Appears in the Proceedings of the 26th IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2018

FlexiGAN: An End-to-End Solution for FPGA Acceleration of Generative Adversarial Networks

Amir Yazdanbakhsh Michael Brzozowski Behnam Khaleghi[†] Soroush Ghodrati
Kambiz Samadi[‡] Nam Sung Kim[§] Hadi Esmaeilzadeh[†]
Alternative **C**omputing **T**echnologies (**ACT**) Lab
Georgia Institute of Technology [†]University of California, San Diego
[‡]Qualcomm Technologies, Inc. [§]University of Illinois at Urbana-Champaign
{a.yazdanbakhsh, mbrzozowski}@gatech.edu bkhalegh@eng.ucsd.edu s.ghodrati@gatech.edu
ksamadi@qti.qualcomm.com nskim@illinois.edu hadi@eng.ucsd.edu

***Abstract*— Generative Adversarial Networks (GANs) are a frontier in deep learning. GANs consist of two models: generative and discriminative. While the discriminative model uses the conventional convolution, the generative model depends on a fundamentally different operator, called transposed convolution. This operator initially inserts a large number of zeros in its input and then slides a window over this expanded input. This zero-insertion step leads to a large number of ineffectual operations and creates distinct patterns of computation across the sliding windows. The ineffectual operations along with the variation in computation patterns lead to significant resource underutilization when using conventional convolution hardware. To alleviate these sources of inefficiency, this paper devises FlexiGAN, an end-to-end solution, that generates an optimized synthesizable FPGA accelerator from a high-level GAN specification. FlexiGAN is coupled with a novel template architecture that aims to harness the benefits of both MIMD and SIMD execution models to avoid ineffectual operations. To this end, the proposed architecture separates data retrieval and data processing units at the finest granularity of each compute engine. Leveraging this separation enables the architecture to use a succinct set of operations to cope with the irregularities of transposed convolution. At the same time, it significantly reduces the on-chip memory usage, which is generally limited in FPGAs. We evaluate our end-to-end solution by generating FPGA accelerators for a variety of GANs. These generated accelerators provide 2.2× higher performance than an optimized conventional convolution design. In addition, FlexiGAN, on average, yields 2.6× (up to 3.7×) improvements in Performance-per-Watt over a Titan X GPU.**

I. Introduction

The training of deep neural networks requires massive labeled datasets. Labeling is laborious and can be prohibitively expensive due to the required human effort. To address this challenge, a new class of networks called Generative Adversarial Networks (GANs) have been developed. GANs [1] automatically generate bigger and richer datasets from a small labeled set and have been proven to be effective in various domains, including but not limited to robotics [2], autonomous driving [3], media synthesis [4], and medicine [5]. GANs comprise a generative model, which generates synthetic data, and a discriminative model, a conventional neural network that distinguishes between synthetic and genuine data [1]. These two models contend, strengthening each other. While GANs have recently become a prominent algorithm in deep learning, accelerator design for them is unexplored. This paper sets out to navigate this uncharted territory as GANs use a new type of mathematical operator in their generative model, called *transposed convolution* [6]–[11]. This operator cannot be executed resource-efficiently in the abundance of the accelerators for conventional deep neural networks [12]–[20]. This inefficiency stems from the fact that transposed convolution first inserts zeros in its input and then convolves a kernel over this expanded input. The inserted zeros cause resource underutilization in a conventional convolution engine. We proceed to identify the sources of these inefficiencies.

Performing multiply-add operations on the inserted zeros is ineffectual. Intuitively, the accelerator should skip over the zeros. Skipping the zeros creates an irregular flow of data and diminishes data reuse if not handled properly in the hardware. As such, reordering computation is necessary but creates divergences that diminish the benefits of SIMD execution. Furthermore, the zeros lead to distinct patterns of operations as the window slides regardless of the reordering or lack thereof. As such, the same sequence of operations cannot be repeated across all the compute engines, a requirement of SMID execution. Therefore, MIMD execution is inevitable but its overhead needs to be properly assessed and mitigated. As such, we propose an architecture that enables a grouping of compute engines to operate in SIMD mode while each group runs its own instruction. Since the SIMD units will be accessing divergent locations of storage, our architecture separates data retrieval from data processing within each individual compute engine.

We make this architecture a template design and devise a complete stack, dubbed FlexiGAN, that uses this template to generate an optimized synthesizable Verilog code for FPGA acceleration. FlexiGAN comes with a compilation workflow that starts from a high-level specification of GAN, reorders computation, optimizes flow of data, separates data retrieval, and generates a two-level hierarchy of instructions to accelerate a given GAN. Additionally, FlexiGAN includes an architecture builder that provides an algorithm to match the final accelerator design to the pair of (GAN, FPGA) specification. We use FlexiGAN to generate accelerators for six recent GAN models [6]–[11] for the Xilinx XCVU13P FPGA. We compare these generated accelerators to similarly optimized designs that only supports conventional convolution. FlexiGAN-generated designs provide 2.2× higher performance. We further compare these accelerators with a Titan X GPU, which shows 2.6× higher Performance-per-Watt. These results demonstrate that FlexiGAN is a promising initial step towards designing accelerators for the next generation of deep learning and artificial intelligence.

II. Related Work

While prior studies focus on the convolution operator for CNNs, FlexiGAN focuses on GANs and consolidates the acceleration of their two algorithmically distinct operators (*conventional convolution* and *transposed convolution*). The following overview the most related work.

NN acceleration. Inspiring work has developed ASIC [12], [14], [15], [21], [22], FPGA [13], [17], and analog [18], [19] CNN accelerators. Other research has explored dataflow optimization techniques for CNNs [20], [23]. The prime focus of these studies is the forward convolution operator. However, GANs, the next wave of emerging deep networks, involve transposed convolution. As mentioned earlier, while it is still possible to use CNN accelerators [12], [21] for GANs, the irregular insertion of zeros in transpose convolution leads to inefficiency and underutilization of resources. An arXiv paper [24] uses Vivado HLS tools to generate an implementation for deconvolution (transposed convolution) from high-level code and explores the effects of bitwidth reduction. As the paper does not offer an architecture, a head to head comparison is challenging. Additionally, the Vivado-generated implementation is merely for transposed convolution. In contrast to these prior works, FlexiGAN offers a customized holistic solution from high-level description to a synthesizable GAN accelerator that covers both conventional convolutions and transposed convolutions, and can efficiently operate for a wider range of deep neural networks, including previously unsupported GANs.

MIMD-SIMD processing. While prior work has studied the benefits of combined MIMD-SIMD acceleration for different applications [25]–[28], designs that combine these two models for deep neural networks, specifically GANs, are lacking. Our proposed solution is coupled with a low-overhead MIMD-SIMD architecture that enables switching between these two modes of parallelism at the granularity of each individual operation. This combination is particularly necessary for GANs as the accelerator needs to selectively skip over the irregularly inserted zeros in a transposed convolution. The accelerator reverts back to the full SIMD mode for the forward convolution.

Decoupled data retrieval and data processing. The seminal work by James Smith [29] introduces the concept of separating data retrieval from data processing. A number of designs [30], [31] follow this paradigm. Our proposed architecture differs from the prior work: (1) the data retrieval and data processing operations in these work are at the coarse granularity of kernels and functions and (2) they generally carry out the operations in a dataflow or SIMD manner. This work extends this paradigm to the finest granularity of computation for each individual accelerator operation.

III. Background and Motivation

Generative Adversarial Networks. GANs [1] combine game theory with recent advances in deep learning to generate realistic-looking synthetic data without manual intervention. Figure 1 illustrates a typical GAN, which consists of two neural networks, a generative model and a discriminative model. The generative model aims to generate synthetic data that deceives the discriminative model. Meanwhile, the discriminative model aims to distinguish whether its input data is synthetic or original. This competition improves both networks. To this end, the generated synthetic data becomes realistic without human intervention. The ability to generate realistic data enables novel applications. For instance, GANs can generate new scenarios for training autonomous cars [32] and explore alternative chemical processes for drug discovery [33]. Due to the growing importance of these novel deep learning algorithms, this paper sets out to design an end-to-end solution to generate optimized FPGA accelerators for GANs.

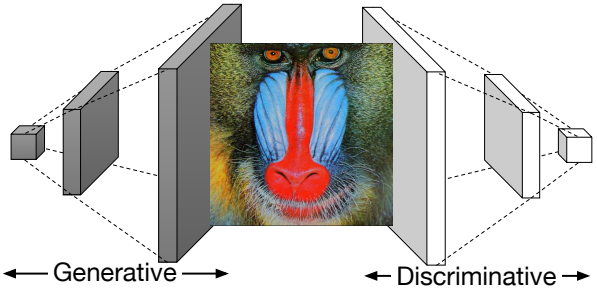

Figure 1: High-level visualization of GAN architecture.

Architecture challenges for GAN acceleration. The primary operation in generative models (transposed convolution or TranConv) fundamentally differs from the one in discriminative models (convolution or Conv). The Conv operation shrinks the input while TranConv expands it by first inserting zeros within its rows and columns, creating an intermediate input, and then sliding a window over this expanded input to perform a series of multiply-adds. The zero insertion makes TranConv and Conv inherently different from a hardware acceleration perspective as it leads to ineffectual operations (multiplications with zero) in conventional accelerators [12]–[14]. In addition, the pattern and number of ineffectual operations change as the TranConv window slides over the zero-inserted intermediate input. We address these challenges by introducing a novel accelerator architecture as a template design that follows three principles: (1) to mitigate the sources of inefficiency, the accelerator needs to properly handle the varied number of ineffectual multiplications per each sliding window; (2) to efficiently accelerate both generative and discriminative models on the same FPGA platform, the accelerator needs to be sufficiently general; and (3) to maximize data reuse for both transposed convolution as well as conventional convolution operations, the accelerator needs to properly handle the data movements between the compute units.

IV. Overview of FlexiGAN

Figure 2 illustrates the FlexiGAN framework. Our proposed workflow starts off by by taking in a high-level specification of a GAN, which is a JSON file that defines the structure of layers (e.g., the size of each layers, the num-

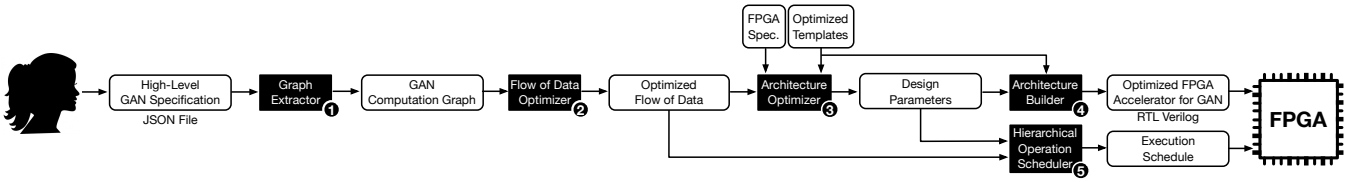

Figure 2: Overview of FlexiGAN end-to-end solution. FlexiGAN receives a high-level description of GAN and the target FPGA specification. At the end, it generates an optimized FPGA accelerator and the instruction schedules.

ber of kernels, etc.) for both generative and discriminative models. After going through the stages, overviewed in this section, FlexiGAN generates an optimized, synthesizable RTL Verilog code that can be readily deployed onto the target FPGA. The synthesizable code comes along with an execution schedule that governs the data movements between compute units, configures the address generators, and controls the order of operations to be executed. FlexiGAN comprises of the following components:

❶ **Graph Extractor.** The *graph extractor* converts the high-level GAN specification, written by the programmer, into a computation graph. Each node in the computation graph represents an operation, and each edge represents the data dependency between the nodes.

❷ **Flow of Data Optimizer.** The *flow of data optimizer* receives the generated compute graph and performs a series of data and operation reorderings to mitigate the inefficiencies introduced due to the zero-insertion step. We describe the challenges and the inefficiencies in GAN flow of data and our proposed solution to mitigate them in Section V.

❸ **Architecture Optimizer.** The *architecture optimizer* receives the optimized flow of data along with the target FPGA specification. The other input is a *template accelerator design* that is the focus of this paper and is detailed in Section VI. We develop a heuristic algorithm that minimizes the overall number of execution cycles by exploring various design parameters of the accelerator. Section VIII elaborates on the details of our heuristic algorithm.

❹ **Architecture Builder.** The *architecture builder* takes the design parameters along with the hand-optimized synthesizable implementation of the template design, such as data retrieval and data processing units, and automatically organizes these components into an optimized FPGA accelerator in RTL Verilog for the given GAN computation graph. The generated RTL Verilog code is deployed onto the target FPGA through commercial synthesis flows.

❺ **Hierarchical Operation Scheduler.** The generated design requires a set of operations and their schedule in order to accelerate GANs. The *hierarchical operation scheduler* fills this gap by taking in the design parameters along with the GAN computation graph as inputs, and generating the instructions for global and local instruction buffers and their schedule for the accelerator.

V. FlexiGAN Flow of Data Optimizer

The FlexiGAN Flow of Data Optimizer reorders operations in the transposed convolution layer to eliminate the ineffectual multiply-add operations.

Inefficiency of using convolution hardware to perform transposed convolution. Using an example, we first demonstrate the source of inefficiencies in performing transposed convolution, if a conventional convolution hardware is used. Figure 3 shows an example of performing a TranConv operation. This TranConv operation applies a 5×5 filter with stride of one and padding of two on a 4×4 2D input. As depicted in Figure 3(a), the TranConv operation inserts one row/column of zeros between successive rows/columns (white squares). Hence, the input is expanded from a 4×4 matrix to a 11×11 one. The next step is to slide the window and perform the multiply operations; Figure 3(a) is only illustrated for generating the output rows 2–5 to avoid clutter. For this step, if we use the typical convolution hardware, as shown in Figure 3(b), there will be ineffectual operations (white squares). The black squares represent the compute units that perform effectual operations. Each square (compute node) performs a vector-vector multiplication between a row of the filter and a row of the zero-inserted input. The filter rows are spatially reused across all the squares in a vertical manner while they work in parallel and generate the partial sums. Then, the squares horizontally aggregate these partial sums while performing the remaining row to row multiplications. Because of zero-insertion, some of the filter rows are not used to compute an output row. For instance, since the 1st, 3rd, and 5th rows of the corresponding input are zero, the 2nd output row only uses the 2nd and 4th filter rows. This example highlights the following three sources of inefficiency when a conventional Conv hardware is used to perform TranConv. First, the presence of non-zero rows between distinct outputs causes a significant number of compute nodes to remain idle. Second, due to the inserted zero columns in the input, a large fraction of multiply-add operations will be ineffectual within each node. Finally, the inserted zeros diminish the benefits of data reuse along the filter rows.

FlexiGAN flow of data for generative models. FlexiGAN addresses these inefficiencies by changing the flow of data. The example in Figure 3 shows that there are only two distinct computation patterns. The even rows use one pattern while the odd rows share another, which is clear from the location of the black and white squares in Figure 3. We leverage this repetition of patterns to optimize the flow of data to overcome the aforementioned challenges. The first optimization reorganizes output rows, maximizing data reuse by making output rows with the same computation pattern adjacent. Figure 4(a) illustrates the computation patterns of the example after applying the output row reorganization. As shown, the even-indexed output rows (2 and 4) become ad-

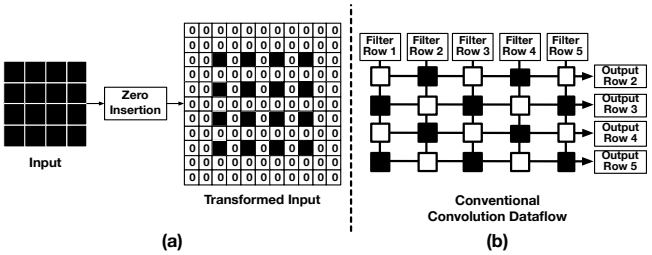

Figure 3: (a) Zero-insertion step in TranConv operation for a 4×4 input and the transformed input. The white-colored squares represent zero values in the transformed input. (b) Using convolution dataflow for performing TranConv operations.

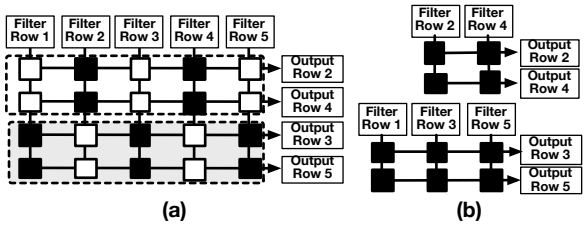

Figure 4: The flow of data after applying (a) output row reorganizaton and (b) filter row reorganization. The combination of these flow optimizations reduces the idle (white) operations and improves the resource utilization.

jacent. Similarly, the odd-indexed rows (3 and 5) are placed adjacent to each other. Although this optimization addresses the data reuse problem, it does not deal with the resource underutilization. To overcome resource underutilization, we introduce the second optimization, which reorganizes the filter rows as depicted in Figure 4(b). After applying this optimization, the 1st, 3rd, and 5th rows become adjacent. Similarly, the 2nd and 4th filter rows are placed adjacent. As shown in Figure 4(b), the combination of output row and filter row reorganization in the example effectively improves the resource utilization for the TranConv computation from 50% to 100%. Furthermore, since there are no squares between the compute nodes, the horizontal accumulation of the partial sums does not incur wasted cycles. Figure 3(b) originally needs five cycles to perform the horizontal accumulation for each output row, no matter odd- or even-indexed. Following these optimizations, the number of cycles is reduced from five to two for the even-indexed rows and from five to three for the odd-indexed rows, as can be seen in Figure 4(b). Due to this discrepancy, if a pure SIMD execution model is used, there will still be resource underutilization. The next section discusses the architecture used in FlexiGAN solution that combines SIMD and MIMD execution to overcome this issue.

VI. FlexiGAN Template Architecture

The FlexiGAN solution is coupled with a novel template architecture which aims to harness the benefits of both MIMD and SIMD execution models. A pure SIMD execution model works for conventional convolution because the number and pattern of multiply-add operations are uniform and identical across the sliding windows. However, the inserted zeros in the transposed convolution results in variation of consequential multiply-adds across the sliding windows.

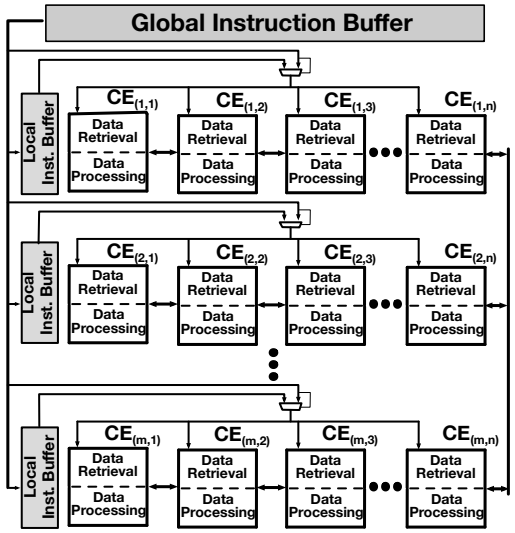

Figure 5: Top-level block diagram of the proposed architecture.

This variation imposes irregularities on data retrieval and computation patterns. Nonetheless, there are still repeated patterns that we organize across distinct groups with our two aforementioned optimizations. The SIMD execution best suits the computation across the CEs in each row, while the MIMD model enables concurrent execution between rows. As such, the template-based architecture combines MIMD and SIMD models.

Figure 5 illustrates a high-level diagram of template architecture for GAN acceleration. This design comprises a memory hierarchy and a set of identical Compute Engines (CE) organized as a (m×n) 2D array wherein each CE is connected to its adjacent CEs in the same row. Prior to execution, the scheduling of operations is determined statically and preloaded to the global instruction buffer. Each CE in the proposed architecture carries out the computation of one filter row and one input row. The generated partial sums in each CE are aggregated spatially across the vertical CEs and produce the final output value for each sliding window. The CEs consist of two units, a data retrieval unit and a data processing unit. The data retrieval unit performs memory access operations, while the data processing unit carries out primitive operations such as addition, multiply-add, and multiplication. Each data processing unit has a scratchpad register file that is used to store the intermediate temporary values. We expound the microarchitectural details of data retrieval and data processing units in Section VI.

The 2D organization of CEs is best suited for performing the Conv operations across distinct sliding windows, conforming to a SIMD execution model. The global instruction buffer supplies an instruction across all the CEs and each of them performs an operation defined by the received instruction on their own local data. However, following the same SIMD execution model for performing the computations of TranConv leads to resource underutilization. The CEs that perform fewer effectual operations will be sitting idle, wasting on-chip resources. A primary solution to avert the

resource underutilization problem for TranConv operations is to switch to a full MIMD execution model and utilize the parallelism across the sliding windows, which may have a different number of operations. Since a full MIMD solution necessitates the augmentation of each CE with a dedicated instruction buffer, this approach is costly, particularly for FPGAs with limited resources. Furthermore, pushing the accelerator design towards a fully MIMD execution model significantly increases the intrinsic von Neumann overhead of instruction fetch and decode. To minimize these overheads, we leverage the observation that, in TranConv operations, the number of sliding windows with distinct data and compute patterns is limited. We group the sliding windows with identical patterns into distinct compute groups. The computations for each compute group are performed by the CEs in the same row. Instead of pushing the accelerator towards a fully MIMD solution, we leverage the algorithmic property of TranConv operations and embrace a middle ground between MIMD and SIMD execution models. Using this approach enables us to increase the on-chip resource utilization and deliver the same level of efficiency as conventional SIMD accelerator for Conv operations.

Two-level instruction buffers. As shown in Figure 5, the global instruction buffer is shared across all the CEs. Each horizontal group of CEs has a dedicated local instruction buffer. At a given cycle, each CE executes an instruction from either the local or global instruction buffer. A 1-bit field in the global instruction indicates whether the accelerator operates in the SIMD mode or MIMD-SIMD mode at the current cycle. In the SIMD mode, the global instruction buffer bypasses the local instruction buffers and broadcasts the instructions globally to all the CEs. In the MIMD-SIMD mode, the global instruction buffer, instead of an instruction, sends an index to each local instruction buffer. Upon receiving the index, each local instruction buffer independently fetches an instruction and broadcasts it to the CEs of the corresponding horizontal group. This organization of instruction buffers enables switching between SIMD and MIMD-SIMD mode in a cycle. While this approach alleviates the resource underutilization issues for TranConv operations, it raises another challenge to our proposed design, i.e., the size of operation buffers. To reduce this overhead, we leverage the insight that operations share similarity across the sliding windows and devise separated data retrieval and data processing units within each CE.

Separated data retrieval and data processing architecture. Figure 6a depicts the organization of the proposed separated data retrieval and data processing architecture. The data retrieval unit produces the addresses for the input, weight, and output buffers, while the data processing unit performs an operation on data from the input and weight buffers, and sends the result back through the output buffer. While the instruction streams for data retrieval and data processing units are completely separated, these units work in tandem to perform an instruction using the FIFOs placed within their boundaries. While the instruction streams for data retrieval units consists of instructions to configure and

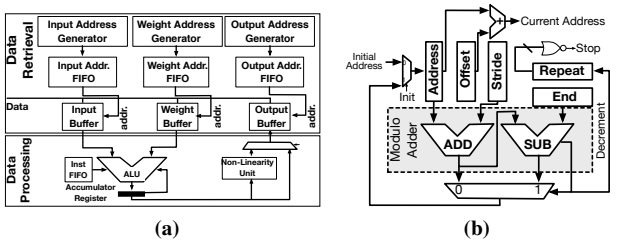

Figure 6: (a) Organization of separated data retrieval and data processing architecture and (b) address generator. The Current Address **signal indicates the current generated address.**

initialize the address generators, the instruction stream for data processing units *merely* specifies the type of computation to be performed on the retrieved data. As such, the data processing instructions *do not* have fields to specify the operand addresses which enables them to reuse the same instruction across many cycles without the need for extra storage, which is a limited resource in FPGAs.

Address generator. We observe that data accesses in both Conv and TranConv are either sequential or strided. The stride value in the strided data access pattern varies across different layers of GAN. These sequential and strided data retrieval patterns are also repeated across a large number of sliding windows. We leverage these insights to simplify the design of the address generator in the data retrieval unit. Figure 6b shows the block diagram of the proposed address generator. The End register in the illustration (Figure 6b) controls the size of the generated range of addresses, while the Offset register adjusts the final values of this range as necessary. The value in the Stride register indicates the step size between two consecutive data addresses in the strided access pattern. Finally, the Repeat register specifies the number of times this pattern of addresses should be replayed. The modulo adder enables the address generator to generate addresses in a rotating manner. Once these configuration registers are initialized via a series of data retrieval instructions, the address generator can yield one address every cycle. When the value of the Repeat register reaches zero, the Stop signal is activated and the process of generating addresses halts.

VII. FlexiGAN Instruction Set Architecture

Limited on-chip memory in FPGAs poses a challenges in designing accelerators. To reduce on-chip storage requirment, we design a succinct set of instructions that suits the FPGA acceleration of GANs. These instructions are grouped into data retrieval instructions and data processing instructions. The separation between the data retrieval and data processing instructions further enables reducing the instruction memory footprint. The following lists the major instructions of the architecture.

A. Data Retrieval Instructions

The first set of instructions handles address generation. The instruction-word consists of OP, DST, and DATA. The OP value determines the instruction type and the other two fields depend on the type and are elaborated below.

Instruction write**.** This instruction configures the address generator by storing DATA, an immediate value (e.g., start

address, end address, stride, etc.), into a configuration register, identified by the DST field.

Instruction run_partial**.** This instruction is a loop instruction that generates addresses for a specified number of iterations, specified by the DATA field.

Instruction run_full**.** This instruction is also a loop one and runs all of the corresponding instructions in the local instruction buffer.

B. Data Processing Instructions

The next set of instructions handles data processing. The instruction-word consists of ALU_OP and NL_Sel. ALU_OP determines the operation, and NL_Sel enables or disables applying the nonlinearity function, which is a Rectifying Linear Unit (ReLU). In all instructions, the resulting value is written to the output register in the data processing unit.

Instruction mul**.** This instruction performs a multiplication.

Instruction mul_nl**.** This instruction performs a multiplication. The result of the multiplication is then passed through the nonlinearity unit (See figure 6a).

Instruction mul_add**.** This instruction performs a multiplication, then adds the result to the content of Accumulator register (See Figure 6a).

Instruction mul_add_nl**.** This instruction performs a multiplication, then adds the result to the content of Accumulator register. The updated value of the Accumulator register is then passed through the nonlinearity unit (See Figure 6a).

VIII. FlexiGAN Architecture Optimizer

Algorithm 1 illustrates the heuristic architecture optimizer (❸ in Figure 2) that attempts to minimize the overall number of execution cycles for a given GAN specification. The algorithm takes as input the GAN optimized flow of data, the FPGA hardware constraints, the size of local instruction buffer, and the FPGA resource utilization of one CE. After iterating over all the valid design points, the algorithm returns the number of rows in the accelerator and the number of CEs per each individual row. Note that, the size of local instruction buffers is fixed and determined by the maximum number of instructions supported in the architecture. Procedure FindMaxNumCE takes in the FPGA resource constraints, such as number of DSP units and size of on-chip memories, along with the hardware implementation of the CE, and returns the maximum number of CEs that the given FPGA hardware can support. Procedure CheckFeasibility prunes the search space for possible design points. The procedure receives as input the FPGA resource constraints, the current number of rows in the accelerator, the current number of CEs per each row, and the size of local instruction buffer per each row. Then, if the total size of local instruction buffers is greater than the size of the on-chip memories, the procedure returns false, indicating that this design point is not valid. Finally, EvalDesign takes in as input the GAN computation graph, the FPGA resource constraints, the current number of rows, the current number of CEs per row, and the size of local instruction buffer. The procedure estimates the number of execution cycles, should the given GAN computation graph is run on the FPGA with the current choices of the architecture parameters. If the current estimated number of execution cycles is less than ExecCycles, the algorithm records the new choice.

Algorithm 1 FlexiGAN Architecture Optimizer

Inputs:
G: GAN Optimized Flow of Data
F: FPGA Constraints
$LocalOpSize$: Size of local instruction buffer
$CE_{Hardware}$: FPGA resource utilization for one CE
Outputs:
$NumRows$: Number of Rows
$NumCEperRows$: Number of CEs per each Row
Arg min:
$ExecCycles$: Estimated number of execution cycles

```
ExecCycles = ∞;
NumCEperRow = 1;
F.max_CEs = FindMaxNumCEs(F, CE_Hardware);
for CEpRow in range (1, F.max_CEs) do
    NumRow = F.max_CEs / CEpRow;
    if CheckFeasibility(F, NumRow, CEpRow, LocalOpSize) then
        Cycles = EvalDesign(G, F, NumRow, CEpRow, LocalOpSize);
        if (Cycles < ExecCycles) then
            ExecCycles = Cycles;
            NumCEperRow = CEpRow;
        end if
    end if
end for
return NumRow, NumCEperRow
```

IX. Evaluation and Methodology

We evaluate the benefits of the generated FPGA accelerator with FlexiGAN using an FPGA chip (Xilinx XCVU13P) and a GPU platform (Titan X). We implement an optimized convolution accelerator based on Eyeriss [12], and synthesize it on the same FPGA. We refer to the FPGA implementation of our accelerator and the conventional accelerator as FlexiGAN-FPGA and Conv-FPGA, respectively. Table II summarizes the the FPGA and GPU specifications.

A. Experimental Setup

Benchmarks. We use several state-of-the-art GANs to evaluate the effectiveness of FlexiGAN, including 3D-GAN [6], ArtGAN [7], DCGAN [8], DiscoGAN [9], GP-GAN [10], and MAGAN [11]. Table I details the evaluated GAN models. These GAN models are used for various applications including text-to-image synthesis, high-resolution image generation, music synthesis, and 3D object generation.

Hardware design. We implement the microarchitectural units of our proposed architecture including the address generator, the CEs, global and local controllers, FIFOs, and buffers in Verilog. All arithmetic operations are performed with 16-bit precision. As such, we design and implement the ALU units with 16-bit fixed-point precision.

FPGA synthesis. We use 64-bit Vivado Design Suite v2017.2 to synthesize the FlexiGAN and conventional accelerator hardware implementations. The global and local instruction buffers are implemented with BRAMs and UltraRAMs. The

Table I: Evaluated GANs, their release year, application, and the number of transposed convolution (TranConv and convolution (Conv)) layers.

Name	Year	Description	# Conv	# TranConv
3D-GAN	2016	Generates 3D objects	5	4
ArtGAN	2017	Generates complex artworks	6	5
DCGAN	2015	Generates bedroom images	4	4
DiscoGAN	2017	Discovers the cross-domain relations between pair of images	10	4
GP-GAN	2017	Generates high-resoultion realisitc images	5	4
MAGAN	2017	Proposes a novel training procedue for GANs	6	12

Table II: FPGA (XCVU13P) and GPU (Titan X) specification.

	FPGA (XCVU13P)		GPU (GTX Titan X)
Logic Cells (K)	747	Cores	3072
Flip-Flops (K)	3456	Frequency (GHz)	1
LUTs (K)	1728	Memory (GB)	12
Total BRAM (Mb)	94.5	Memory Clock (GHz)	6.6
UltraRAM (Mb)	360	Technology (nm)	28
DSP Slices	12288	Platform	CUDA V8.0.44

Table III: The resource utilization of the target FPGA (XCVU13P), for DC-GAN generated with FlexiGAN workflow.

	LUTs (K)	BRAMs (Mb)	UltraRAMs (Mb)	DSP Slices
Used	1325	88	280	1560
Available	1728	94.5	360	12,288
Utilization	77%	93%	78%	13%

synthesis tool maps the FIFOs and registers to Flip-Flops. The arithmetic units, multiplexers, and other logics are implemented with DSP slices and LUTs. We use a similar methodology to synthesize the optimized conventional accelerator on an FPGA (Conv-FPGA). The frequency of operation in both FlexiGAN-FPGA and Conv-FPGA is ≈190 MHz.

GPU baseline. We use TensorFlow v1.4.1, which uses prebuilt highly-optimized binaries with NVIDIA CUDA DNN library (cuDNN 6.0). We also use the Nvidia CUDA Compiler (NVCC) v8.0.44 with maximum compiler optimizations enabled. We measure the execution time for GPU implementation by measuring the wall clock time, averaged over 10 runs. We use the Nvidia Management Library (NVML) to obtain the average power while running each GAN model.

B. Experimental Results

FPGA resource utilization. Table III shows the XCVU13P resource utilization of the generated accelerator for DC-GAN using FlexiGAN workflow. The resource utilization is limited by the amount of available on-chip memory. The number of entries for global and local instruction buffers are set to 512 and 128, respectively. Finally, we fix the number of entries for each of the data retrieval FIFOs and buffers between data retrieval and data processing units to 32. Under this setting and due to the limited size of on-chip memories, we could only map 1,560 CEs on our evaluated FPGA.

Speedup. Figure 7 illustrates the speedup of our accelerator (FlexiGAN-FPGA) and GPU Titan X normalized to Conv-FPGA. On average, FlexiGAN-FPGA offers 2.2× higher speedup compared to Conv-FPGA. The major source of speedup is FlexiGAN's ability to efficiently bypass the computation of a large percentage of zero values, in TranConv layers. Compared to Titan X, the number of compute nodes and the clock frequency in FlexiGAN are ≈2.0× and ≈5.0× lower, respectively. However, due to the novel design of the template architecture for bypassing ineffectual operations, FlexiGAN, on average, experiences only a 2.2× slowdown.

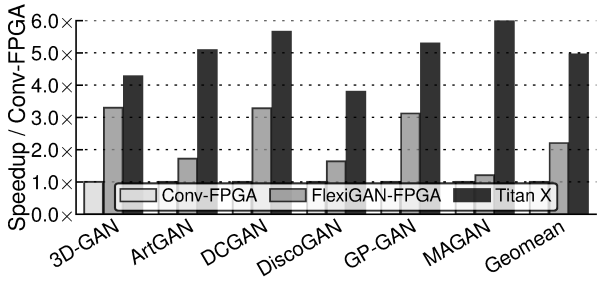

Figure 7: Speedup with FlexiGAN-FPGA and GPU (Titan X) vs Conv-FPGA.

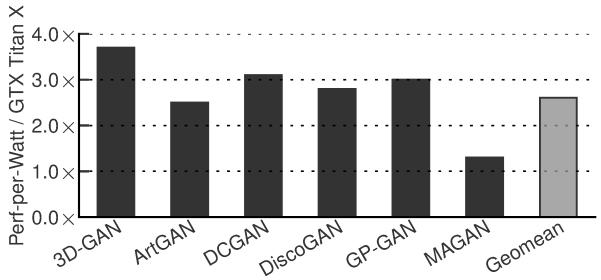

Figure 8: Performance-per-Watt, FlexiGAN-FPGA over GPU (Titan X).

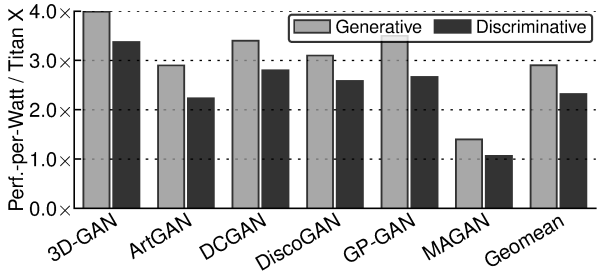

Figure 9: The performance-per-watt comparison of FlexiGAN-FPGA over GPU for generative and discriminative models.

Performance-per-Watt. As expected, FlexiGAN-FPGA outperforms the Conv-FPGA in Performance-per-Watt with the same pattern as the speedup trends, since they are both deployed on the same FPGA platform and FlexiGAN-FPGA is significantly faster. Nonetheless, to provide a better picture off the advantages for FlexiGAN, Figure 8 shows the Performance-per-Watt improvement with FlexiGAN over the high-end GPU baseline for each GAN model. The average performance-per-watt improvement for FlexiGAN is 2.6×. FlexiGAN yields this significant energy efficiency by eliminating the ineffectual operations and minimizing the von Neumann overhead of instruction fetch and decode.

Per-model Performance-per-Watt. To further analyze the potential benefits of our proposed GAN accelerator, we measure the FlexiGAN-FPGA Performance-per-Watt over GPU per generative and discriminative models. Figure 9 shows the FlexiGAN-FPGA Performance-per-Watt over GPU for generative and discriminative models. FlexiGAN-FPGA delivers 2.9× and 2.3× Performance-per-Watt for generative and discriminative models, respectively. GPUs do not have a

mechanism to bypass ineffectual operations. As such, compared to a discriminative model, the generative models, in which a large fraction of operations are ineffectual, enjoy a higher Performance-per-Watt with FlexiGAN-FPGA.

X. Conclusion

Due to the algorithmic properties of transposed convolution and the inherent irregularities in its computation, using the conventional convolution accelerators for GANs leads to inefficiencies and underutilization of resources. To alleviate these issues, the paper devised FlexiGAN, an end-to-end solution, from high-level description to an optimized FPGA accelerator for GANs. The proposed solution comes with a novel template architecture that combines MIMD and SIMD models while separating data retrieval and data processing units at the finest granularity possible. Leveraging the separated data retrieval-data processing architecture, we introduce a succinct set of operations that enables us to significantly reduce the on-chip memory usage, which is generally a limited resource in FPGA platforms. Evaluation with a variety of GANs shows that FlexiGAN-generated accelerators, on average, provide 2.2× higher performance than an optimized conventional accelerator design. Compared to a Titan X GPU, these accelerators provide 2.6× better Performance-per-Watt, averaged across the benchmark GANs. The benefits are 2.9× for the generative models and 2.3× for the discriminative models.

XI. Acknowledgments

We thank Hardik Sharma and Ecclesia Morain for insightful discussions that greatly improved the manuscript. Amir Yazdanbakhsh is supported by a Microsoft Research PhD Fellowship. This work was in part supported by NSF awards CNS#1703812, ECCS#1609823, CCF#1553192, Air Force Office of Scientific Research (AFOSR) Young Investigator Program (YIP) award #FA9550-17-1-0274, and gifts from Google, Microsoft, Xilinx, and Qualcomm.

References

[1] I. Goodfellow *et al.*, "Generative Adversarial Nets," in *NIPS*, 2014.
[2] J. Ho and S. Ermon, "Generative Adversarial Imitation Learning," in *NIPS*, 2016.
[3] A. Ghosh *et al.*, "SAD-GAN: Synthetic Autonomous Driving using Generative Adversarial Networks," in *arXiv*, 2016.
[4] H.-W. Dong, W.-Y. Hsiao, L.-C. Yang, and Y.-H. Yang, "MuseGAN: Symbolic-domain Music Generation and Accompaniment with Multi-track Sequential Generative Adversarial Networks," in *arXiv*, 2017.
[5] D. Nie *et al.*, "Medical Image Synthesis with Context-aware Generative Adversarial Networks," in *International Conference on Medical Image Computing and Computer-Assisted Intervention*, 2017.
[6] J. Wu *et al.*, "Learning a Probabilistic Latent Space of Object Shapes via 3D Generative-Adversarial Modeling," in *NIPS*, 2016.
[7] W. R. Tan *et al.*, "ArtGAN: Artwork Synthesis with Conditional Categorial GANs," in *arXiv*, 2017.
[8] A. Radford *et al.*, "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks," in *arXiv*, 2015.
[9] T. Kim *et al.*, "Learning to Discover Cross-Domain Relations with Generative Adversarial Networks," in *arXiv*, 2017.
[10] H. Wu *et al.*, "GP-GAN: Towards Realistic High-Resolution Image Blending," in *arXiv*, 2017.
[11] R. Wang *et al.*, "MAGAN: Margin Adaptation for Generative Adversarial Networks," in *arXiv*, 2017.
[12] Y.-H. Chen *et al.*, "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks," in *ISCA*, 2016.
[13] H. Sharma *et al.*, "From High-Level Deep Neural Models to FPGAs," in *MICRO*, 2016.
[14] T. Chen *et al.*, "DianNao: A Small-footprint High-throughput Accelerator for Ubiquitous Machine-learning," in *ASPLOS*, 2014.
[15] S. Han *et al.*, "EIE: Efficient Inference Engine on Compressed Deep Neural Network," in *ISCA*, 2016.
[16] D. Mahajan *et al.*, "TABLA: A Unified Template-based Framework for Accelerating Statistical Machine Learning," in *HPCA*, 2016.
[17] T. Moreau *et al.*, "SNNAP: Approximate Computing on Programmable SoCs via Neural Acceleration," in *HPCA*, 2015.
[18] R. S. Amant *et al.*, "General-Purpose Code Acceleration with Limited-Precision Analog Computation," in *ISCA*, 2014.
[19] A. Shafiee *et al.*, "ISAAC: A Convolutional Neural Network Accelerator with In-situ Analog Arithmetic in Crossbars," in *ISCA*, 2016.
[20] W. Lu *et al.*, "FlexFlow: A Flexible Dataflow Accelerator Architecture for Convolutional Neural Networks," in *HPCA*, 2017.
[21] A. Parashar *et al.*, "SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks," in *ISCA*, 2017.
[22] A. Yazdanbakhsh *et al.*, "Neural Acceleration for GPU Throughput Processors," in *MICRO*, 2015.
[23] L. Song *et al.*, "PipeLayer: A Pipelined ReRAM-based Accelerator for Deep Learning," in *HPCA*, 2017.
[24] X. Zhang *et al.*, "A Design Methodology for Efficient Implementation of Deconvolutional Neural Networks on an FPGA," in *arXiv*, 2017.
[25] H. J. Siegel *et al.*, "PASM: A Partitionable SIMD/MIMD System for Image Processing and Pattern Recognition," *IEEE TC*, no. 12, pp. 934–947, 1981.
[26] A. Nieto *et al.*, "PRECISION: A reconfigurable SIMD/MIMD coprocessor for Computer Vision Systems-on-Chip," *IEEE TC*, vol. 65, no. 8, pp. 2548–2561, 2016.
[27] H. M. Waidyasooriya *et al.*, "FPGA Implementation of Heterogeneous Multicore Platform with SIMD/MIMD Custom Accelerators," in *ISCAS*, 2012.
[28] X. Wang and S. G. Ziavras, "Performance-energy Tradeoffs for Matrix Multiplication on FPGA-based Mixed-mode Chip Multiprocessors," in *ISQED*, 2007.
[29] J. E. Smith, "Decoupled Access/Execute Computer Architectures," in *ACM SIGARCH Computer Architecture News*, 1982.
[30] T. Nowatzki *et al.*, "Stream-Dataflow Acceleration," in *ISCA*, 2017.
[31] K. Wang and C. Lin, "Decoupled Affine Computation for SIMT GPUs," in *ISCA*, 2017.
[32] Y. Li *et al.*, "Inferring The Latent Structure of Human Decision-Making from Raw Visual Inputs," in *arXiv*, 2017.
[33] M. Benhenda, "ChemGAN Challenge for Drug Discovery: Can AI Reproduce Natural Chemical Diversity?," in *arXiv*, 2017.