Appears in the Proceedings of the 45th International Symposium on Computer Architecture (ISCA), 2018

GANAX: A Unified MIMD-SIMD Acceleration for Generative Adversarial Networks

Amir Yazdanbakhsh Kambiz Samadi[‡] Nam Sung Kim[♭] Hadi Esmaeilzadeh[§]
Alternative **C**omputing **T**echnologies (**ACT**) Lab
Georgia Institute of Technology [‡]Qualcomm Technologies, Inc. [♭]University of Illinois at Urbana-Champaign [§]UC San Diego
a.yazdanbakhsh@gatech.edu ksamadi@qti.qualcomm.com nskim@illinois.edu hadi@eng.ucsd.edu

***Abstract*— Generative Adversarial Networks (GANs) are one of the most recent deep learning models that generate synthetic data from limited genuine datasets. GANs are on the frontier as further extension of deep learning into many domains (e.g., medicine, robotics, content synthesis) requires massive sets of labeled data that is generally either unavailable or prohibitively costly to collect. Although GANs are gaining prominence in various fields, there are no accelerators for these new models. In fact, GANs leverage a new operator, called transposed convolution, that exposes unique challenges for hardware acceleration. This operator first inserts zeros within the multidimensional input, then convolves a kernel over this expanded array to add information to the embedded zeros. Even though there is a convolution stage in this operator, the inserted zeros lead to underutilization of the compute resources when a conventional convolution accelerator is employed. We propose the GANAX architecture to alleviate the sources of inefficiency associated with the acceleration of GANs using conventional convolution accelerators, making the first GAN accelerator design possible. We propose a reorganization of the output computations to allocate compute rows with similar patterns of zeros to adjacent processing engines, which also avoids inconsequential multiply-adds on the zeros. This compulsory adjacency reclaims data reuse across these neighboring processing engines, which had otherwise diminished due to the inserted zeros. The reordering breaks the full SIMD execution model, which is prominent in convolution accelerators. Therefore, we propose a unified MIMD-SIMD design for GANAX that leverages repeated patterns in the computation to create distinct microprograms that execute concurrently in SIMD mode. The interleaving of MIMD and SIMD modes is performed at the granularity of single microprogrammed operation. To amortize the cost of MIMD execution, we propose a decoupling of data access from data processing in GANAX. This decoupling leads to a new design that breaks each processing engine to an access micro-engine and an execute micro-engine. The proposed architecture extends the concept of access-execute architectures to the finest granularity of computation for each individual operand. Evaluations with six GAN models shows, on average, 3.6× speedup and 3.1× energy savings over EYERISS without compromising the efficiency of conventional convolution accelerators. These benefits come with a mere ≈7.8% area increase. These results suggest that GANAX is an effective initial step that paves the way for accelerating the next generation of deep neural models.**

***Keywords*-Generative Adversarial Networks; GAN; Accelerators; Dataflow; SIMD-MIMD; Deep Neural Networks; DNN; Convolution Neural Networks; CNN; Transpoed Convolution; Access-Execute Architecture**

I. Introduction

Deep Neural Networks (DNNs) have been widely used to deliver unprecedented levels of accuracy in various applications. However, they rely on the availability of copious amount of labeled training data, which can be costly to obtain as it requires human effort to label. To address this challenge, a new class of deep networks, called Generative Adversarial Networks (GANs), have been developed with the intention of automatically generating larger and richer datasets from a small initial labeled training dataset. GANs combine a generative model, which attempts to create synthetic data similar to the original training dataset, with a discriminative model, a conventional DNN that attempts to discern if the data produced by the generative model is synthetic, or belongs to the original training dataset [1]. The generative and discriminative models compete with each other in a minimax situation, resulting in a stronger generator and discriminator. As such, GANs can create new impressive datasets that are hardly discernible from the original training datasets. With this power, GANs have gained popularity in numerous domains, such as medicine, where overtly costly human-centric studies need to be conducted to collect relatively small labeled datasets [2], [3]. Furthermore, the ability to expand the training datasets has gained considerable popularity in robotics [4], autonomous driving [5], and media synthesis [6]–[12] as well.

Currently, advances in acceleration for conventional DNNs are breaking the barriers to adoption [13]–[18]. However, while GANs are set to push the frontiers in deep learning, there is a lack of hardware accelerators that address their computational needs. This paper sets out to explore this state-of-the-art dimension in deep learning from the hardware acceleration perspective. Given the abundance of the accelerators for conventional DNNs [15]–[43], designing an accelerator for GANs will only be attractive if they pose new challenges in architecture design. By studying the structure of emerging GAN models [6]–[12], we observe that they use a fundamentally different type of mathematical operator in their generative model, called *transpose convolution*, that operates on multidimensional input feature maps.

The transposed convolution operator aims to extrapolate information from input feature maps, in contrast to the conventional convolution operator which aims to interpolate the most relevant information from input feature maps. As such, the transposed convolution operator first inserts zeros within multidimensional input feature maps and then convolves a kernel over this expanded input to augment information to the inserted zeros. The transposed convolution in GANs fundamentally differs from the operators in the backward pass of training conventional DNNs, as these do not insert zeros. Moreover, although there is a convolution stage in the transposed convolution operator, the inserted zeros lead to underutilization of the compute resources if a conventional convolution accelerator were to be used. The following highlights the sources of underutilization and outlines the contributions of this paper, making the first accelerator design for GANs.

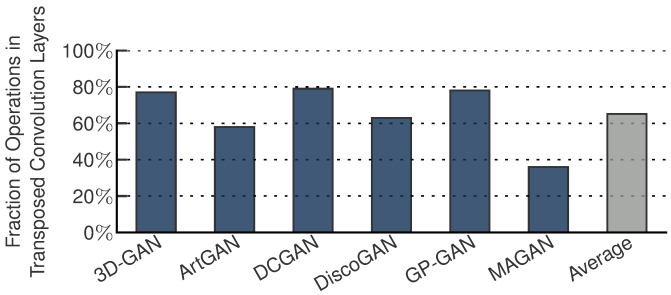


Figure 1: The fraction of multiply-add operations in transposed convolution layers that are inconsequential due to the inserted zeros in the inputs.

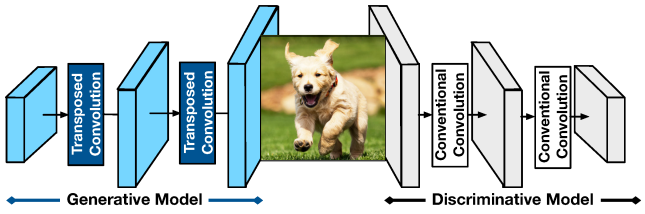
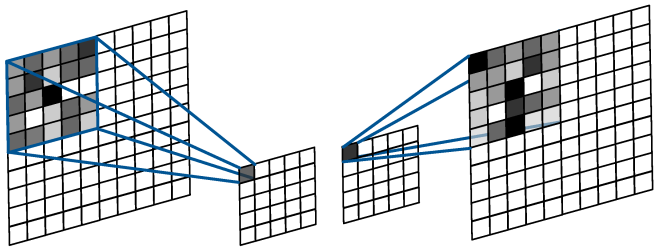


Figure 2: High-level visualization of a Generative Adversarial Network (GAN).

1) **Performing multiply-add on the inserted zeros is inconsequential.** Unlike conventional convolution, the accelerator should skip over the zeros as they constitute more than 60% of all the multiply-add operations as Figure 1 illustrates. Skipping the zeros creates an irregular dataflow and diminishes data reuse if not handled adequately in the microarchitecture. To address this challenge, we propose a reorganization of the output computations that allocates computing rows with similar patterns of zeros to adjacent processing engines. This forced adjacency reclaims data reuse across these neighboring compute units.
2) **Reorganizing the output computations is inevitable but breaks the SIMD execution model.** The inserted zeroes, even with the output computation reorganization, create distinct patterns of computation when sliding the convolution window. As such, the same sequence of operations cannot be repeated across all the processing engines, breaking the full SIMD execution model. Therefore, we propose a unified MIMD-SIMD accelerator architecture that exploits repeated patterns in the computations to create different microprograms that can execute concurrently in SIMD mode. To maximize the benefits from both levels of parallelism, we propose an architecture, called GANAX, that supports interleaving MIMD and SIMD operations at the finest granularity of a single microprogrammed operation.
3) **MIMD is inevitable but its overhead needs to be amortized.** Changes in the dataflow and the computation order necessitate irregular accesses to multiple different memory structures while the operations are still the same. That is, the data processing part can be SIMD but the irregular data access patterns prevent using this execution model. For GANAX, we propose the decoupling of data accesses from data processing. This decoupling leads to breaking each processing engine into an access micro-engine and an execute micro-engine. The proposed architecture extends the concept of access-execute architecture [44]–[47] to the finest granularity of computation for each individual operation. Although GANAX addresses these challenges to enable efficient execution of the transposed convolution operator, it does not impose extra overhead, but instead offers the same level of performance and efficiency. To establish the effectiveness of our architectural innovation, we evaluate GANAX using six recent GAN models, on distinct applications. On average, GANAX delivers 3.6× speedup and 3.1× energy savings over a conventional convolution accelerator. These results indicate GANAX is an effective step towards designing accelerators for the next generation of deep networks.



(a) Conventional Convolution (Data Reduction)

(b) Transposed Convolution (Data Expansion)

Figure 3: (a) Convolution operations decreases the size of data (data reduction). (b) Transposed convolution increases the size of data (data expansion).

II. Flow of Data in Generative Models

Generative Adversarial Networks (GANs) have revolutionized modern machine learning by significantly improving generative models while using only limited number of labeled training data. Figure 2 shows an overall visualization of a GAN, consisting of two deep neural network models, a generative model and a discriminative model. These two neural network models oppose each other in a minimax situation. Specifically, the generative model tries to generate data that will trick the discriminative model to believing the data is from the original training dataset. Meanwhile, the discriminative model is handed data from either the generative model or the training data and tries to discern between the two. After these networks compete with each other, they refine their abilities to generate and discriminate, respectively. This process creates a stronger generative model and discriminative model than could be obtained otherwise [1]. This arrangement of neural networks has opened up many applications, some of which include music generation with accompaniment [9] and the discovery new drugs to cure diseases [48]. GANs are enabling our future by pushing forward development in autonomous vehicles, allowing us to imitate human drivers [49] and simulate driving scenarios to save testing and training costs [5]. GANs enable imagination [50], a major advancement for machine learning and a key step towards true general artificial intelligence. Here, we overview the challenges and opportunities that were encountered while designing hardware accelerators for GANs.

Challenges and opportunities for GAN acceleration. The generative models in GANs are fundamentally different from the discriminative models. As Figure 2 illustrates, while the discriminative

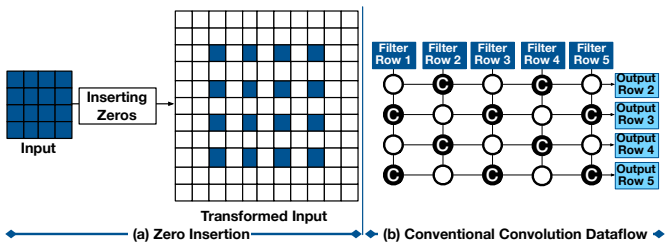


Figure 4: (a) Zero-insertion step in a transposed convolution operation for a 4×4 input and the transformed input. The light-colored squares display zero values in the transformed input. (b) Using conventional dataflow for performing a transposed convolution operation.

model mostly consists of convolution operations, the generative model uses transposed convolution operations. Accelerating convolution operations has been the focus of a handful of studies [15]–[17], [19]–[36]; however, accelerating transposed convolution operations has remained unexplored. Figure 3 depicts the fundamental difference between the conventional convolution and transposed convolution operations. The convolution operation performs *data reduction* and generally transforms the input data to a smaller representation. On the other hand, the transposed convolution implements a *data expansion* and transforms the input data to a larger representation. The transposed convolution operation expands the data by first transforming the input data through inserting zeros between the input rows and columns and then performing the computations by sliding a convolution window over the transformed input data. Due to this fundamental difference between convolution and transposed convolution operations, using the same conventional convolution dataflow for generative model may lead to inefficiency. The main reason for such inefficiency can be attributed to the variable number of operations per each convolution window in the transposed convolution. The variable number of operations per each convolution window is the main result of zero insertion step in transposed convolution. Because of this zero-insertion step, distinct convolution windows may have a different number of consequential multiplications between inputs and weights.[1] This discrepancy in the number of operations is the root cause for inefficiency in the computations of generative models, if the same convolution dataflow is used. As such, we aim to design an efficient flow of data for GANs by focusing on: (1) managing the discrepancy in the number of operations per each convolution window in order to mitigate the inefficiencies in the execution of generative models, (2) leveraging the similarities between convolution and transposed convolution operations in order to accelerate both discriminative and generative models on the same hardware platform, and (3) improving the data reuse in discriminative and generative models.

Why using a conventional convolution dataflow is not efficient for transposed convolution? Going through a simple example of a 2-D transposed convolution, we illustrate the main sources of inefficiency in performing transposed convolution, if a conventional convolution dataflow is used. Figure 4(a) illustrates an example of performing a transposed convolution operation using a conventional convolution dataflow. In this transposed convolution operation, a 5×5 filter with stride of one and padding of two is applied on a 4×4 2D input. In the initial step, the transposed convolution operation inserts one row and one column of zeros between successive rows and columns (white squares). Performing this zero-insertion step, the input is expanded from a 4×4 matrix to a 11×11 one. The number of zeros to be inserted for each transposed convolution layer in the generative models may vary from one layer to another and is a parameter of the network. After performing the zero-insertion, the next step is to slide a convolution window over the transformed input and perform the multiply-add operations. Figure 4(b) illustrates performing this convolution operation using a conventional convolution dataflow [16], [20], [22]. To avoid clutter in Figure 4(b), we only show the dataflow for generating the output rows 2-5.

Each circle in Figure 4(b) represents a compute node that can perform vector-vector multiplications between a row of the filter and a row of the zero-inserted input. The filter rows are spatially reused across each of the computation nodes in a vertical manner. Once a vector-vector multiplications finish, the partial sums are aggregated horizontally to yield the results of performing transposed convolution operation for each output row. The black circles represent the compute nodes that are performing consequential operations, whereas the white circles which represent the compute nodes performing inconsequential operations. As depicted in Figure 5(b), there will be inconsequential operation (white circles) if a conventional convolution dataflow is used for the execution of transposed convolution operations. Because of the inserted zeros, some of the filter rows are not used to compute the value of an output row. For example, since the 1st, 3rd, and 5th rows of the input are zero, the 2nd output row only needs to perform the operations for non-zero elements; hence using only the 2nd and 4th filter rows, leaving three compute nodes idle. Overall, in this example, 50% of the compute nodes remain idle during the execution of this transposed convolution operation. Analyzing this transposed convolution operation reveals three main sources of inefficiency when a conventional convolution dataflow is used.

(1) **Coarse-grain resource underutilization:** Since the consequential filter rows vary from one output row to another, a significant number of compute nodes remain idle. In the aforementioned example, this underutilization applies to 50% of the compute nodes, which perform vector-vector multiplications.
(2) **Fine-grain resource underutilization:** Even within a compute node a large fraction of the multiply-add operations are inconsequential due to the columnar zero insertion.
(3) **Reuse reduction:** While the compute units pass along the filter rows for data reuse, the inserted zeros render this data transfer futile.

We address the first two sources of inefficiency with a series of optimizations on the flow of data in GANs. Then, to address the last source of inefficiency that arises because of the inconsequential multiply-add operations within each compute node, we introduce an architectural solution (Section III).

Flow of data for generative models in GANAX. Figure 5 illustrates the proposed flow of data optimizations for generative

[1] A consequential multiplication is a multiplication in which none of the source operands are zero and contributes to the final value of the convolution operation.

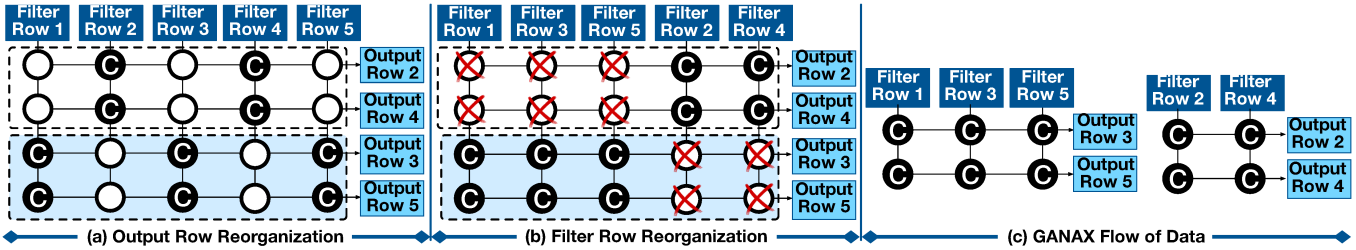


Figure 5: The GANAX flow of data after applying (a) output row reorganization and (b) filter row reorganization. (c) The GANAX flow of data after applying both output and filter row reorganization and eliminating the idle compute nodes. The combination of these flow optimizations reduces the idle (white) compute nodes and improves the resource utilization.

models in GANAX. To mitigate the challenges of using conventional convolution dataflow for transposed convolution operations in generative models, we leverage the insight that even though the patterns of computation may vary from one output row to another, they are still structured. Taking a closer look at Figure 4, we learn that there are only two distinct patterns[2] in the output row computations. In this example, the even output rows (i.e., 2nd and 4th) use one pattern of computation, whereas the odd output rows (i.e., 3rd and 5th) use a different pattern for their computations. Building upon this observation, we introduce a series of flow of data optimizations to mitigate the aforementioned inefficiencies in the computation of transposed convolution operation, if a conventional convolution dataflow used.

The first optimization maximizes the data reuse by reorganizing the computation of the output rows in a way that the rows with the same pattern in their computations become adjacent. Figure 5(a) illustrates the flow of data after applying this output row reorganization. Applying the output row reorganization in this example, make the even-indexed (2nd and 4th output rows) output rows adjacent. Similar adjacency is established for odd-indexed (3rd and 5th output rows) output rows. Although this optimization addresses the data reuse problem, it does not deal with the resource underutilization (i.e., idle compute nodes (white circles) still exist). To mitigate this resource underutilization, we introduce the second optimization that reorganizes the filter rows. As shown in Figure 5(b), applying the filter row reorganization establishes an adjacency for the 1st, 3rd, and 5th filter rows. Similarly, the 2nd and 4th filter rows become adjacent. After applying output and filter row reorganization, as shown in Figure 5(b), the idle compute nodes can be simply eliminated from the dataflow. Figure 5(c) illustrates the GANAX flow of data after performing both optimizations, which improves the resource utilization for transposed convolution operation from 50% to 100%.

The proposed GANAX flow of data also addresses the inefficiency in performing the horizontal accumulation of partial sums. As shown in Figure 4(b), the conventional convolution dataflow requires five cycles to perform the horizontal accumulation for each output row, regardless of their locations. However, comparing Figure 4(b) and Figure 5(c), we observe that after applying output and filter row reorganization optimizations, the number of required cycles for performing the horizontal accumulation reduces from five to two for even-indexed output rows and from five to three

[2]The location of white and black circles (compute nodes) defines each pattern.

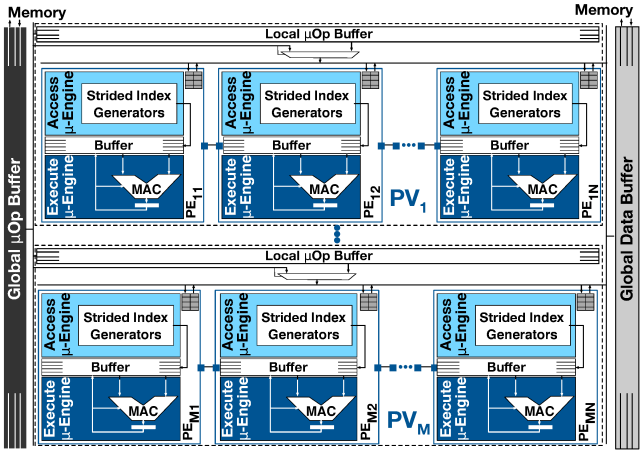


Figure 6: Top-level block diagram of GANAX architecture.

for odd-indexed output rows. While the proposed flow of data optimizations effectively improve the resource utilization for transposed convolution, there arises an interesting architectural challenge: *how to fully utilize the parallelism between the computations of the output rows that require different number of cycles for horizontal accumulation (two cycles for even-indexed and three cycles for odd-indexed output rows)?* If a SIMD execution model is used, some of the compute nodes have to remain idle until the accumulations for the output rows that require more cycles for horizontal accumulation, finish. The next section elaborates on the GANAX architecture that exploits the introduced flow of data for transposed convolution and fully utilize the parallelism between distinct output rows by conjoining the MIMD and SIMD execution models.

III. Architecture Design for GANAX

The execution flow of the generative model (*i.e.* zero-insertion and variable number of operations per each convolution window) in GANs poses unique architectural challenges that the traditional convolution accelerators [16]–[18], [20], [22] can not adequately address. There are two fundamental architectural challenges for GAN acceleration as follows:

Resource underutilization. The first challenge arises due to the variable number of operations per each convolution window in transposed convolution operation. In most of recent accelerators [17], [18], [20], [22], which mainly target conventional convolution operation, the processing engines generally work in a SIMD manner. The convolution windows in conventional convolution operation follow a regular pattern and the number of operations for

each of these windows remains invariable. Due to these algorithmic characteristics of conventional convolution operation, a SIMD execution model is an efficient and practical model. However, since the convolution windows in transposed convolution operations exhibit a variable number of operations, a SIMD execution model is not an adequate design choice for these operations. While using a SIMD model utilizes the data parallelism between the convolution windows with the same number of operations, its efficiency is limited in exploiting this execution model for the windows with a different number of operations. That is, if one uses a convolution accelerator with a SIMD execution model for transposed convolution operations, the processing engines that are performing the operations for a convolution window with fewer number of operations have to remain idle until the operations for other convolution windows finish. To address this challenge, we introduce a unified MIMD-SIMD architecture to accelerate the transposed convolution operation without compromising the efficiency of conventional convolution accelerators for convolution operations. This unified MIMD-SIMD architecture effectively maximizes the utilization of accelerator compute resources while effectively utilizing the parallelism between the convolution windows with different number of operations.

Inconsequential computations. The second challenge emanates from the large number of zeros inserted in the multidimensional input feature map for transposed convolution operations. Performing MAC operations on these zeros is inconsequential and wastes accelerator resources (See Figure 1), if not skipped. We address this challenge by leveraging an observation that even though the data access patterns in transposed convolution operations are irregular, they are still structured. Furthermore, these structured patterns are repetitive across the execution of transposed convolutional operations. Building upon these observations, the GANAX architecture decouples the operand access and execution. Each processing engine in this architecture consists of a simple access engine that repetitively generates the addresses for operand accesses without interrupting the execute engine. In the next sections, we examine these architectural challenges in details for GAN acceleration and expound the proposed microarchitectural solutions.

A. *Unified MIMD-SIMD Architecture*

In order to mitigate the resource underutilization, we devise a unified SIMD-MIMD architecture that reaps the benefits of SIMD and MIMD execution models at the same time. That is, while our architecture executes the operations for convolution windows with distinct computation patterns in a MIMD manner, it performs the operations of the convolution windows with the same computation pattern in a SIMD manner. Figure 6 illustrates the high-level diagram of the GANAX architecture, which is comprised of a set of identical processing engines (PE). The PEs are organized in a 2D array and connected through a dedicated network. Each PE consists of two μ-engines, namely the access μ-engine and the execute μ-engine. The access μ-engine generates the addresses for source and destination operands, whereas execute μ-engine merely performs simple operations such as multiplication, addition, and multiply-add. The memory hierarchy is composed of an off-chip memory and two separate on-chip global buffers, one for data and one for μops. These global on-chip buffers are shared across all the PEs. Each PE operates on one row of filter and one row of input and generates one row of partial sum values. The partial sum values are further accumulated horizontally across the PEs to generate the final output value. Using a SIMD model for transposed convolution operations leads to resource underutilization. The PEs that perform the computation for convolution windows with fewer number of operations remains idle, wasting computational resources. The simple solution is to replace the SIMD model with a fully MIMD computing model and utilize the parallelism between the convolution windows with different number of operations. However, a MIMD execution model requires augmenting each processing engine with a dedicated operation buffer. While this design resolves the underutilization of resources, it imposes a large area overhead, increasing area consumption by $\approx 3\times$. Furthermore, fetching and decoding instructions from each of these dedicated operation buffers significantly increases the von Neumann overhead of instruction fetch and decode. To address these challenges, we design the GANAX architecture upon this observation that PEs in the same row perform same operations for a large period of time. As such, the proposed architecture leverages this observation and develop a middle ground between a fully SIMD and a fully MIMD execution model. The goal of designing the GANAX architecture is multi-faceted: (1) improve the PE underutilization by combining MIMD/SIMD model of computation for transposed convolution operations (2) without compromising the efficiency of SIMD model for conventional convolution operations. Next, we explain the two novel microarchitectural components that enable an efficient MIMD-SIMD accelerator design for GAN acceleration.

Hierarchical μop buffers. To enable a unified MIMD and SIMD model of execution, we introduce a two-level μop buffer. Figure 6 illustrates the high-level structure of the two-level μop buffer. The two-level μop buffer consists of a global and a local μop buffer. The local and global μop buffers work cooperatively to perform the computations for GANs. Each horizontal group of PEs, called processing vector (PV), shares a local μop buffer, whereas, the global μop buffer that is shared across all the PVs. The GANAX accelerator can operate in two distinct modes: SIMD mode and MIMD-SIMD mode. Since all the convolution windows in the convolution operation have the same number of multiply-adds, the SIMD execution model is a best fit. As such for this case, the global μop buffer bypasses the local μops and broadcasts the fetched μop to all the PEs. On the other hand, since the number of operations varies from one convolution window to another in transposed convolution operation, the accelerator works in MIMD-SIMD mode. In this mode, the global μop buffer sends distinct indices to each local μop buffer. Upon receiving the index, each local μop buffer broadcasts a μop, at the location pointed by the received index, to all the underlying PEs. Using MIMD-SIMD mode enables the GANAX accelerator to not only utilize the parallelism between the convolution windows with the same number of operations, but also utilize the parallelism across the windows with distinct number of operations.

Global μop buffer. Before starting the computations of a layer, a sequence of high-level instructions, which defines the structure of each GAN layer, are statically translated into a series of μops. These μops are pre-loaded into the global μop buffer, and then the execution starts. Each of the μops either performs an operation across all the PEs (SIMD) or initiates an μop in each PV (MIMD-SIMD). The initiated operation in the MIMD-SIMD mode may vary from one PV to another. The SIMD and MIMD μops can be stored in the global μop buffer in any order. A 1-bit field in the global μop identifies the type of μop: SIMD or MIMD-SIMD. In the SIMD mode—all the PEs share the same μop globally but execute it on distinct data—the global μop defines the intended operation to be performed by all the PEs. In this mode, the local μop buffer is bypassed and the global μop are broadcasted to all the PEs at the same time. Upon receiving the μop, all the PEs perform the same operation, but on distinct data. In the MIMD-SIMD mode—all the PEs within the same PV share the same μop but different PVs may execute different μops—the global μop is partitioned into multiple fields (one filed per each PV), each of which defines an index for accessing an entry in the local μop buffer. Upon receiving the index, each local μop buffer retrieves the corresponding μop stored at the given index and broadcasts it to all the PEs which it controls. The global μop buffer is double-buffered so that the next set of μops for performing the computations of GAN layer$_{i+1}$ can be loaded into the buffer while the μops for GAN layer$_i$ are being executed.

Local μop buffer. In the GANAX architecture, each PV has a dedicated local μop buffer. In the SIMD mode, the local μop buffers are completely bypassed and all the PEs perform the same operation that are sent from global μop buffer. In the MIMD-SIMD mode, each local μop buffer is accessed at the location specified by a dedicated field in the global μop. This location may vary from one local μop buffer to another. Then, the fetched μop is broadcasted to all the PEs within a PV to perform the same operation but on distinct data. Each GAN layer may require a distinct sequence of μops both globally and locally. Furthermore, each PE may need to access millions of operands at different locations to perform the computations of a GAN layer. Therefore, we may need not to only add large μop buffers to each PE, but also drain and refill the μop buffers multiple times. Adding large buffers to the PEs adds a large area overhead, which could have been utilized to improve the computing power of the accelerator. Also, the process of draining and refilling the μop buffers imposes a significant overhead in terms of both performance and energy. To mitigate these overheads, we introduce decoupled access-execute microarchitecture that enables us to significantly reduce the size of μop buffers and eliminate the need to drain and refill the local μop buffers for each GAN layer.

B. Decoupled Access-Execute μEngines

Though the data access patterns in transposed convolution operation are irregular they are still structured. Furthermore, the data access patterns are repetitive across the convolution windows. Building upon this observation, we devise a microarchitecture that decouples the data accesses from from the data processing.

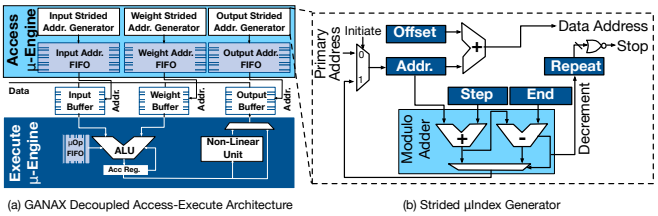


Figure 7: Organization of decoupled Access-Execute architecture.

Figure 7 illustrates the organization of the proposed decoupled access-execute architecture. The GANAX decoupled access-execute architecture consists of two major microarchitectural units, one for address generation (access μ-engine) and one for performing the operations (execute μ-engine).

The access μ-engine generates the addresses for the input, weight, and output buffers. The input, weight, and output buffers consume the generated addresses for each data read/write. The execute μ-engine, on the other hand, receives the data from the input and weight buffers, performs an operation, and stores the result in the output buffer. The μops of these two engines are entirely segregated. However, the access and execute μ-engines work cooperatively to perform an operation. The μops for access μ-engine handle the configuration of index generator units. The μops for execute μengine *only* specify the type of operation to be performed on data. As such, the execute μops do *not* need to include any fields for specifying the source/destination operands. Every cycle, the access μengine sends out the addresses for source and destination operands based on its preconfigured parameters. Then, the execute μengine performs an operation on the source operands. The result of the operation is, then, stored in the location that is defined by the access μengine. Having decoupled μ-engines for accessing the data and executing the operations has a paramount benefit of reusing execute μops. Since there is no address field in the execute μops, we can reuse the same execute μop on distinct data over and over again without the need to change any fields in the μops. Reusing the same μop on distinct data helps to significantly reduce the size of μop buffers.

Access μ-engine. Figure 7 illustrates the microarchitectural units of access μ-engine. The main function of access μ-engine is to generate the addresses for source and destination operands based on a preloaded configuration. While designing a full-fledged access μ-engine that is capable of generating various patterns of data addresses enables flexibility for the GANAX accelerator, but it is an overkill for our target application (i.e., GANs). As mentioned in the dataflow section (Section II), the data access patterns for transposed convolution operations are irregular, yet structured. Based on our analysis over the evaluated GANs, we observe that the data accesses in the GANAX dataflow are either *strided* or *sequential*. The stride value for a strided data access pattern depends on the number of inserted zeros in the multidimensional input activation. Furthermore, these data access patterns are repetitive across a large number of convolution windows and for large number of cycles. We leverage these observations to simplify the design of the access μ-engine. Figure 7(a) depicts the block diagram of the access μengine in GANAX. The access engine mainly consists of one or more

strided μindex generators. The μindex generator can generate one address every cycle, following a pattern governed by a preloaded configuration. Since the data access patterns may vary from one layer to another, we design a reconfigurable μindex generator.

Figure 7(b) depicts the block diagram of the proposed reconfigurable μindex generator. There are five configuration registers that govern the pattern for data address generation.

The Addr. configuration register specifies the initial address from which the data address generation starts, while the Offset configuration register can be used to offset the range of generated addresses as needed. The Step configuration register specifies the step size between two consecutive addresses, while the End configuration register specifies the final value up to which the addresses should be generated. Finally, the Repeat configuration register indicates the number of times that a configured data access pattern should be replayed. The modulo adder, which consists of an adder and a subtractor, is used to enable data address generation in a rotating manner. The modulo adder performs a modulo addition on the values stored in the Addr. and Step registers. If the result of this modulo addition is fewer than the value in End register, the calculated result is sent to the output. This means that the next address to be generated is still within the range of Addr. and End register values. Otherwise, the result of the modulo addition minus the value of End register is sent to the output. That is, the next address to be generated is beyond the End register value and the address generation process must start over from the beginning. In this scenario, the Decrement signal is also asserted which cause the value of the Repeat register to be decreased by one, indicated one round of address generation is finished. Once the Repeat register reaches zero, the Stop signal is asserted and no more addresses are generated. After configuring the parameters, the strided μindex generator can yield one address per cycle without any further interventions from the controller. Using this configurable μindex generator along the observation that the data address patterns in GANs are structured, the GANAX architecture can bypass the inconsequential computations and save both cycles and energy.

Execute μ-engine. Figure 7(b) depicts the microarchitectural units of execute μ-engine. The execute μ-engine consists of an ALU, which can perform simple operations such as addition, multiplication, comparison, and multiply-add. The main job of execute μ-engine is *just* to perform an operation on the received data. At each cycle the execute μ-engine consumes one μop from the μop FIFO and performs the operation on the source operands and store the result back into the destination operand. If the μOp FIFO becomes empty, the execute μop halts and no further operation is performed. In this case, all the input/weight/output buffers are notified to stop their reads/writes. The decoupling between access and execute μengines enables us to remove the address field from the execute μops. Removing the address field from the execute μops allow us to reuse the same μops over and over again on different data. Furthermore, we leverage this μop reuse and the fact that the computation of the CNN requires a small set of μops ($\approx$ 16) to simplify the design of the μop buffers. Instead of draining and refilling the μop buffers, we preload all the necessary μops for convolution and transposed convolution operations in the μop buffers. For the local μop buffer, we load *all* the μops before starting the computation of a GAN.

Synchronization between μengines. In the GANAX architecture (Figure 7), there is one address FIFO for each strided μindex generator. The address FIFOs perform the synchronization between access μ-engine and execute μ-engine. Once an address is generated by a strided μindex generator, the generated address is pushed into the corresponding address FIFO. The addresses in the address FIFOs are later consumed to read/write data from/into the data buffers (i.e., input/weight/output buffers). If any of the address FIFOs are full, the corresponding strided μindex generator stops generating new addresses. In the case that any of the address FIFOs are empty, no data is read/written from/into its corresponding address FIFO.

IV. Instruction Set Architecture Design (μOps)

The GANAX ISA should provide a set of μops to efficiently map the proposed flow of data for both generative and discriminative models onto the accelerator. Furthermore, these μops should be sufficiently *flexible* to serve distinct patterns in the computation for both convolution and transposed convolution operations. Finally, to keep the size of μop buffers modest, the set of μops should be *succinct*. To achieve these multifaceted goals, we first introduce a set of algorithmic observations that are associated with GAN models. Then, we introduce the major μops that enable the execution of GAN models on GANAX.

A. Algorithmic Observations

The following elaborates a set of algorithmic observations that are the foundation of the GANAX μops.

(1) MIMD/SIMD execution model. Due to the regular and structured patterns in the computation across the convolution windows in conventional DNNs, they are best suited for SIMD processing. However, the patterns in the computation of GANs are inherently different between generative and discriminative models. Due to the inserted zeros in the generative models, their patterns in the computation vary from one convolutional window to another. We observe that exploiting a combination of SIMD and MIMD execution model can be more efficient in accelerating GAN models than solely relying on SIMD. Therefore, the focus of the GANAX μops is to include the operations that enable GANAX to fully utilize the SIMD and MIMD execution models.

(2) Repetitive computation patterns. We observe that even though GANs require a large number of computations, most of these computations are similar between generative and discriminative models. In addition, these computations are repetitive over a long period of time. Building upon this observation, we introduce a customized `repeat` μop that significant reduces the μop footprints. In addition, the commonality between the operations in generative and discriminative models allows us to design a succinct, yet representative, set of μops. To further reduce the μop footprints, we introduce a dedicated set of execute μops that only define the type of operations. These μops are reused for distinct data during the execution of generative and discriminative models on the GANAX architecture.

(3) Structured and repetitive memory access patterns. We observe that despite the irregularity of memory access patterns in generative models, they are still structured and repetitive. Analyzing the data access patterns of various GANs reveals that their memory access patterns are either sequential or strided. Building upon this observation and our decoupled access-execute architecture, we introduce a set of access μops that are used merely to configure the access μengines and initiate the address generation process. Once initiated, the access μengines generate the configured access patterns over and over until they are intervened.

B. Access μOps

GANAX access μops are used to configure the access μengine and initiate/stop the process of address generation. These μops are executed across all the PEs within a PV whose index is indicated by pv_index field in the μops. Furthermore, in all of these μops, %addrgen_idx specifies the index of the targeted address generator in the access μengine. The supported μops in the access μengines are as follows:

1) **`access.cfg`** %pv_idx, %addrgen_idx, %dst, imm: This μop loads a 16-bit imm value into one of the five %dst configuration registers (*i.e.,* as shown in Figure 7(b), these configuration registers are Addr., Offset, Step, End, and Repeat) of one of the address generators in the access μengine.
2) **`access.start`** %pv_idx, %addrgen_idx: This μop initiates the address generation in one of the address generators in the access μengine. The process of address generation continues until an **`acceess.stop`** μop is executed or the iteration register reaches zero.
3) **`access.stop`** %pv_idx, %addrgen_idx: This μop intervenes the address generation of one of the address generators in the access μengine. The address generation can be re-initiated again by executing an access.start μop.

C. Execute μOps

Execute μops are categorized into two groups: (1) SIMD μops are fetched from each PE's local μop buffer and executed locally within each PE and (2) the MIMD μops are fetched from the global μop buffer and executed across all PEs. The SIMD μops can be executed in the MIMD manner as well. That is, the MIMD μops are a superset of the SIMD μops. We first introduce the SIMD μops, then explain the extra μops that belong to the MIMD group.

SIMD μops. SIMD group only comprises a succinct, yet representative set of μops for performing convolution and transposed convolution operations. The combination of SIMD μops and the decoupled access-execute architecture in GANAX helps to reduce the size of local μop buffers. The SIMD μops do not have source or destination fields and only specify the type of operation to be executed. Once executed, depending on the type of operation, a given PE consumes the generated addresses by the μindex generators and delivers the data to the execute μengine. Since these μops do not have any source or destination register, they are pre-loaded into the local μop buffers before starting the execution. Then, they are re-used over and over, on distinct data whose addresses are generated by the access μengines. The SIMD μops are as follows:

Table I: The evaluated GAN models, their released year, and the number of convolution (Conv) and transposed convolution (TConv) layers per generative and discriminative models.

			Generative		Discriminative	
Name	**Year**	**Description**	**# Conv**	**# TConv**	**# Conv**	**# TConv**
3D-GAN [11]	2016	3D objects generation	-	4	5	-
ArtGAN [6]	2017	Complex artworks generation	-	5	6	-
DCGAN [12]	2015	Unsupervised representation learning	-	4	5	-
DiscoGAN [8]	2017	Style transfer from one domain to another	5	4	5	-
GP-GAN [7]	2017	High-resolution image generation	-	4	5	-
MAGAN [51]	2017	Stable training procedure for GANs	-	6	6	6

1) **`add`**, **`mul`**, **`mac`**, **`pool`**, and **`act`**: Depending on the type, these μops consume one or more addresses from the μindex generators for source and destination operands. For example, **`add`** consumes two addresses for the source operands and one address for the destination operand, but **`act`** uses one address for the source operand and one address for the destination operand.
2) **`repeat`**: This μop causes the next fetched μop to be repeated a specified number of times. This number is specified in a microarchitectural register in each PE. This register is pre-loaded with a MIMD μop before the execution starts.

MIMD μops. The MIMD μops are loaded into the global μop buffers and executed globally across all the PEs. In addition to all the SIMD μops, the following μops execute in a MIMD manner:

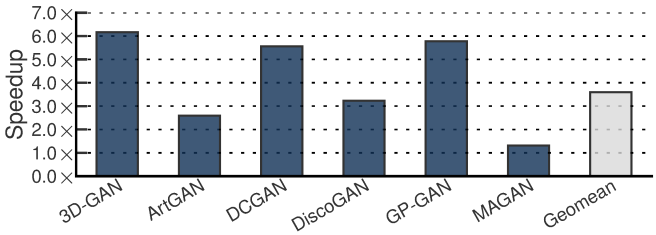
1) **`mimd.ld`** %pv_idx, %dst, imm: This μop loads the immediate value (imm) into one of the microarchitectural registers (%dst) of all the PEs with a PV. The %pv_idx, specifies the index of the target PV. This μop is mainly used to load an immediate value into the repeat register.
2) **`mimd.exe`** %μop_index$_1$,..., %μop_index$_i$: Upon receiving this μop, the ith PV fetches a μop located at location %μop_index$_i$ from its local μop buffer and executes it across all the PEs horizontally. Since the value of the %μop_index may vary from one PV to another, this μop causes GANAX to operate in a MIMD manner.

V. Methodology

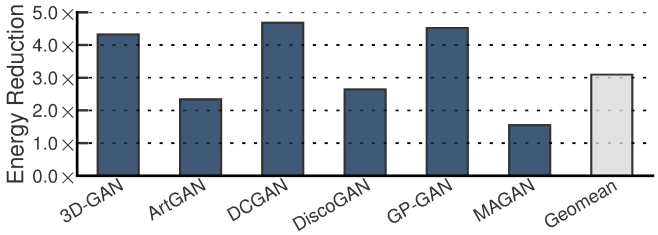
Workloads. We use several state-of-the-art GANs to evaluate the GANAX architecture. Table I, shows the evaluated GANs, a brief description of their applications, and the number of convolution (Conv) and transposed convolution (TConv) layers per generative and discriminative models.

Hardware design and synthesis. We implement the GANAX microarchitectural units including the strided μindex generator, the arithmetic logic of the PEs, controllers, non-linear function, and other logic hardware units in Verilog. We use TSMC 45 nm standard-cell library and Synopsys Design Compiler (L-2016.03-SP5) to synthesize these units and obtain the area, delay, and energy numbers.

Energy measurements. Table II shows the energy numbers for major micro-architectural units, memory operations, and buffer accesses in TSMC 45nm technology. To measure the area and read-/write access energy of the register files, SRAMs, and local/global buffers, we use CACTI-P [51]. To have a fair comparison, we use energy numbers reported in Tetris [22], which has a similar PE architecture as Eyeriss. In Table II, the energy overhead of strided μindex generators is included in the normalized energy cost of PE. For DRAM accesses, we use the Micron's DDR4



(a) Speedup



(b) Energy Reduction

Figure 8: Speedup and energy reduction of generative models compared to EYERISS [16].

system power calculator [52]. The same frequency (500 MHz) is used for both EYERISS and GANAX in all the experiments.

Architecture configurations. In this paper, we study a configuration of GANAX with 16 Processing Vectors (PVs) each with 16 Processing Engines (PEs). We use the default EYERISS configurations for on-chip memories such as the size of input and partial sum registers, weight SRAM, and global data buffer. The same on-chip memory sizes are used for GANAX. Each local μop buffer has 16 entries. The number of entries is sufficient to encompass all the execute μops. The global μop buffer has 32 entries each with 64 bits, four bits per each PV. Each local μop uses these four bits to index its local μop buffer. An extra one bit in the global μops determines the execution model of the accelerator for the current operation (i.e., SIMD or MIMD-SIMD).

Area analysis. Table III shows the major architectural components for the baseline architecture (EYERISS [16], [20]) and GANAX in 45 nm technology node. For logic of the microarchitectural units, we use the reported area from the synthesis. For the memory elements, we use CACTI-P [51] and the reported numbers in EYERISS [20]. In order to be consistent in the results, we scaled down the reported area numbers in EYERISS from 65 nm to 45 nm. To have a fair comparison between EYERISS and GANAX, the same number of PEs and on-chip memory are used for both accelerators. Under this setting, GANAX has an area overhead of ≈7.8% compared to EYERISS.

Microarchitectural simulation. Table III shows the major microarchictural parameters of GANAX. We implement a microarchitectural simulator on top of the EYERISS simulator [22]. The extracted energy numbers from logic synthesis and CACTI-P are integrated into the simulator to measure the energy consumption of the evaluated network models on GANAX. To evaluate our proposed accelerator, we extend the EYERISS simulator with the proposed ISA extensions and the GANAX flow of data. For all the baseline numbers, we use the plain version of the simulator.

Table II: Energy comparison between GANAX microarchitectural units and memory. PE energy includes the energy consumption of an arithmetic operation and the strided μindex generators.

Operation	Energy (pJ/Bit)	Relative Cost
Register File Access	0.20	1.0
16-bit Fixed Point PE	0.36	1.8
Inter-PE Communication	0.40	2.0
Global Buffer Access	1.20	6.0
DDR4 Memory Access	15.00	75.0

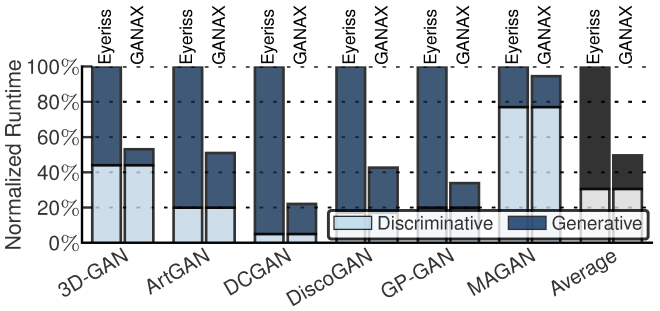
Table III: Area measurement of the major hardware units with TSMC 45nm.

			Area	
	GANAX Hardware Units	Configuration	um^2	%
Processing Engine (PE)	Input Register	12 × 16 Bits	766.9	2.6%
	Partial Sum Register	24 × 16 Bits	1533.7	5.2%
	Weight SRAM	224 × 16 Bits	14378.7	48.8%
	Multiply-and-Accumulate	16-bit Fixed Point	2875.7	9.8%
	Non-Linear Function	Lookup Table	95.9	0.3%
	Strided µIndex Generator	3	479.3	1.6%
	Local µOP Buffer	16 × 16 Bits	958.6	3.3%
	I/O FIFOs	8 × 32 Bits	5026.8	17.1%
	PE Controller	N/A	3356.0	11.4%
Total Area / PE			29471.6	100.0%
GANAX	Total PE Array	16 × 16	7544466.2	83.2%
	Global µOP Buffer	32 × 64 Bits	9585.8	0.1%
	Global Data Buffer	108 KBytes	1102366.9	12.2%
	Global Instruction Buffer	27 KBytes	275591.7	3.0%
	Others (NoC, Config Buffers)	N/A	115029.6	1.3%
	Global Controller	N/A	19171.6	0.2%
GANAX Total Area			9066211.8	100.0%

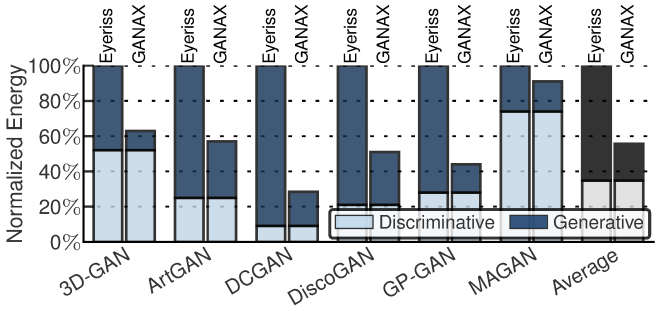
VI. EVALUATION

Overall performance and energy consumption comparison. Figure 8a depicts the speedup of the generative models with GANAX over EYERISS [16]. On average, GANAX yields 3.6× speedup improvement over EYERISS. The generative models with a larger fraction of inserted zeros in the input data and larger number of inconsequential operations in transposed convolution layers enjoy a higher speedup with GANAX. Across all the evaluated models, 3D-GAN achieves the highest speedup (6.1×). This higher speedup is mainly attributed to its larger number of inserted zeros in its transposed convolution layers. On average, the number of inserted zeros for 3D-GAN is around 80% (See Figure 1). On the other extreme, MAGAN enjoys a speedup of merely 1.3×, which is attributed to the lowest number of inserted zeros in its transposed convolution layers compared to other GANs.

Figure 8b shows the energy reduction achieved by GANAX over EYERISS. On average, GANAX effectively reduces the energy consumption by 3.1× over the EYERISS accelerator. The GANs (3D-GAN, DCGAN, and GP-GAN) with the highest fraction of zeros and inconsequential operations in the transposed convolution layers enjoy an energy reduction of more than 4.0×. These results reveal that our proposed architecture is efficient in addressing the main sources of inefficiency in the generative models. Figure 9 shows the normalized runtime and energy breakdown between the discriminative and generative models. The first (second) bar shows the normalized runtime (energy) for EYERISS (GANAX). To be consistent across all the networks, for the discriminative model of MAGAN, we *only* consider the contribution of convolution layers in the overall runtime and energy consumption. As the results show, while GANAX significantly reduces both the runtime and energy



(a) Runtime



(b) Energy

Figure 9: Breakdown of (a) runtime and (b) energy consumption between discriminative and generative models normalized to the runtime and energy consumption of EYERISS. For each network, the first (second) bar show the normalized value when the application is executed on EYERISS (GANAX).

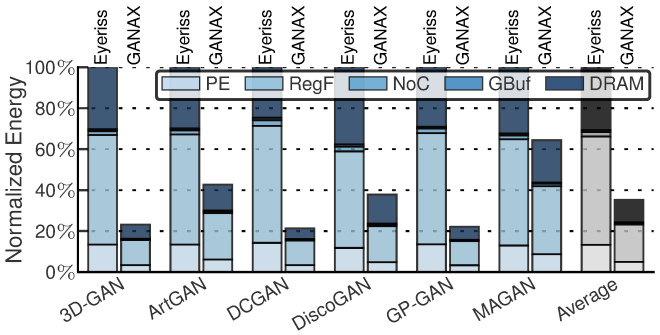


Figure 10: Breakdown of energy consumption of the generative models between different microarchitectural units. The first bar shows the normalized energy breakdown for EYERISS. The second bar show the energy breakdown for GANAX normalized to EYERISS.

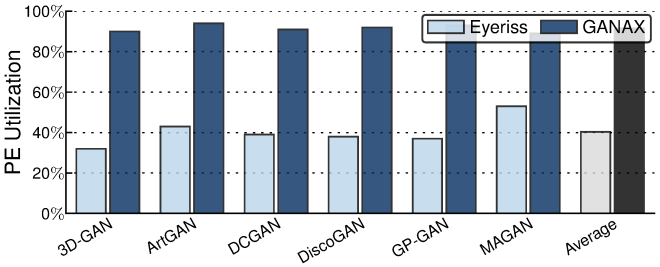


Figure 11: Average PE utilization for the generative models in EYERISS and GANAX.

consumption of generative models, it delivers the same level of efficiency as EYERISS for the discriminative models.

Energy breakdown of the microarchitectural units. Figure 10 illustrates the overall normalized energy breakdown of the generative models between distinct microarchitectural components of the GANAX architecture. The first and second bars show the normalized energy consumed by EYERISS and GANAX, respectively. As the results show, GANAX reduces the energy consumption of all the microarchitectural units. This reduction is mainly attributed to the efficient flow of data in GANAX and the decoupled access-execute architecture that cooperatively diminishes the sources of inefficiency in the execution of transposed convolution operations.

Processing elements utilization. To show the effectiveness of GANAX dataflow in improving the resource utilization, we measure what percentage of the total runtime, the PEs are actively performing a consequential operation. Figure 11 depicts the utilization of PEs for EYERISS and GANAX. GANAX exhibits a high percentage of PE utilization, around 90% across all the evaluated GANs. This high resource utilizations in GANAX is mainly attributed to the proposed dataflow that can effectively force the computation of the rows with similar computation pattern adjacent to each other. This forced adjacency of similar computation patterns eliminates inconsequential operations, which leads to a significant improvement in the utilization of the processing engines.

VII. RELATED WORK

GANAX has fundamentally a different accelerator architecture than the prior proposals for deep network acceleration. In contrast to prior work that mostly focus on convolution operation, GANAX accelerates transposed convolution operation, a fundamentally different operation than conventional convolution. Below, we overview the most relevant work to ours along two dimensions: neural network acceleration and MIMD-SIMD acceleration.

Neural network acceleration. Accelerator design for neural networks has become a major line of computer architecture research in recent years. A handful of prior work explored the design space of neural network acceleration, which can be categorized into ASICs [15], [16], [18]–[22], [26], [27], [30], [34], [37], [38], [41], [42], FPGA implementations [17], [28], [35], [36], [43], using unconventional devices for acceleration [29], [33], [40], and dataflow optimizations [16], [23]–[25], [31], [32], [39]. Most of these studies have focused on accelerator design and optimization of merely one specific type of convolutional as the most compute-intensive operation in deep convolutional neural networks.

EYERISS [16] proposes a row stationary dataflow that yields high energy efficiency for convolutional operation. EYERISS exploits data gating to skip zero inputs and further improves the energy efficiency of the accelerator. However, EYERISS still wastes cycles for detecting the zero-valued inputs. Cnvlutin [30] can save compute cycle and energy for zero-values inputs but still wastes resources for zero-valued weights. In contrast, Cambricon-X [26] can skip zero-valued weights but still wastes compute cycles and energy for zero-input values. SCNN [21] proposes an accelerator that can skip both zero-valued inputs and weights and efficiently performs convolution on highly sparse data. However, not only SCNN cannot handle dynamic zero-insertion in input feature maps, but also it is not efficient for non-sparse vector-vector

multiplications, which are the dominant operation in discriminative models of GANs. None of these works can perform zero-insertion into the input feature maps, which is fundamentally a requisite for transposed convolution operation in the generative models. Compared to these successful prior work in neural network acceleration, GANAX proposes a unified architecture for efficient acceleration of both conventional convolution and transposed convolution operations. As such, GANAX encompasses the acceleration of a wider range of neural network models.

MIMD-SIMD accelerators. While the idea of access-execute is not brand-new, GANAX extends the concept of access-execute architecture [44]–[47] to the finest granularity of computation for each individual operand for deep network acceleration. A wealth of research has studied the benefits of MIMD-SIMD architecture in accelerating specific applications [53]–[61]. Most of these works have focuses on accelerating computer vision applications. For example, PRECISION [54] proposes a reconfigurable hybrid MIMD-SIMD architecture for embedded computer vision. In the same line of research, a recent work [61] proposes a multicore architecture for real-time processing of augmented reality applications. The proposed architecture leverages SIMD and MIMD for data- and task-level parallelism, respectively. While these works have studied the benefits of MIMD-SIMD acceleration mostly for computer vision applications, they did not study the potential gains of using MIMD and SIMD accelerators for modern machine learning applications. Prior to this work, the benefits, limits, and challenges of MIMD-SIMD architectures for modern deep model acceleration was unexplored. Conclusively, the GANAX architecture is the first to explore this uncharted territory of MIMD-SIMD acceleration for the next generation of deep networks.

VIII. Conclusion

Generative adversarial networks harness both generative and discriminative deep models in a game theoretical framework to generate close-to-real synthetic data. The generative model uses a fundamentally different mathematical operator, called transposed convolution, as opposed to the conventional convolution operator. Transposed convolution extrapolates information by first inserting zeros and then applying convolution that needs to cope with irregular placement of none-zero data. To address the associated challenges for executing generative models without sacrificing accelerator performance for conventional DNNs, this paper devised the GANAX accelerator. In the proposed accelerator, we introduced a unified architecture that conjoins SIMD and MIMD execution models to maximize the efficiency of the accelerator for both generative and discriminative models. On the one hand, to conform to the irregularities in the generative models, which are formed due to the zero-insertion step, GANAX supports selective execution of only the required computations by switching to a MIMD-SIMD mode. To support this mixed execution mode, GANAX offers a decoupled micro access-execute paradigm at the finest granularity of its processing engines. On the other hand, for the conventional discriminator DNNs, it sets the architecture in a purely SIMD mode. The evaluation results across a variety of generative adversarial networks reveal that the GANAX accelerator delivers, on average, 3.6× speedup and 3.1× energy reduction for the generative models. These significant benefits are attained without sacrificing the execution efficiency of the conventional discriminator DNNs.

IX. Acknowledgments

We thank Hardik Sharma, Ecclesia Morain, Michael Brzozowski, Hajar Falahati, and Philip J. Wolfe for insightful discussions and comments that greatly improved the manuscript. Amir Yazdanbakhsh is partly 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, NSF-1705047, Samsung Electronics, and gifts from Google, Microsoft, Xilinx, and Qualcomm.

References

[1] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative Adversarial Nets," in *NIPS*, 2014.
[2] D. Nie, R. Trullo, J. Lian, C. Petitjean, S. Ruan, Q. Wang, and D. Shen, "Medical Image Synthesis with Context-aware Generative Adversarial Networks," in *MICCAI*, 2017.
[3] P. Costa, A. Galdran, M. I. Meyer, M. Niemeijer, M. Abràmoff, A. M. Mendonça, and A. Campilho, "End-to-end Adversarial Retinal Image Synthesis," *T-MI*, 2017.
[4] J. Ho and S. Ermon, "Generative Adversarial Imitation Learning," in *NIPS*, 2016.
[5] A. Ghosh, B. Bhattacharya, and S. B. R. Chowdhury, "SAD-GAN: Synthetic Autonomous Driving using Generative Adversarial Networks," *arXiv*, 2016.
[6] W. R. Tan, C. S. Chan, H. Aguirre, and K. Tanaka, "ArtGAN: Artwork Synthesis with Conditional Categorial GANs," *arXiv*, 2017.
[7] H. Wu, S. Zheng, J. Zhang, and K. Huang, "GP-GAN: Towards Realistic High-Resolution Image Blending," *arXiv*, 2017.
[8] T. Kim, M. Cha, H. Kim, J. K. Lee, and J. Kim, "Learning to Discover Cross-Domain Relations with Generative Adversarial Networks," *ArXiv*, 2017.
[9] L.-C. Y. Y.-H. Y. Hao-Wen Dong, Wen-Yi Hsiao, "MuseGAN: Symbolic-domain Music Generation and Accompaniment with Multi-track Sequential Generative Adversarial Networks," *arXiv*, 2017.
[10] L.-C. Yang, S.-Y. Chou, and Y.-H. Yang, "MidiNet: A Convolutional Generative Adversarial Network for Symbolic-domain Music Generation using 1D and 2D Conditions," *arXiv*, 2017.
[11] J. Wu, C. Zhang, T. Xue, W. T. Freeman, and J. B. Tenenbaum, "Learning a Probabilistic Latent Space of Object Shapes via 3D Generative-Adversarial Modeling," in *NIPS*, 2016.
[12] A. Radford, L. Metz, and S. Chintala, "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks," *arXiv*, 2015.
[13] Microsoft, "Microsoft unveils Project Brainwave for real-time AI." https://www.microsoft.com/en-us/research/blog/microsoft-unveils-project-brainwave/, 2017.
[14] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, *et al.*, "In-datacenter Performance Analysis of a Tensor Processing Unit," in *ISCA*, 2017.
[15] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: Efficient Inference Engine on Compressed Deep Neural Network," in *ISCA*, 2016.
[16] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks," in *ISCA*, 2016.
[17] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From High-Level Deep Neural Models to FPGAs," in *MICRO*, 2016.
[18] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "DianNao: A Small-footprint High-throughput Accelerator for Ubiquitous Machine-learning," in *ASPLOS*, 2014.
[19] H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, J. K. Kim, V. Chandra, and H. Esmaeilzadeh, "Bit Fusion: Bit-Level Dynamically Composable Architecture for Accelerating Deep Neural Networks," in *ISCA*, 2018.
[20] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An Energy-efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks," *JSSC*, 2017.

[21] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks," in *ISCA*, 2017.

[22] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "TETRIS: Scalable and Efficient Neural Network Acceleration with 3D Memory," in *ASPLOS*, 2017.

[23] Y. Ma, Y. Cao, S. Vrudhula, and J.-s. Seo, "Optimizing Loop Operation and Dataflow in FPGA Acceleration of Deep Convolutional Neural Networks," in *FPGA*, 2017.

[24] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li, "FlexFlow: A Flexible Dataflow Accelerator Architecture for Convolutional Neural Networks," in *HPCA*, 2017.

[25] L. Song, X. Qian, H. Li, and Y. Chen, "PipeLayer: A Pipelined ReRAM-based Accelerator for Deep Learning," in *HPCA*, 2017.

[26] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-X: An Accelerator for Sparse Neural Networks," in *MICRO*, 2016.

[27] S. K. Esser, P. A. Merolla, J. V. Arthur, A. S. Cassidy, R. Appuswamy, A. Andreopoulos, D. J. Berg, J. L. McKinstry, T. Melano, D. R. Barch, C. di Nolfo, P. Datta, A. Amir, B. Taba, M. D. Flickner, and D. S. Modha, "Convolutional Networks for Fast, Energy-Efficient Neuromorphic Computing," *ArXiv*, 2016.

[28] D. Mahajan, J. Park, E. Amaro, H. Sharma, A. Yazdanbakhsh, J. K. Kim, and H. Esmaeilzadeh, "Tabla: A Unified Template-based Framework for Accelerating Statistical Machine Learning," in *HPCA*, 2016.

[29] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "Prime: A Novel Processing-in-memory Architecture for Neural Network Computation in ReRAM-based Main Memory," in *ISCA*, 2016.

[30] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing," in *ISCA*, 2016.

[31] X. Yang, J. Pu, B. B. Rister, N. Bhagdikar, S. Richardson, S. Kvatinsky, J. Ragan-Kelley, A. Pedram, and M. Horowitz, "A Systematic Approach to Blocking Convolutional Neural Networks," *ArXiv*, 2016.

[32] S. Han, H. Mao, and W. J. Dally, "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization, and Huffman Coding," in *ICLR*, 2016.

[33] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "ISAAC: A Convolutional Neural Network Accelerator with In-situ Analog Arithmetic in Crossbars," in *ISCA*, 2016.

[34] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "ShiDianNao: Shifting Vision Processing Closer to the Sensor," in *ISCA*, 2015.

[35] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks," in *FPGA*, 2015.

[36] T. Moreau, M. Wyse, J. Nelson, A. Sampson, H. Esmaeilzadeh, L. Ceze, and M. Oskin, "SNNAP: Approximate Computing on Programmable SoCs via Neural Acceleration," in *HPCA*, 2015.

[37] S. Eldridge, A. Waterland, M. Seltzer, J. Appavoo, and A. Joshi, "Towards General-Purpose Neural Network Computing," in *PACT*, 2015.

[38] A. Yazdanbakhsh, J. Park, H. Sharma, P. Lotfi-Kamran, and H. Esmaeilzadeh, "Neural Acceleration for GPU Throughput Processors," in *MICRO*, 2015.

[39] B. Grigorian and G. Reinman, "Accelerating Divergent Applications on SIMD Architectures Using Neural Networks," *TACO*, 2015.

[40] R. S. Amant, A. Yazdanbakhsh, J. Park, B. Thwaites, H. Esmaeilzadeh, A. Hassibi, L. Ceze, and D. Burger, "General-Purpose Code Acceleration with Limited-Precision Analog Computation," in *ISCA*, 2014.

[41] B. Belhadj, A. Joubert, Z. Li, R. Héliot, and O. Temam, "Continuous Real-World Inputs Can Open Up Alternative Accelerator Designs," in *ISCA*, 2013.

[42] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural Acceleration for General-Purpose Approximate Programs," in *MICRO*, 2012.

[43] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun, "NeuFlow: A Runtime Reconfigurable Dataflow Processor for Vision," in *CVPR Workshops*, 2011.

[44] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, "Stream-Dataflow Acceleration," in *ISCA*, 2017.

[45] K. Wang and C. Lin, "Decoupled Affine Computation for SIMT GPUs," in *ISCA*, 2017.

[46] T. Chen and G. E. Suh, "Efficient Data Supply for Hardware Accelerators with Prefetching and Access/Execute Decoupling," in *MICRO*, 2016.

[47] J. E. Smith, "Decoupled Access/Execute Computer Architectures," in *ACM SIGARCH Computer Architecture News*, 1982.

[48] M. Benhenda, "ChemGAN challenge for drug discovery: can AI reproduce natural chemical diversity?," *arXiv*, 2017.

[49] Y. Li, J. Song, and S. Ermon, "Inferring The Latent Structure of Human Decision-Making from Raw Visual Inputs," *ArXiv*, 2017.

[50] H. Che, B. Hu, B. Ding, and H. Wang, "Enabling Imagination: Generative Adversarial Network-Based Object Finding in Robotic Tasks," in *NIPS*, 2017.

[51] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "CACTI-P: Architecture-level Modeling for SRAM-based Structures with Advanced Leakage Reduction Techniques," in *ICCAD*, 2011.

[52] "DDR4 Spec - Micron Technology, Inc." https://goo.gl/9Xo51F.

[53] H. J. Siegel, L. J. Siegel, F. C. Kemmerer, M. PT Jr, S. HE Jr, and S. D. Smith, "PASM: A Partitionable SIMD/MIMD System for Image Processing and Pattern Recognition," *IEEE TC*, 1981.

[54] A. Nieto, D. L. Vilarino, and V. M. Brea, "PRECISION: A reconfigurable SIMD/MIMD coprocessor for Computer Vision Systems-on-Chip," *IEEE TC*, 2016.

[55] A. N. Choudhary, J. H. Patel, and N. Ahuja, "NETRA: A Hierarchical and Partitionable Architecture for Computer Vision Systems," *IEEE TPDS*, 1993.

[56] H. P. Zima, H.-J. Bast, and M. Gerndt, "SUPERB: A Tool for Semi-Automatic MIMD/SIMD Parallelization," *Parallel Computing*, 1988.

[57] P. P. Jonker, "An SIMD-MIMD architecture for Image Processing and Pattern Recognition," in *Computer Architectures for Machine Perception*, 1993.

[58] A. Nieto, D. L. Vilariño, and V. M. Brea, "SIMD/MIMD Dynamically-reconfigurable Architecture for High-performance Embedded Vision Systems," in *ASAP*, 2012.

[59] H. M. Waidyasooriya, Y. Takei, M. Hariyama, and M. Kameyama, "FPGA Implementation of Heterogeneous Multicore Platform with SIMD/MIMD Custom Accelerators," in *ISCAS*, 2012.

[60] X. Wang and S. G. Ziavras, "Performance-energy Tradeoffs for Matrix Multiplication on FPGA-based Mixed-mode Chip Multiprocessors," in *ISQED*, 2007.

[61] G. Kim, K. Lee, Y. Kim, S. Park, I. Hong, K. Bong, and H.-J. Yoo, "A 1.22 TOPS and 1.52 mW/MHz Augmented Reality Multicore Processor with Neural Network NoC for HMD Applications," *JSSC*, vol. 50, no. 1, 2015.