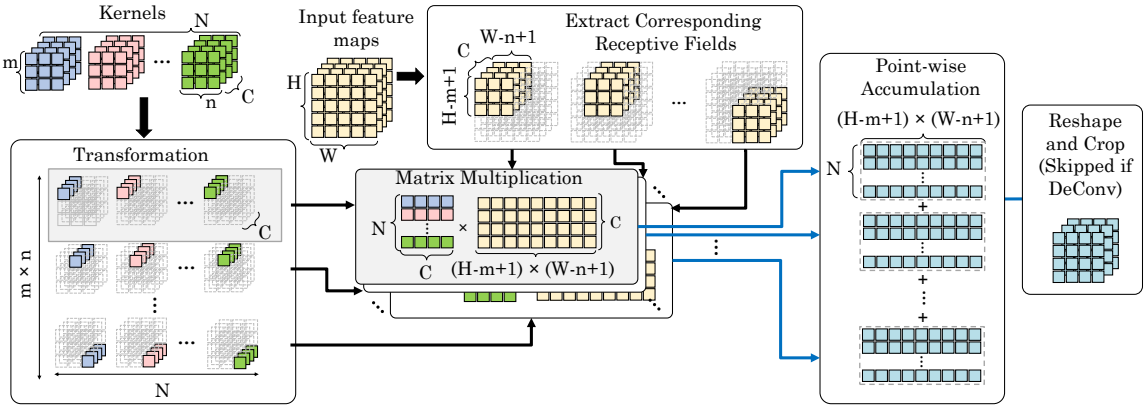

Fig. 31. Untangling a standard CONV into a set of 1x1 CONV [10]

Xu et al. [44] propose a technique for efficiently implementing DeCONV on the existing processors. DeCONV is converted to independent CONV operations by first expanding the size of the filter if it is not divisible by the stride. The DeCONV filters are expanded by inserting zeros in the left and top sides of the original filter. This ensures that the kernel size remains the same when the DeCONV is converted to the multiple CONV operations. Thereafter, the expanded DeCONV filter is split into smaller CONV filters by performing 180-degree rotation and sampling. This ensures that the computation is correct. Moreover, these steps are done in the offline stage with the software approach. This process is done once, and the resultant smaller CONV filters are reused. Each DeCONV is divided into s^2 CONV operations, where s is the stride. This is known as the split CONV. Split CONV is done with the stride of 1.

This is followed by the online stage performed on the CNN processor, where the padded input convolves with the smaller filters. Padding is done in the ifmap so that boundry pixels are also included in the computation. Finally, the generated outputs from the split CONVs are reorganized to form the final deconvolved output. To enhance the efficiency of the output reorganization, CONV output data is stored in the channel-major layout. On commodity neural processors, their technique achieves higher performance than the conventional zero-padding DeCONV.

4.3 Using Winograd-based CONV

Chang et al. [32] propose a Winograd-based DeCONV accelerator. Their overall approach is shown in Figure 32(a). First, using the "Transforming DeCONV to CONV" method, a $M \times N$ DeCONV layer is converted to a $s^2 \times M \times N$ CONV layer, where M and N are the numbers of ofmaps and ifmaps, respectively. These s^2 convolutional filters and input tiles are transformed to the 'Winograd domain' by using transform matrices 'G' and 'B', respectively.

The Winograd algorithm transforms the 2D CONVs into the element-wise multiplication. The element-wise multiplication is done on the transformed tiles and the filter except for the positions where filter weights are zero. A channel-wise summation follows this. The inverse transform is done with the help of the transformation matrix 'A'. Each filter generates a $S \times S$ output block, and simultaneously it also generates a $m \times m$ output tile, thus generating a $mS \times mS$ output block.

Let the size of the input tile be $n \times n$, that of the output tile be $m \times m$, and that of the filter be $r \times r$. Then, the spatial CONV requires $m^2 \times n^2$ multiplications, whereas the Winograd method requires only n^2 multiplications. The authors also explore the dataflow optimization to enhance efficiency and exploit the sparse multiplication pattern. The Winograd dataflow is shown in Figure 33. Each transformed filter has zeros in different ratios. Input tiles and Winograd filters are reorganized into $n^2 \times N$ sized matrix and M matrices of size $n^2 \times N$, respectively. If the filters have a vector-level sparsity pattern, performance can be improved by reducing the accelerator's idle cycles and skipping the zero-valued multiplications. For the filter having no sparsity, no performance benefit is obtained.

The hardware architecture consists of several PE, input line, and output line buffers as shown in Figure 32(b). On-chip memory cannot store all the intermediate outputs and fmaps. To solve this problem, they use a line buffer. It also helps in overlapping the data transfer time between the PEs and the computation latency between filters and the inputs. The architecture consists of a pre-PE, compute-PE, and post-PE. The pre-PE first collects and then changes the input tiles. It also rearranges them with the transformed filters. These reordered inputs and filters are moved to the compute-PE in the acceleration engine, from where they are moved to the post-PE. In the post-PE, these outputs are again transformed into the spatial domain using 'sparse inverse transform'. The authors show that the proposed accelerator achieves 1.78 $\times \sim$ 8.38$\times$

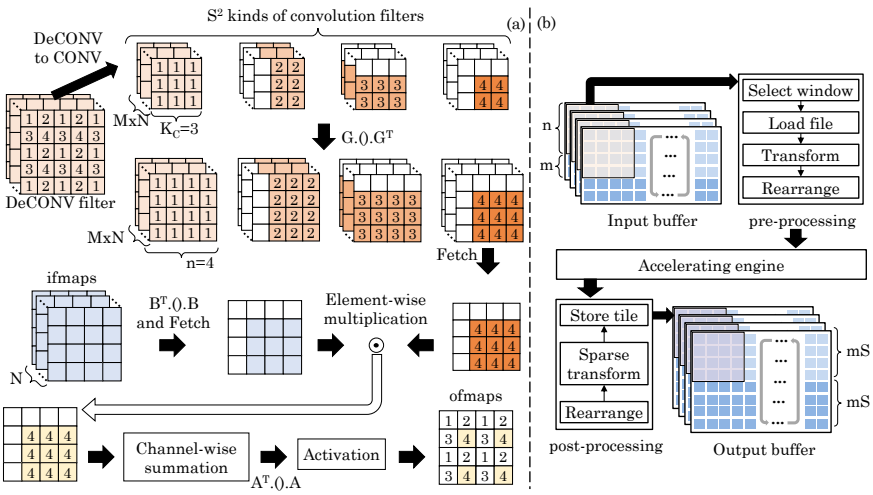

Fig. 32. Block diagram of the accelerator proposed by Chang et al. [32]. In 2D Winograd CONV, $Output = A^T[(GfG^T) \odot (B^T IB)]A$. Here, A, B and G are the transformation matrices and $\odot$ shows the element-wise multiplication. f and I show filter and input, respectively.

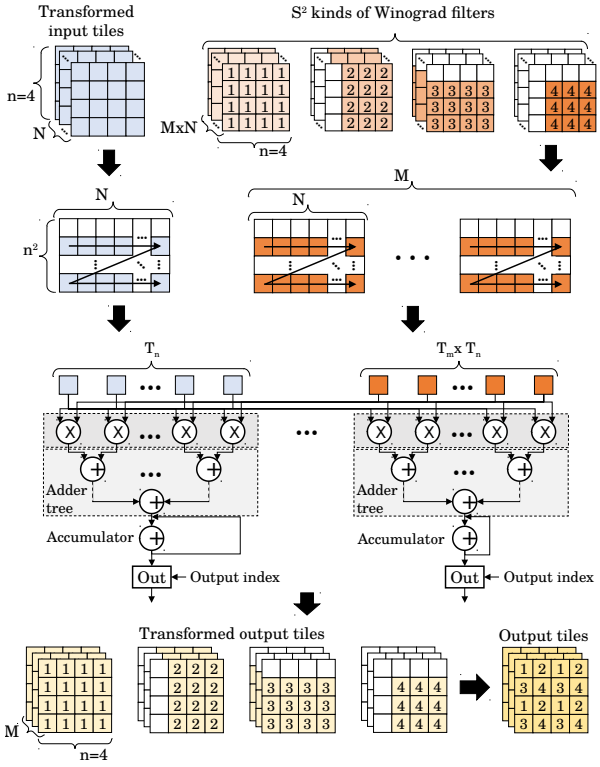

Fig. 33. Overview of Winograd DeCONV dataflow. [32]. The input tiles and the Winograd filters are reorganized into a $n^2 \times N$ sized matrix and M matrices of size $n^2 \times N$ respectively. The reordered filters have a vector-level sparsity.

higher throughput as compared to other accelerators. An improvement of 1.74 $\times$ in energy consumption is observed as compared to the DeCONV methods based on the TDC [25].

Di et al. [42] present an architecture for implementing the TrCONV on an FPGA. They note that the CONV of a 5×5 filter with 6×6 padded input can be broken into a CONV with 4 sub-filters, which have a different number of non-zero elements. They rearrange these sub-filters into 3×3 shape, and thus, one TrCONV with 5×5 filter can be split into four CONV operations between 3×3 filters and suitably padded ifmap. Notably, all the ineffectual operations are removed. This operation is termed decomposition. Then, these four CONV operations are implemented using the Winograd fast algorithm. Reduction of the sub-filter size to 3×3 ensures that the Winograd fast algorithm can be used.

Figure 34 shows the overview of their proposed technique. The stages S2 to S4 perform the Winograd algorithm, which includes transformation to Winograd domain, element-wise multiplication, and transformation back to the spatial domain. Stage 1 performs decomposition, and stage 5 performs rearrangement. In the rearrangement stage, four intermediate output patterns generated from stage 4 are rearranged into a single ofmap.

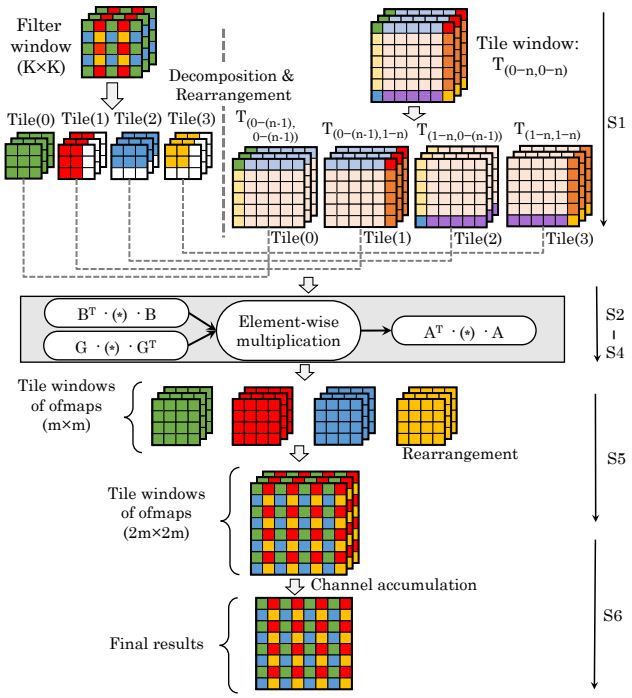

Fig. 34. Overall technique of Di et al. [42] (S1: kernel decomposition, S2: Winograd transform, S3: element-wise multiplication, S4: inverse Winograd transform, S5: rearrangement, S6: channel accumulation)

Their architecture has an array of PEs, each of which executes the TrCONV operation on a tile of ifmap. A PE executes 6 stages mentioned above and is implemented on the DSP or LUT-registers of an FPGA. They leverage both inter-PE and intra-PE level parallelism. As for inter-PE parallelism, every PE processes the data from one channel of an ifmap to one channel of an ofmap. Since each PU operates on four pairs of data (viz., sub-tile, Winograd-filter, etc.), they process them in parallel to achieve intra-PE parallelism. On FPGA, DSPs are required only for performing element-wise multiplication. The remaining steps are performed using programmable logic resources. To improve memory access efficiency, they partition the memory into multiple segments that can be concurrently accessed. On a ZCU102 FPGA platform, their technique achieves 639 GOPS. Also, compared to a CONV-accelerator, their design provides an 8.6× speedup.

4.4 Using Fermat Number Transform Based CONV

While the Winograd transform only works with a fixed filter dimension, the Fast Fourier Transform(FFT) incurs a high transform overhead. To overcome their limitations, Xu et al. [33] utilize the Fermat number transform ("FNT"). The computation flow of the Fermat number transform is nearly similar to that of the FFT method. Yet, it incurs a lower transform overhead, has lower complexity, and offers a bigger design space. The Fermat number transform uses a real transform kernel in the finite field and stores only the unsigned integers. This reduces the requirement of intermediate memory. Also, the multiplications are replaced with shifts and additions.

They propose two algorithms based on the FNT: a 2D overlap-and-save algorithm for performing fast CONV and a 1D overlap-and-save algorithm for performing the TrCONV. For performing the 2D overlap-and-save for the CONV, the first tiles of the filters and the ifmaps are transformed into a 'finite field' by performing 2D FNT. The transformed values are then multiplied element-wise for each channel. Finally, the inverse FNT is performed to obtain the output tiles. This is shown in Figure 35.

The 2D overlap-and-save method can be used to find the TrCONV also. The difference is before transforming the values into the finite field, we need to first do the zero-padding in the ifmap. These extra padded zeros cause redundancy and lead to poor off-chip memory bandwidth. As a result, the 2D overlap-and-save method is not a preferred choice for the TrCONV. The zero-padded TrCONV comes with the drawback of high complexity and high bandwidth requirement. For perfroming a fast TrCONV, the authors introduced the 1D overlap-and-save. As shown in Figure 36, $2\times(K-P-1)+(H-1)\times(S-1)$ rows are added in the ifmap. The image border comprises of $2\times(K-P-1)$ rows of zero, where the filter is of size $K\times K$, S is the stride, P are the padding size, and the TrCONV accepts the ifmap of size $H\times W$. Instead of processing the whole ifmap, only H non-zero rows of the ifmap is processed. The 1D FNT is performed only on the non-zero rows, followed by the element-wise multiplication between the fmaps and filters. Finally, the inverse FNT is performed to get the final ofmap. To further enhance the computing efficiency, batching is used. The algorithms reduced the computational complexity and provided a large improvement in performance.

Figure 37 shows the fast FNT kernel, which can perform both 1-D and 2-D FNT. The pipelined register relaxes the critical path. For performing 1D FNT, column FNT and transpose matrix are bypassed.

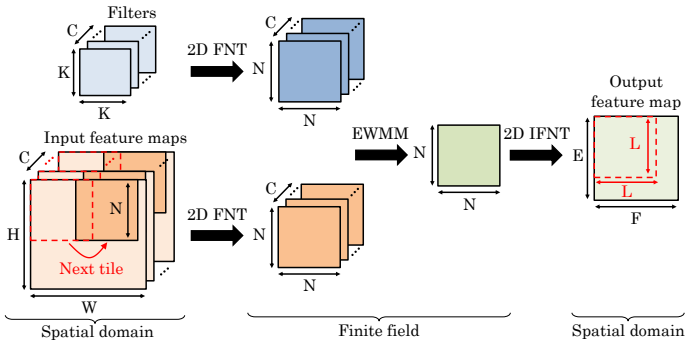

Fig. 35. Working of 2D overlap-and-save FNT [33]. 2D FNT transforms ifmap and filters to a finite feild. Then, elementwise multiplicaiton is performed, followed by the IFNT.

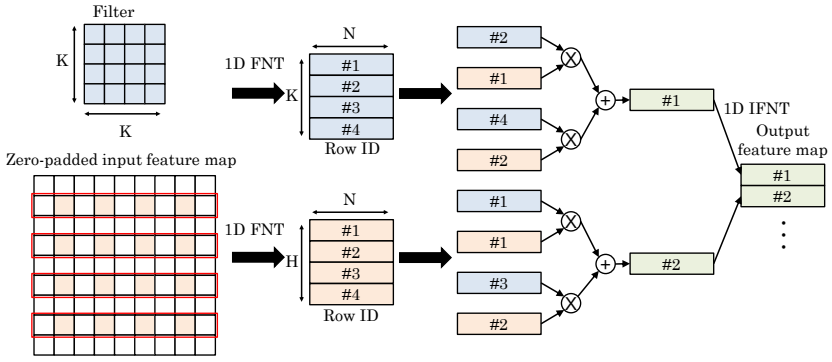

Fig. 36. TrCONV computation based on 1D overlap-and-save FNT. [33]. Instead of processing whole zero-padded fmap, 1D FNT is performed on the non-zero rows.

Their hardware accelerator is shown in Figure 38. The accelerator comprises five 2-D FNT modules. A 2D FNT module is used in between the input buffer and the input-output interface. Moreover, the 2D FNT of filters is enlarged by the factor of $(N/K)^2$ where the output is of size $N \times N$ and the filter is of size $K \times K$. To reduce the off-chip memory bandwidth, the on-chip processing of the filter is done. The four 2D FNT modules are used in a 4-way 2D FNT module to process the weights. To avoid any data stalls, the global buffer uses a double buffering technique. The computing element (CE) array performs element-wise multiplication and accumulation on the FNTs. Each CE row shares the same data and realizes the parallel execution by unrolling the output channel loop. The work results in a power dissipation of 805 mW. Their design achieves the energy efficiency of 12.5× and 275.3× in the GTX 1080 Ti GPU and the i7-7700 CPU, respectively. Also, as compared to the existing accelerators, a speed-up of atleast 1.7× is achieved.

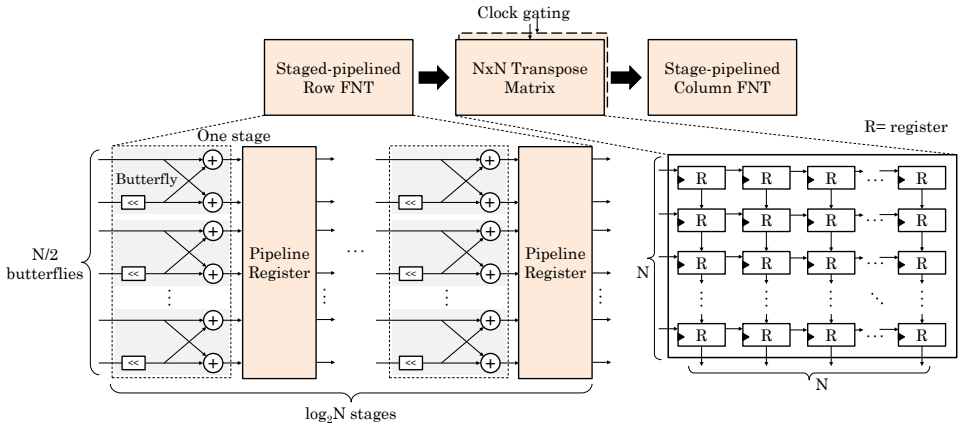

Fig. 37. Stage-pipelined fast FNT kernel for FNT size N [33] which can perform both 2D and 1D FNT. For 1D FNT results are obtained directly from the output row of FNT, thus bypassing both the columns and transpose FNT. For 2D FNT outputs of the row FNT are fed to transpose matrix then column FNT

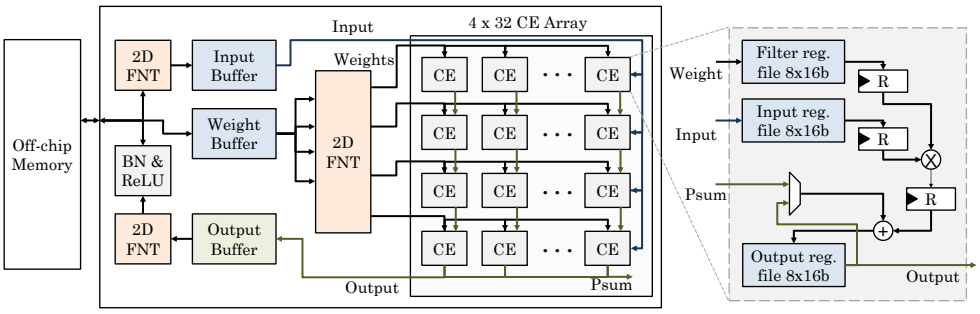

Fig. 38. Overall design of the proposed accelerator for GAN and CNN acceleration (CE = computing element). [33]

4.5 Using Sparsity-Related Techniques

Chen et al. [31] propose a sparsity-based accelerator for GANs. In their design, the PE comprises a FIFO group, an accumulator, a zero-removing unit, and a dual-channel register group. The authors propose an ineffectual data removing method, which eliminates both regular sparsity and irregular sparsity. For the irregular sparsity, zeros are filtered out using the zero-removing unit. After this, the effectual data is delivered to the FIFO with the least amount of data. This is shown in Figure 39. From there, the data is passed to the 'dual-channel accumulation' path. An accumulator performs the 'dual-channel accumulation', which is based on the time-division multiplexing. The ineffectual data removing method causes load imbalance as the PE processing the maximum number of non-zero operands becomes a bottleneck. To avoid this, they propose a load-balancing mechanism. In this mechanism, filters are first sorted in descending order based on the sparsity value. Then filter groups are created. In the case of the 16 filters, four filter groups are created. These groups are assigned to a PE cluster. Again, filters are sorted in a PE cluster. The filters with the highest and lowest sparsity are sent to PE in the first column, while the other filters are sent to PE in the second column. In the inter-tile level, the ifmap is assumed to be divided into 4 blocks. The first block and the last block are sent to the PEs in the first row. The rest are send to the second row PE. Their technique achieves a high energy efficiency of 3.72-TOPs/W at 50.1 mW.

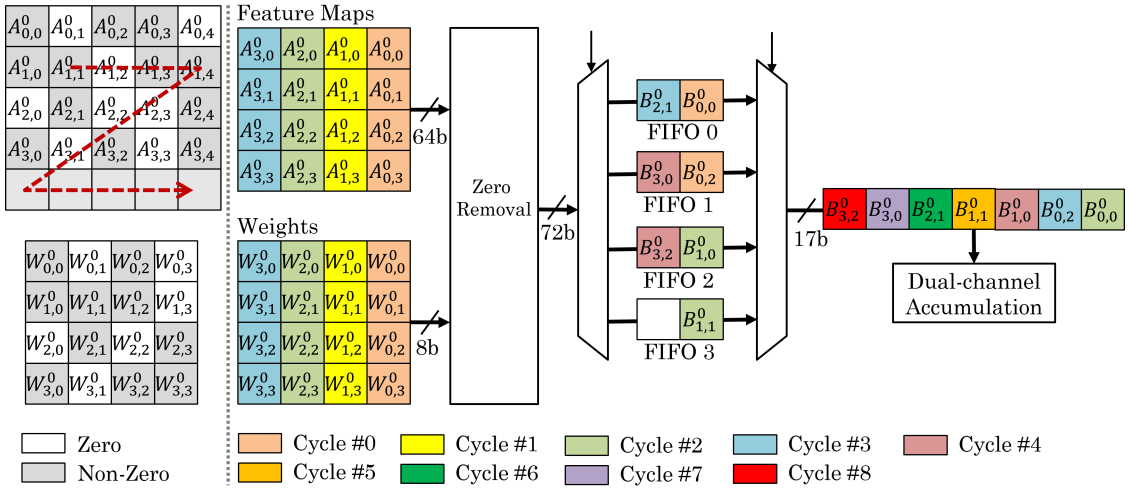

Fig. 39. Dataflow in the technique of Chen et al. [31]. At cycle 0 first four activations and weights are fetched and supplied to the "zero-removal unit" which filters out the ineffectual data. Effectual data is send to FIFO0 and FIFO1. Cycle 1 is simlar to the cycle 0, effectual data is send to FIFO2 and FIFO3. FIFO with the largest amount of data, transfers the data to the "dual-channel accumulation".

Yazdanbakhsh et al. [28] present an architecture for accelerating GANs on FPGA. Consider TrCONV between a 4×4 input and 5×5 filter (padding=2 and stride=1). In TrCONV, the input is enlarged from 4x4 to 11x11, as shown in Figure 40(a). Figure 40(b) shows the architecture of a conventional CONV accelerator for producing rows 2 to 5. Every PE multiplies a row of input with a row of the filter. The filter rows are vertically reused across the PEs, however, if an input row is zero, the corresponding filter row is not utilized for computing the output. Evidently, the presence of zeros leads to resource-idling, ineffectual computations, and challenges in exploiting data-reuse along the filter rows.

To mitigate these limitations, they propose changing the dataflow. They note that there are only two different computation-styles, which are used by odd and even rows. Based on this, they first reorder output rows to keep all the even rows together. This is shown in Figure 40(c). Then, they reorder the filter rows so that different rows feed only selected output rows. The final organization is shown in Figure 40(d). These optimizations improve data-reuse and increase resource utilization from 50% to 100%. While the original architecture needed 5 cycles to horizontally accumulate the partial

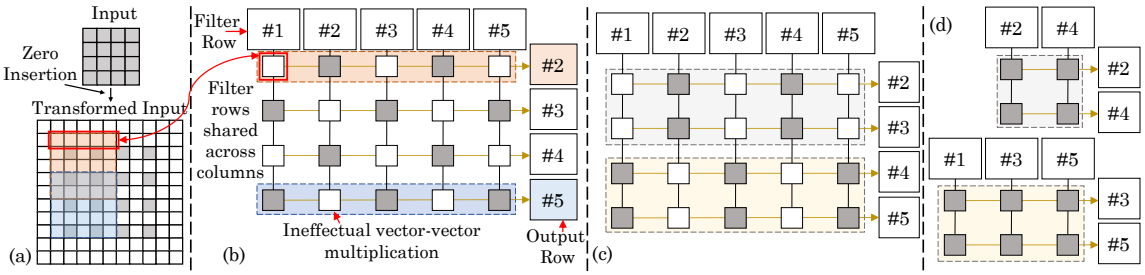

Fig. 40. (a) Zero-insertion step in TranConv operation for a 4x4 input and the transformed input. The white-colored squares represent zero values in the transformed input. (b) Using convolution dataflow for performing TranConv operations [28]. The flow of data after applying (c) output row reorganizaton and (d) filter row reorganization. The combination of these flow optimizations reduces the idle (white) operations and improves the resource utilization.

sums, their technique brings this latency to two and three cycles for even and odd (respectively) numbered rows.

They note that in the traditional CONV, the computation pattern of different sliding windows is the same, and hence, a SIMD-style execution works well. However, in TrCONV, the computation-pattern of different windows is not uniform, which leads to irregular memory accesses. Still, some patterns repeat, and they are organized in two subsets with the above optimizations. They use MIMD execution across the rows and SIMD execution across PEs in a row. Thus, they exploit the best of both SIMD and MIMD. In their architecture, PEs are organized in a 2D array. Each PE has one unit for performing memory accesses and another unit for performing operations such as summation, multiplication, and MAC. A global instruction buffer is shared by all the PEs, and a horizontal group of PEs shares a local instruction buffer. A bit in the instruction decides whether the accelerator works in SIMD or MIMD-SIMD mode in any cycle. In SIMD mode, all PEs execute the instruction from the global instruction buffer. In the MIMD-SIMD mode, all PEs in a row execute the instruction fetched by their local instruction buffer.

The data-fetching engines generate the addresses for the output, input, and weight buffers. The data-processing engine operates on the data stored in input/weight buffers and saves the output data in the output buffer. The instruction stream for data-processing engines mentions only the kind of computation to be done and has no field to specify the operand addresses. This allows reusing the instructions over many cycles since the instructions in different sliding windows have a similarity. It also reduces the storage overhead of instruction buffers. To reduce the requirement of on-chip memory, they design an ISA which has separate instructions for data-fetching and data-processing. They also propose an algorithm for finding the number of rows and columns of PEs that the accelerator should have for minimizing the execution latency. They implement their accelerator on an FPGA. It provides a 2.2× speedup over an optimized traditional CONV accelerator. Further, it provides 2.6× higher performance-per-watt compared to Titan X GPU. However, the performance of GPU is higher than their technique due to its higher frequency and compute resources.

Chang et al. [45] present a DeCONV accelerator for image super-resolution application. On using a $K_D \times K_D$ filter, a $K_C \times K_C$ input block produces a $S \times S$ output block. To deal with the issue of the overlapping sum in DeCONV operation, they find the number of input pixels required for generating a non-overlapping output block. Let N_o show the number of vertical or horizontal neighboring blocks that overlap within $\lfloor (K_D/2) \rfloor$. In up-scaling, $S > 1$ and the input elements mapped to the ofmap have a spacing of S. Also, an output block may overlap with neighboring output blocks in the range of $\lfloor (K_D/2) \rfloor$. Hence, $N_o = \lfloor (K_D/2) \rfloor \times 1/S$. The amount of overlap is determined by the fractional part of N_o. Based on this, they find the size of input block $K_C \times K_C$ that can generate non-overlapping output.

Their technique computes $S \times S$ pixels of the output block in parallel. For their DeCONV-to-CONV transformation approach, they find the coefficients of the filter in the CONV layer. From each DeCONV filter of size K_D^2, S^2 sparse CONV filters of size K_C^2 are obtained, e.g., when $S = 2$, four CONV filters are obtained. These filters have a different number of non-zero weights. To mitigate this load-imbalance, they rearrange these filters in the offline phase such that they have the nearly same number of non-zero weights. This is shown in Figure 41(a). Thus, their technique leverages sparsity in both weights and inputs. They also parallelize the ifmaps and ofmaps using loop optimization strategies.

As for the hardware architecture, they perform tiling and layer-fusion. A single CONV-layer engine executes all the layers for a single tile completely. Multiple such CONV-layer engines are used for achieving parallel execution between the tiles. The tile-size is found by comparing the execution latency of every CONV-layer engine with the data-transfer latency of pixel data from the "display driver" or CONV-layer engine. To enable the reuse of the data sent by the "display driver", they use a line buffer since the data is transferred at the granularity of a line. The line buffer is architected as a block-RAM, with two ports for allowing simultaneous read-write operations. To reduce the requirement of block-RAMs, they perform quantization, whereby the weights, input pixels, and partial sums are quantized from 32b floating-point to 13b fixed-point. Going below 13b leads to a sharp reduction in PSNR.

They further note that a DSP48 block in Xilinx FPGA can do up to a 25×18-bit multiplication. To ensure optimal utilization of DSP48 resources with 13×13-bit multiplication, they utilize the double MAC scheme [59]. In this scheme, two multiplications can be simultaneously performed if they share one operand, for example, A×B and A×C. They divide a 13×13-bit multiplication into three parts: 8×8-bit, 5×8-bit, and 13×5-bit and finally add the partial results. In

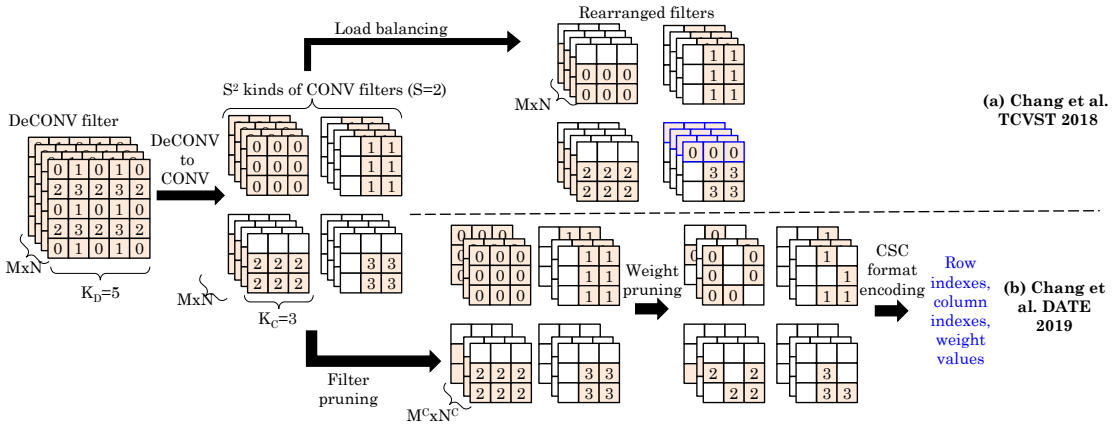

Fig. 41. Converting DeCONV to CONV and optimizing it using (a) load-balancing approach [45] [Chang et al. TCVST 2018] and (b) pruning and encoding approach [23] [Chang et al. DATE 2019]

CNNs, multiple operations are iteratively executed on the same ifmap, and hence, the "double MAC" scheme is effective. They evaluate their technique for the "Light FSRCNN" benchmark on a Kintex-7 XC7K410T FPGA. At super-resolution factors of 2 and 4, their technique achieves 145 GOPS/W and 500 GOPS/W, respectively. Their method provides better throughput and power efficiency as compared to conventional CONV/DeCONV accelerators. Also, for restoring high-resolution images to a higher quality, their technique needs less parameters and works with lower data-width than the traditional super-resolution techniques.

Chang et al. [23] highlight that although transforming a DeCONV layer to a CONV layer reduces the computational complexity over the conventional DeCONV operation, and it has a drawback of increasing the number of output feature maps, as it transforms each DeCONV filter into four CONV filters (each belonging to a different kind). To overcome this issue and further increase the computational efficiency of a DeCONV neural network, they propose a four-step method, which is shown in Figure 41(b). They first transform DeCONV operation into CONV operation. Then, they employ both filter pruning and weight pruning methods to decrease the number of feature maps and weights. Finally, they encode the sparse matrices in the "compressed sparse column" format.

Then, to efficiently process the generated sparse DeCONV neural networks, they modify the conventional dataflow to process different kinds of filters separately. This helps in balancing the load among PEs, as different kinds of filters have different average sparsity levels. Finally, they propose a hardware accelerator for FPGAs that receives tiled data and implements each type of filter separately for computing the output. They evaluate their technique on Virtex-7 485T FPGA. Their technique provides 2.6× speedup over a previous DeCONV accelerator [25].

4.6 Handling Overlapping Sum Problem

Mao et al. [43] propose a method to convert DeCONV to CONV operation. DeCONV can be converted to CONV after zero insertion in the original ifmaps. Multiplication of the weight elements with the zero-valued activation lowers the computational efficiency. If the weight elements of the DeCONV kernel, which are multiplied with the zero value activation, are removed, the kernel gets divided into subkernels. For a DeCONV kernel of size $K_d \times K_d$, the maximum and minimum sizes of the subkernels can be computed according to the DeCONV stride S_d. The maximum size of the subkernels, K_c^{Max} is $\lceil K_d/S_d \rceil$, while the minimum size, K_c^{Min}, is $\lfloor K_d/S_d \rfloor$. There are three cases based on the sizes of the subkernels.

Case 1: Here, K_d mod $S_d = 0$ and hence, the sizes of all the subkernels are equal. Figure 42(a), where $K_d = 4$ and $S_d = 2$ illustrates a simple DeCONV operation. Figure 42 (b) presents the condition when the DeCONV kernel is divided and simplied into four subkernels. For transforming kernel into subkernels, those weights are eliminated which were going to be multiplied with the zero-valued intput. Figure 42(c) presents the final fine-grained DeCONV method. The final output activation for each subkernel is achieved by directly convolving subkernels with the original ifmap. In this case, no load balancing is required.

Case 2: Here, K_d mod $S_d \neq 0$ and $K_c^{Max} \leq 3$. Here, the load of different subkernels is unbalanced. To achieve load-balancing, the subkernels of dimensions $K_c^{Max} \times K_c^{Max}$ and $K_c^{Min} \times K_c^{Max}$ are divided into two parts. However, the subkernels of dimensions $K_c^{Max} \times K_c^{Min}$ and $K_c^{Min} \times K_c^{Min}$ are not divided further. Assuming $K_d = 5$ and $S_d = 2$ and hence, $K_c^{Max} = 3$ and $K_c^{Min} = 2$. The subkernels of dimensions 3×3 and 2×3 are again divided into two subkernels, whereas those of dimensions 2×2 and 3×2 are not divided again. This converts the DeCONV into CONV with four subkernels, while also achieving better load-balance. $K_c^{Min} - 1$ columns in the results overlap.

Case 3: Here, K_d mod $S_d \neq 0$ and $K_c^{Max} > 3$. To achieve load-balancing, the subkernels are redivided until the size of every subkernel is at most 3.

Figure 43 presents the architecture. Here, the computing element comprises an adder tree and a CONV-DeCONV unit (CU) array, which is based on the Fast FIR algorithm. Row accumulator consists of an adder array and SRAM banks. Figure

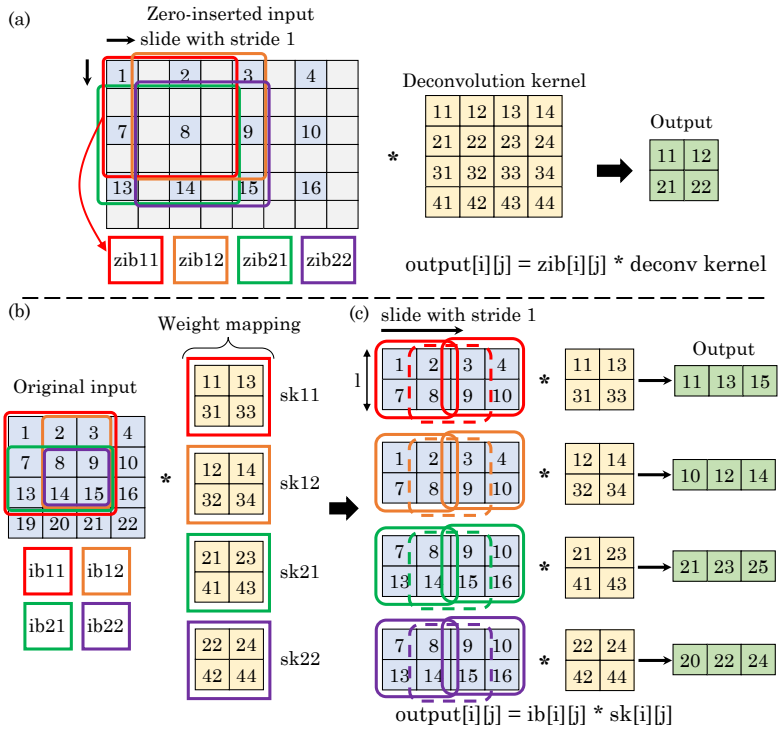

Fig. 42. Case 1: Example of $K_d \text{mod} S_d = 0$. The 4×4 DeCONV kernel and stride 2 is split into four 2×2 subkernels for performing CONVs. ib=input block, zib = zero-inserted input block, sk=subkernel. Performing DeCONV by (a) zero-insertion (b) removing zero-related operations (c) using fine-grain approach proposed by Mao et al. [43].

44 presents the detailed design of the computing element. The fast FIR algorithm reduces the multiplication at the cost of a few additions. A 2-parallel filter can perform 2×2 and 1×1 CONVs. By cascading small FIR filters, a large parallel FIR filter can be realized. For example, three 2-parallel filters are cascaded for realizing a 4-parallel filter that can perform 4x4 and 3x3 CONVs. This explains the reconfigurable capabilities of the CU.

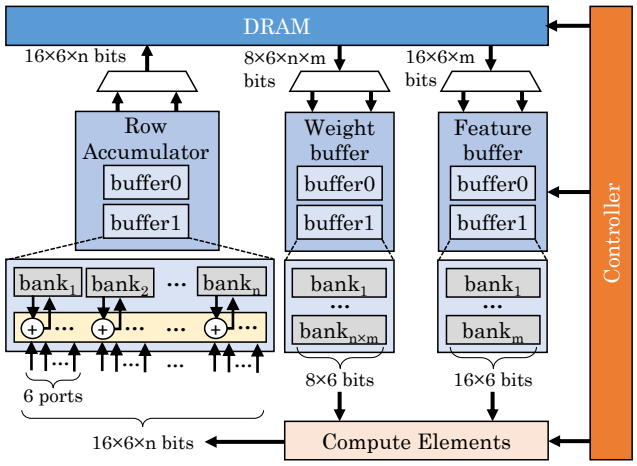

Fig. 43. The architecture for CONV-to-DeCONV system. It comprises of weight buffers, row accumulator, feature buffers, computing elements, controller and selection cells. Selection cells selects ping-pong buffers [43].

They also propose techniques to decrease the number of overlapped columns. They note that in the baseline computing approach, the outputs' part that contribute to the same overlapped locations are generated in different cycles. These results need to be temporarily stored in a buffer. To avoid this, they insert a zero at the beginning of the input row. This row is fed into the second computing unit, whereas the original row is fed into the first computing unit. With this approach, the outputs of the two computing units that add up to the same ofmap are produced in the same cycle. In Xilinx Virtex Ultrascale, the design leads to enhanced computational efficiency and reduced memory requirements with a power consumption of 3.71 W.

Liu et al. [36] propose a hardware accelerator for executing both the DeCONV and CONV operations for performing

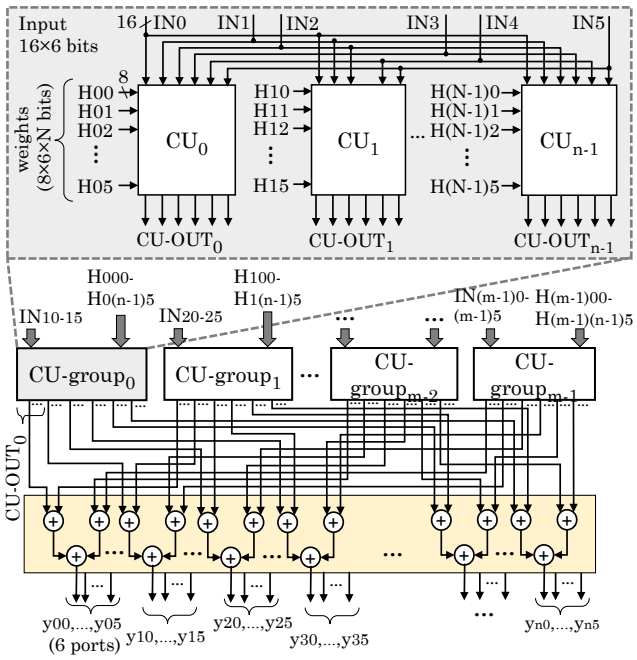

Fig. 44. Design of computing elements [43], which have m CU groups with n CUs each, and an adder tree array. An input row is shared between a CU group and a CU performs 1D CONV by rows. Adder tree of $m \times 6$ adder sums up the results from the CUs.

the 'semantic segmentation'. The state-of-the-art architecture for DeConv is implemented by adding zero paddings around or between the ifmap, which leads to poor computation efficiency on FPGA. In contrast to this method, the authors use an FPGA-friendly method in which a single input pixel is multiplied with the $k \times k$ kernel. This is performed for all the input pixels. Results are summed up wherever the output pixels overlap. In the end, border pixels are removed from the final ofmap based on the number of zero paddings.

The top-level architecture is shown in Figure 45(a). For the hardware design of DeCONV, the authors stored the processing data and coefficients of the network in the external memory. The ifmap is read into an input buffer. Corresponding coefficients are also cached into a coefficient buffer ($k \times k$ FIFO). From coefficient buffer, column-wise k coefficients are read into the coefficient register. A single input data and one column of the coefficient kernel from the coefficient buffer are then sent to the DeCONV kernel, which performs the DeCONV operation. Finally, outputs are sent to the output buffers. To process the next filter and to transfer the current filter result to the main memory concurrently, ping-pong FIFO is used for the output buffer.

Figure 45(b) shows the design of the DeCONV kernel. It can support DeCONV layers of different configurations. If s is the stride and k is the kernel's width, there are two scenarios when $k = s$ and $s < k$. When $s < k$, $k - s$ columns/rows overlap, requiring $k \times (k-s)$ register array size to deal with the overlap as shown in Figure 45(b). In each cycle, one column of the kernel coefficients is accessed requiring k coefficient registers for storage. These coeffficients are multiplied by single input data. This operation requires k multiplications. The results are stored in $k \times (k-s)$ array register. The content in the register array is shifted every cycle. To deal with the row overlap, the results of the first $(k-s)$ adders are also summed with the partial results then it is transferred to the output buffer. For the column overlap, the result of the multiplies and the last column of the registers are added before they are send in the buffer. When $k = s$, no overlap occurs.

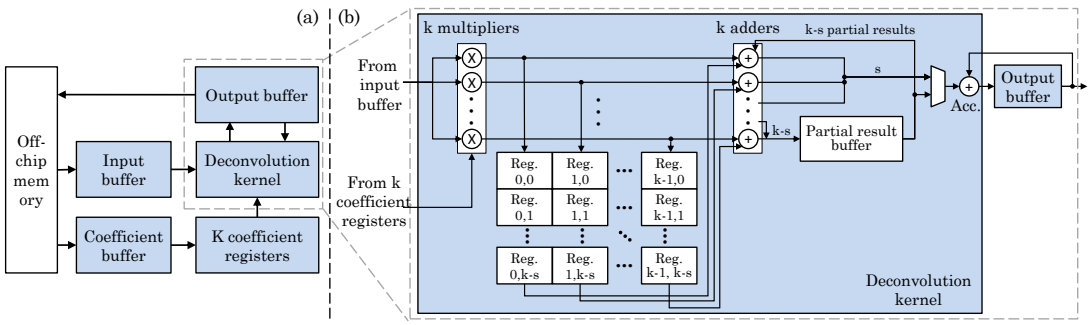

Fig. 45. (a) Overall architecture of Liu et al. [36]. All processing data and coefficients are stored in off-chip memory. (b) Design of DeCONV accelerator which supports different parameters of the DeCONV layer.

The authors propose an optimized hardware accelerator for CNN-based image segmentation, integrating both DeCONV

and CONV computational units. For increasing the throughput of the design, the input buffer is shared between DeCONV and CONV modules. Still, the input buffer cannot be wide enough to store the complete input map, so the ifmap is divided into several parts. This reduces communication time.

They completely parallelize the kernel matrix computation for both DeCONV and CONV layers. Moreover, data-level parallelism is achieved by accessing various inputs in a single cycle, and accordingly, computation kernels are also replicated. However, filter level parallelism requires replicating both filters and output buffers, limiting the memory and logic overheads. For further optimization, authors use multiple fixed-point data formats, such as 16b, 24b, or 32b, instead of floating-point.

The authors further propose a framework for hardware mapping that can generate a low-latency hardware architecture for any CNN. First of all, the framework finds optimal design parameters. Afterward, a software code is generated to configure the layers' parameters. Then, based on the above data, the final Verilog hardware code is generated. This framework improves the design quality and enhances the designer's productivity.

Finally, the authors propose an optimized U-Net architecture, which improves hardware efficiency as compared to the previous implementation. To decrease the total number of computations, the number of filters is reduced, and, to maintain a simple hardware design, padded (and not unpadded) CONVs are done for each layer. They evaluate their technique on semantic segmentation task on the cityscapes dataset using Zynq board with a power consumption of only 9.6 W. Their optimizations reduce the operations from 30 giga-operations to 5.92 giga-operations but reduce the pixel-level accuracy from 68% to 60.8%.

4.7 Handling Irregular Memory Accesses

Xia et al. [29] propose an FPGA-based inference accelerator for complex CONV operations such as DeCONV, dilated CONV, CONV with upsampling, etc. Their accelerator has multiple pipelines with CONV units and post-processing units. Their post-processing units implement operations such as activation, pooling, etc. The input, weight, and output are stored in respective buffers. These buffers can support different dataflows. Irregular access is supported in the buffer and the off-chip memory, and this is helpful for assembling ofmap in DeCONV. Since the data-access granularity is already large, irregular accesses do not significantly degrade the throughput. Different pipelines can work separately on different CONV operations using their buffers, or they can work on a single CONV such that its data is spread across the buffers.

The core computing pattern of the CONV engine is a 'vector dot-product' unit. It is realized using either cascaded DSP blocks or by using a LUT-based multipliers followed by an adder tree. To achieve high data-reuse, both broadcast and systolic array schemes are used. The systolic array allows the reuse of both inputs and weights. For reducing the propagation delay, the dot-product units are split into multiple systolic arrays, and the input and weights are broadcast to multiple arrays. They use a compiler for performing optimizations such as quantization, reconstruction of DeCONV, etc. Depending on the network characteristics and FPGA resources, a suitable parallelism level is chosen in the channel and ofmap dimensions.

To avoid ineffectual computations in the execution of DeCONV, they reconstruct DeCONV in a compiler and implement it on the CONV architecture itself. They partition every weight filter into 4 smaller filters. Then, CONV is performed between the original ifmap (without padding) and these 4 filters. The pixels of the 4 ofmaps thus produced are not contiguous, and hence, they perform concatenation operation for combining them, as shown in Figure 46(a). Figure 46(b) focuses on a specific sequence of operations. As shown in Figure 46(c), concatenation is implemented without additional overhead through storing the ofmap and loading the ifmap. For this, the ofmap is stored in a non-contiguous fashion at memory location such that CONV1 and CONV2 together finally take contiguous memory region (Figure 46(c)). Further, when CONV3 is processed, the ifmap is loaded in a non-contiguous manner from this contiguous memory (Figure 46(c)). Their technique on FPGA achieves higher throughput and consumes lower energy than a GPU-based technique. For example, for U-Net, an FP32 implementation on NVIDIA P4 GPU reaches 99 FPS and 1.3 FPS per watt, whereas an INT16 implementation of their technique on Xilinx KU115 reaches 105 FPS and 2.1 FPS per watt.

5 Open Research Challenges and Future Roadmap

Accelerators for Advanced GANs: Over the past few years, the field of GANs has progressed a lot, and along the way, many variants of GANs have emerged. Although most of the core operations are the same across the variants, there are some core differences that lead to underutilization of the available computational resources in hardware accelerators, thereby leading to higher latency, lower throughput, and less energy efficiency. For example, CycleGAN [60] is one of the variants mainly popular for image-to-image translation applications. It includes two generators and two discriminators, and apart from the adversarial losses, the objective includes cycle-consistency loss term. Therefore, the same accelerators that are designed for conventional GANs, such as DCGAN, may not offer optimal performance and resource efficiency for CycleGANs. Some other examples include StackGAN [61], which uses two stages of GANs, and GANs used for high-quality text-to-image translation applications [62], requiring LSTM cells. Thus, to support the rapid progress in GANs, there is a need to shift the focus towards building tools and frameworks that make use of the hardware architectures proposed so far to construct specialized accelerators for advanced GANs based on an abstract description. Moreover, reconfigurable hardware can be used with these frameworks to significantly reduce the time to market. FlexiGAN [28] is one such solution that receives a GAN's high-level description and target specifications and generates synthesizable Verilog code for FPGA.

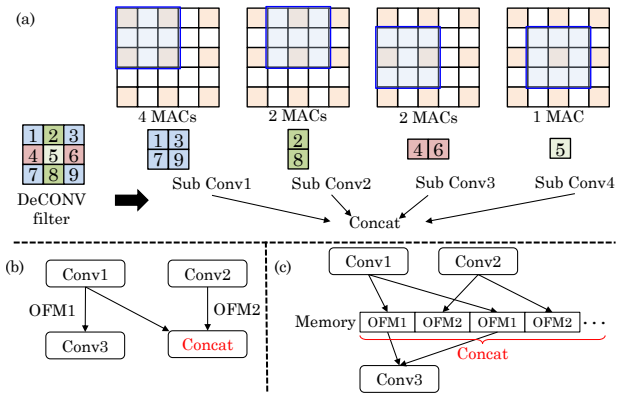

Fig. 46. (a) Achieving DeCONV by dividing it into four CONV operations and then, concatenating the results (b) The sequence of operations (c) storing ifmap1 and ifmap2 in non-contiguous locations to achieve concatenation operation without additional overhead [29]

Reconfigurability: To accommodate changing GAN algorithms and evolution, it is imperative to devise specialized GAN hardware that support adaptability / reconfigurability without any changes to the underlying compute fabric. However, the level of reconfigurability affects the efficiency. Therefore, it is important to study the commonalities between given GAN architectures and offer only the required level of reconfigurability to preserve the efficiency benefits.

Data and Memory Management: An off-chip memory (i.e., DRAM) access costs orders of magnitude higher energy and latency compared to a MAC operation. Most of the dataflow schemes are designed to increase local data reuse and avoiding redundant DRAM accesses. SmartShuttle [63] proposed an adaptive data partitioning and scheduling scheme for reducing DRAM accesses in CNN accelerators. Similar studies are required for GANs, specifically for optimizing the off-chip memory accesses during the training process. These studies can further be combined with design space exploration for optimal on-chip memory size and organization, as the availability of sufficient on-chip memory helps increase data locality and thus, reduce off-chip memory accesses. CPUs are invariably present in all systems and are especially prominent in accelerating AI algorithms in mobile-systems [64]. Also, they incorporate advanced data and memory-management techniques. Hence, optimization of GANs on CPUs is vital to deploy them on a wide range of usage scenarios.

Reliability of GAN Hardware: GAN hardware architectures (e.g., on-chip memories and PE arrays) are exposed to various reliability threats like soft errors [65], aging, thermal issues, and manufacturing defects. Moreover, off-chip memories can also exhibit different types of faults due to manufacturing defects or longer refresh rate intervals for increasing DRAM energy efficiency. Towards this, systematic fault-resilience studies need to be conducted to study the impact of various types of faults on the application-level accuracy of GANs. These studies can help in designing low-cost fault-mitigation techniques. Moreover, the effectiveness of conventional fault-aware training approaches that are designed for CNNs and recurrent neural networks (RNNs) [66] must be evaluated for GANs, and customized approaches should be designed considering the difficulty of GAN training.

Security of GAN Hardware: AI models are a crucial IP for any company and hence, there is an increasing concern for ensuring their security against a range of attacks such as Trojan attack, side-channel attack and fault-injection attack [67]. Going forward, security needs to be taken as a first principle in the design of GAN accelerators.

Processing-in-memory: To avoid the large number of expensive memory accesses required during training and inference of a DNN, physical attributes of resistive memory devices, in which memory cells are arranged in the form of crossbar arrays, can be used to efficiently perform computations in place in a "non-von Neumann" manner [49]. Most of the works that make use of these attributes assume ideal device behavior and do not discuss the associated challenges in detail. One of the crucial issues is the inability to alter the resistance/conductance of the memory cells at a finer granularity in a reliable manner and write precise values. This leads to imprecise computations, which eventually affects the application-level accuracy. For example, a customized mixed-precision training technique is proposed by Gallo et al. [68] to overcome the challenges. Their evaluation shows that, even with a customized training approach and for less complex problems such as MNIST handwritten digit classification, the achieved application-level accuracy is always less than the same DNN architecture trained and evaluated using digital circuitry. Other key problems associated with resistive crossbar-based in-memory computing include voltage drop in wires and device variations [69]. As all these factors cannot be ignored from the practicality perspective, the architectures that make use of resistive crossbars for computations should be evaluated considering state-of-the-art non-ideal device models to clearly show the impact on accuracy. Moreover, there is a need to focus on developing techniques that can overcome the non-idealities of these devices without significantly affecting their benefits.

Optimization and approximation for GANs to improve Energy-Efficiency: Similar to the case of DNN optimization [70, 71], different types of software and hardware approximations can also be employed in GAN systems, covering both the hardware architecture and the GAN model compression. Some example ideas could be hardware-aware network compression, conjoint network compression and quantization, voltage scaling to near-threshold voltage level [72], and exploiting functional approximations in processing elements for higher area and energy efficiency. Combinations of

these techniques can also be investigated to achieve higher benefits. Recent works, such as GAN Compression [73] and QGAN [74] have shown that conventional techniques that are designed for CNNs and RNNs cannot be directly employed due to the difficulty of GAN training and underrepresentation of original values. Therefore, tailored approaches are required that can offer a better accuracy-efficiency tradeoff. Alongside on-device analytics, distributed machine learning is also gaining a lot of attention due to its wide scope in IoT-related use cases. Towards this, dedicated hardware/software co-design frameworks that jointly optimize hardware, DNNs, and communication requirements are required for unlocking highly energy-efficient systems.

Specialized Hardware for Private Inference: The increase in cloud-based machine learning applications has given rise to privacy concerns. Over the past few years, various techniques have been proposed to ensure private inference, i.e., data processing without disclosing inputs. These techniques incorporate additional concepts from cryptographic techniques to ensure privacy [75]. However, the additional functionality results in significant overheads. Optimization techniques have been proposed to design neural architectures that incur lower overheads while ensuring high accuracy. These techniques can be complemented with specialized hardware to enable the use of high accuracy models in real-world systems while ensuring data privacy.

AI Frameworks for Designing Hardware: The success of machine learning and artificial intelligence in computer vision and language processing has led to their adoption in many other fields. Recently, researchers from Google have shown that AI can be used to design hardware that offers comparable or better results than the designs made by experts, and it requires only a few hours to generate such designs compared to weeks/months of time required by human experts [76]. As this results in a significant reduction in the design cost, the use of AI for designing efficient hardware is going to be one of the prominent areas of research in the coming years.

6 Conclusion

In this work, we presented a comprehensive survey of hardware accelerators for GANs. We organized the works based on key metrics to bring out their differences as well as similarities. We have also presented some future research problems in the field of GAN architectures. We believe that this survey will help both experts and novice in the area of artificial intelligence, computer architecture and chip-design.

References

[1] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial networks," *arXiv preprint arXiv:1406.2661*, 2014.

[2] A. Sadeghian, V. Kosaraju, A. Sadeghian, N. Hirose, H. Rezatofighi, and S. Savarese, "SoPhie: An Attentive GAN for Predicting Paths Compliant to Social and Physical Constraints," in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019, pp. 1349–1358.

[3] C. Vondrick, H. Pirsiavash, and A. Torralba, "Generating videos with scene dynamics," in *Advances in Neural Information Processing Systems*, vol. 29, 2016, pp. 613–621.

[4] C. Ledig, L. Theis, F. Huszar, J. Caballero, A. Cunningham, A. Acosta, A. Aitken, A. Tejani, J. Totz, Z. Wang, and W. Shi, "Photo-realistic single image super-resolution using a generative adversarial network," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 105–114.

[5] T. Xu, P. Zhang, Q. Huang, H. Zhang, Z. Gan, X. Huang, and X. He, "AttnGAN: Fine-Grained Text to Image Generation with Attentional Generative Adversarial Networks," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018, pp. 1316–1324.

[6] J. Johnson, A. Alahi, and L. Fei-Fei, "Perceptual losses for real-time style transfer and super-resolution," in *European Conference on Computer Vision (ECCV)*, 2016, pp. 694–711.

[7] S. Mittal and S. Vaishay, "A Survey of Techniques for Optimizing Deep Learning on GPUs," *Journal of Systems Architecture*, 2019.

[8] Z. Fan, Z. Li, B. Li, Y. Chen, and H. H. Li, "RED: A ReRAM-based deconvolution accelerator," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2019, pp. 1763–1768.

[9] O. Krestinskaya, B. Choubey, and A. James, "Memristive GAN in Analog," *Scientific Reports*, vol. 10, no. 1, pp. 1–14, 2020.

[10] F. Shi, Z. Xu, T. Yuan, and S.-C. Zhu, "HUGE2: a Highly Untangled Generative-model Engine for Edge-computing," *arXiv preprint arXiv:1907.11210*, 2019.

[11] https://datascience.stackexchange.com/questions/22387/what-is-the-difference-between-dilated-convolution-and-deconvolution.

[12] https://github.com/vdumoulin/conv_arithmetic.

[13] https://towardsdatascience.com/types-of-convolutions-in-deep-learning-717013397f4d.

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

[15] T. Kaneko, M. Ikebe, S. Takamaeda-Yamazaki, M. Motomura, and T. Asai, "Hardware-Oriented Algorithm and Architecture for Generative Adversarial Networks," *Journal of Signal Processing*, vol. 23, no. 4, pp. 151–154, 2019.

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

[17] F. Liu and C. Liu, "A Memristor based Unsupervised Neuromorphic System Towards Fast and Energy-Efficient GAN," *arXiv preprint arXiv:1806.01775*, 2018.

[18] A. S. Rakin, S. Angizi, Z. He, and D. Fan, "PIM-TGAN: A processing-in-memory accelerator for ternary generative adversarial networks," in *2018 IEEE 36th International Conference on Computer Design (ICCD)*. IEEE, 2018, pp. 266–273.

[19] H. Mao, J. Shu, M. Song, and T. Li, "LrGAN: A Compact and Energy Efficient PIM-based Architecture for GAN Training," *IEEE Transactions on Computers*, 2020.

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

[21] H. Xu, Y. Wang, Y. Wang, J. Li, B. Liu, and Y. Han, "ACG-Engine: An Inference Accelerator for Content Generative Neural Networks," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2019, pp. 1–7.

[22] D. Wang, J. Shen, M. Wen, and C. Zhang, "Towards a Uniform Architecture for the Efficient Implementation of 2D and 3D Deconvolutional Neural Networks on FPGAs," in *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2019, pp. 1–5.

[23] J.-W. Chang, K.-W. Kang, and S.-J. Kang, "SDCNN: An efficient sparse deconvolutional neural network accelerator on FPGA," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2019, pp. 968–971.

[24] D. Xu, K. Tu, Y. Wang, C. Liu, B. He, and H. Li, "FCN-engine: Accelerating deconvolutional layers in classic CNN processors," in *Proceedings of the International Conference on Computer-Aided Design*. ACM, 2018, p. 22.

[25] J.-W. Chang and S.-J. Kang, "Optimizing FPGA-based convolutional neural networks accelerator for image super-resolution," in *Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2018, pp. 343–348.

[26] S. Liu, C. Zeng, H. Fan, H.-C. Ng, J. Meng, Z. Que, X. Niu, and W. Luk, "Memory-Efficient Architecture for Accelerating Generative Networks on FPGA," in *2018 International Conference on Field-Programmable Technology (FPT)*. IEEE, 2018, pp. 30–37.

[27] F. Chen, L. Song, H. H. Li, and Y. Chen, "ZARA: a novel zero-free dataflow accelerator for generative adversarial networks in 3D ReRAM," in *Design Automation Conference*, 2019, p. 133.

[28] A. Yazdanbakhsh, M. Brzozowski, B. Khaleghi, S. Ghodrati, K. Samadi, N. S. Kim, and H. Esmaeilzadeh, "FlexiGAN: An end-to-end solution for FPGA acceleration of generative adversarial networks," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2018, pp. 65–72.

[29] L. Xia, L. Diao, Z. Jiang, H. Liang, K. Chen, L. Ding, S. Dou, Z. Su, M. Sun, J. Zhang *et al.*, "PAI-FCNN: FPGA Based Inference System for Complex CNN Models," in *30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, vol. 2160, 2019, pp. 107–114.

[30] D. Im, D. Han, S. Choi, S. Kang, and H.-J. Yoo, "DT-CNN: An Energy-Efficient Dilated and Transposed Convolutional Neural Network Processor for Region of Interest Based Image Segmentation," *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2020.

[31] Q. Chen, Y. Huang, R. Sun, W. Song, Z. Lu, Y. Fu, and L. Li, "An Efficient Accelerator for Multiple Convolutions From the Sparsity Perspective," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 6, pp. 1540–1544, 2020.

[32] J.-W. Chang, S. Ahn, K.-W. Kang, and S.-J. Kang, "Towards Design Methodology of Efficient Fast Algorithms for Accelerating Generative Adversarial Networks on FPGAs," in *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2020, pp. 283–288.

[33] W. Xu, Z. Zhang, X. You, and C. Zhang, "Reconfigurable and Low-Complexity Accelerator for Convolutional and Generative Networks over Finite Fields," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020.

[34] A. Roohi, S. Sheikhfaal, S. Angizi, D. Fan, and R. F. DeMara, "ApGAN: Approximate GAN for Robust Low Energy Learning from Imprecise Components," *IEEE Transactions on Computers*, 2019.

[35] J. Yan, S. Yin, F. Tu, L. Liu, and S. Wei, "GNA: Reconfigurable and efficient architecture for generative network acceleration," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2519–2529, 2018.

[36] S. Liu, H. Fan, X. Niu, H.-c. Ng, Y. Chu, and W. Luk, "Optimizing CNN-based segmentation with deeply customized convolutional and deconvolutional architectures on FPGA," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 11, no. 3, p. 19, 2018.

[37] Y. Yu, T. Zhao, M. Wang, K. Wang, and L. He, "Uni-OPU: An FPGA-Based Uniform Accelerator for Convolutional and Transposed Convolutional Networks," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2020.

[38] S. Perri, C. Sestito, F. Spagnolo, and P. Corsonello, "Efficient Deconvolution Architecture for Heterogeneous Systems-on-Chip," *Journal of Imaging*, vol. 6, no. 9, p. 85, 2020.

[39] S.-F. Hsiao, K.-C. Chen, C.-C. Lin, H.-J. Chang, and B.-C. Tsai, "Design of a Sparsity-Aware Reconfigurable Deep Learning Accelerator Supporting Various Types of Operations," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 10, no. 3, pp. 376–387, 2020.

[40] M. A. Hanif, M. Z. Akbar, R. Ahmed, S. Rehman, A. Jantsch, and M. Shafique, "MemGANs: Memory Management for Energy-Efficient Acceleration of Complex Computations in Hardware Architectures for Generative Adversarial Networks," in *International Symposium on Low Power Electronics and Design (ISLPED)*, 2019, pp. 1–6.

[41] A. Yazdanbakhsh, K. Samadi, N. S. Kim, and H. Esmaeilzadeh, "GANAX: A unified MIMD-SIMD acceleration for generative adversarial networks," in *International Symposium on Computer Architecture (ISCA)*, 2018, pp. 650–661.

[42] X. Di, H. Yang, Z. Huang, N. Mao, Y. Jia, and Y. Zheng, "Exploring Resource-Efficient Acceleration Algorithm for Transposed Convolution of GANs on FPGA," in *International Conference on Field-Programmable Technology (ICFPT)*, 2019, pp. 19–27.

[43] W. Mao, J. Lin, and Z. Wang, "F-DNA: Fast Convolution Architecture for Deconvolutional Network Acceleration," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 8, pp. 1867–1880, 2020.

[44] D. Xu, C. Liu, Y. Wang, K. Tu, B. He, and L. Zhang, "Accelerating generative neural networks on unmodified deep learning processors-software approach," *IEEE Transactions on Computers*, vol. 69, no. 8, pp. 1172–1184, 2020.

[45] J.-W. Chang, K.-W. Kang, and S.-J. Kang, "An energy-efficient FPGA-based deconvolutional neural networks accelerator for single image super-resolution," *IEEE Transactions on Circuits and Systems for Video Technology*, 2018.

[46] L. Bai, Y. Lyu, and X. Huang, "A Unified Hardware Architecture for Convolutions and Deconvolutions in CNN," *arXiv preprint arXiv:2006.00053*, 2020.

[47] W. Mao, J. Wang, J. Lin, and Z. Wang, "Methodology for Efficient Reconfigurable Architecture of Generative Neural Network," in *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2019, pp. 1–5.

[48] X. Zhang, S. Das, O. Neopane, and K. Kreutz-Delgado, "A design methodology for efficient implementation of deconvolutional neural networks on an FPGA," *arXiv preprint arXiv:1705.02583*, 2017.

[49] S. Mittal, "A Survey of ReRAM-based Architectures for Processing-in-memory and Neural Networks," *Machine learning and knowledge extraction*, vol. 1, p. 5, 2018.

[50] S. Umesh and S. Mittal, "A survey of spintronic architectures for processing-in-memory and neural networks," *Journal of Systems Architecture*, vol. 97, pp. 349–372, August 2019.

[51] A. Paszke, A. Chaurasia, S. Kim, and E. Culurciello, "ENet: A deep neural network architecture for real-time semantic segmentation," *arXiv preprint arXiv:1606.02147*, 2016.

[52] S. Mittal, "A Survey of FPGA-based Accelerators for Convolutional Neural Networks," *Neural computing and applications*, vol. 32, no. 4, pp. 1109–1139, 2020.

[53] S. Mittal and Vibhu, "A Survey of Accelerator Architectures for 3D Convolution Neural Networks," *Journal of Systems Architecture*, vol. 115, p. 102041, May 2021.

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

[55] M. Zieba, P. Semberecki, T. El-Gaaly, and T. Trzcinski, "BinGAN: Learning compact binary descriptors with a regularized GAN," in *Advances in neural information processing systems*, 2018, pp. 3608–3618.

[56] S. Mittal, "A Survey Of Techniques for Approximate Computing," *ACM Computing Surveys*, vol. 48, no. 4, pp. 62:1–62:33, 2016.

[57] D. Moolchandani, A. Kumar, and S. R. Sarangi, "Accelerating CNN Inference on ASICs: A survey," *Journal of Systems Architecture*, p. 101887, 2020.

[58] S. Pattanayak, S. Nag, and S. Mittal, "CURATING: A Multi-Objective based Pruning Technique for CNNs," *Journal of Systems Architecture*, vol. 116, p. 102031, January 2021.

[59] D. Nguyen, D. Kim, and J. Lee, "Double MAC: Doubling the performance of convolutional neural networks on modern FPGAs," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 2017, pp. 890–893.

[60] J.-Y. Zhu, T. Park, P. Isola, and A. A. Efros, "Unpaired image-to-image translation using cycle-consistent adversarial networks," in *Proceedings*

of the IEEE international conference on computer vision, 2017, pp. 2223–2232.
[61] H. Zhang, T. Xu, H. Li, S. Zhang, X. Wang, X. Huang, and D. N. Metaxas, "Stackgan: Text to photo-realistic image synthesis with stacked generative adversarial networks," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 5907–5915.
[62] S. K. Gorti and J. Ma, "Text-to-image-to-text translation using cycle consistent adversarial networks," *arXiv preprint arXiv:1808.04538*, 2018.
[63] J. Li, G. Yan, W. Lu, S. Jiang, S. Gong, J. Wu, and X. Li, "Smartshuttle: Optimizing off-chip memory accesses for deep learning accelerators," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018, pp. 343–348.
[64] S. Mittal, P. Rajput, and S. Subramoney, "A Survey of Deep Learning on CPUs: Opportunities and Co-optimizations," *IEEE Transactions on Neural Networks and Learning Systems*, 2021.
[65] S. Mittal, "A Survey on Modeling and Improving Reliability of DNN Algorithms and Accelerators," *Journal of Systems Architecture*, vol. 104, p. 101689, March 2020.
[66] S. Mittal and S. Umesh, "A Survey on Hardware Accelerators and Optimization Techniques for RNNs," *Journal of Systems Architecture*, vol. 112, p. 101839, January 2021.
[67] S. Mittal, H. Gupta, and S. Srivastava, "A Survey on Hardware Security of DNN Models and Accelerators," *Journal of Systems Architecture*, 2021.
[68] M. L. Gallo, I. Boybat, B. Rajendran, A. Sebastian, E. Eleftheriou *et al.*, "Mixed-precision training of deep neural networks using computational memory," *arXiv preprint arXiv:1712.01192*, 2017.
[69] S. Yu, P.-Y. Chen, Y. Cao, L. Xia, Y. Wang, and H. Wu, "Scaling-up resistive synaptic arrays for neuro-inspired architecture: Challenges and prospect," in *2015 IEEE International Electron Devices Meeting (IEDM)*. IEEE, 2015, pp. 17–3.
[70] A. Marchisio, M. A. Hanif, F. Khalid, G. Plastiras, C. Kyrkou, T. Theocharides, and M. Shafique, "Deep learning for edge computing: Current trends, cross-layer optimizations, and open research challenges," in *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 2019, pp. 553–559.
[71] M. A. Hanif, A. Marchisio, T. Arif, R. Hafiz, S. Rehman, and M. Shafique, "X-DNNs: Systematic cross-layer approximations for energy-efficient deep neural networks," *Journal of Low Power Electronics*, vol. 14, no. 4, pp. 520–534, 2018.
[72] S. Mittal, "A survey of architectural techniques for near-threshold computing," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 12, no. 4, pp. 46:1–46:26, 2015.
[73] M. Li, J. Lin, Y. Ding, Z. Liu, J.-Y. Zhu, and S. Han, "GAN compression: Efficient architectures for interactive conditional GANs," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 5284–5294.
[74] P. Wang, D. Wang, Y. Ji, X. Xie, H. Song, X. Liu, Y. Lyu, and Y. Xie, "QGAN: Quantized Generative Adversarial Networks," *arXiv preprint arXiv:1901.08263*, 2019.
[75] Z. Ghodsi, A. Veldanda, B. Reagen, and S. Garg, "Cryptonas: Private inference on a relu budget," *arXiv preprint arXiv:2006.08733*, 2020.
[76] A. Mirhoseini, A. Goldie, M. Yazgan, J. W. Jiang, E. Songhori, S. Wang, Y.-J. Lee, E. Johnson, O. Pathak, A. Nazi *et al.*, "A graph placement methodology for fast chip design," *Nature*, vol. 594, no. 7862, pp. 207–212, 2021.