EcoFlow: Efficient Convolutional Dataflows on Low-Power Neural Network Accelerators

Lois Orosa, Skanda Koppula, Yaman Umuroglu, Konstantinos Kanellopoulos, Juan Gómez-Luna, Michaela Blott, Kees Vissers, Onur Mutlu

Abstract—Dilated and transposed convolutions are widely used in modern convolutional neural networks (CNNs). These kernels are used extensively during CNN training and inference of applications such as image segmentation and high-resolution image generation. We find that commonly-used low-power CNN inference accelerators are *not* optimized for both these convolutional kernels. Dilated and transposed convolutions introduce significant zero padding when mapped to the underlying spatial architecture, significantly degrading performance and energy efficiency. Existing approaches that address this issue require significant design changes to the otherwise simple, efficient, and well-adopted architectures used to compute direct convolutions. To address this challenge, we propose EcoFlow, a new set of dataflows and mapping algorithms for dilated and transposed convolutions. These algorithms are tailored to execute efficiently on existing low-cost, small-scale spatial architectures and requires minimal changes to existing accelerators. At its core, EcoFlow eliminates zero padding through careful dataflow orchestration and data mapping tailored to the spatial architecture. We evaluate EcoFlow on CNN training workloads and Generative Adversarial Network (GAN) workloads. Experiments in our new cycle-accurate simulator show that, using a common CNN inference accelerator, EcoFlow 1) reduces end-to-end CNN training time between 7-85%, and 2) improves end-to-end GAN training performance between 29-42%, compared to state-of-the-art CNN dataflows.

Index Terms—Convolutional Neural Networks, Hardware Accelerators

1 Introduction

Deep convolutional neural networks (CNNs) have been widely adopted to solve hard problems in computer vision, natural language understanding, speech processing, medical applications, and more [1], [2], [3]. Transposed and dilated convolutions are the two key workhorses used to train CNNs, and run a variety of other deep learning models. For example, both kernels are employed in applications requiring significant upsampling or downsampling to process high-resolution media such as image generation (using Generative Adversarial Networks (GANs) and Variational Auto-encoders (VAEs) [4])), image super-resolution [5], and image segmentation [6]. Additionally, more emerging machine learning works in text-to-speech generation [7], speech recognition [8], and audio synthesis [9] use dilated convolutions. Other experimental machine learning models, such as hierarchical capsule networks [10] and dilated residual networks [11] for improved image modeling, use both these convolution types.

Meanwhile, specialized architectures for CNN inference have gained traction to support the demand for low-cost deep learning on a variety of devices, such as Internet-of-Things devices (IoT) [12], or various embedded electronics [13]. While these works demonstrate efficient execution of direct convolutions (i.e., regular or 'standard' convolutions), we find that existing dataflows for transposed and dilated convolutions are poorly tailored for these architectures, causing significant bottlenecks for emerging edge workloads that use transpose and dilated convolutions. Despite this issue, these workloads are of growing interest to manufacturers, because they can enable: (1) on-device model training for improved user data privacy [14], (2) high-resolution image generation critical for augmented reality [15], (3) real-time speech recognition and generation [16], and many other applications employing dilated and transposed convolutions.

To address this issue, we introduce *EcoFlow*, a new set of dataflows and data mappings designed to efficiently perform transposed and dilated convolutions on low-cost, small-scale spatial architectures that are already widely in-use for regular CNN inference. We identify key bottlenecks introduced by these operations, originating from structural and predictable padding and zero-insertions required to up- and down-sample feature maps. EcoFlow circumvents these bottlenecks by meticulously orchestrating the data mapping and dataflows onto the target spatial architecture. The key idea is based on the observation that, with small modifications to the Network on Chip (NoC) and a proper data reorganization, dilated and transposed convolutions can be executed very efficiently in a common low-power CNN inference accelerator. EcoFlow is based on two key ideas. First, a data organization mechanism that avoids the structural and predictable zero padding required by dilated and transpose convolutions to be dispatched to the PE array, neither it creates unnecessary bubbles in the PE array. Second, a dataflow that uses regular data distribution patterns in the PE array that keeps the NoC design very simple, just requiring minimal changes to the multicast NoC of a CNN inference accelerator. Using these ideas, EcoFlow achieves significant improvements in performance and energy consumption.

We improve on several prior works that propose specialized accelerators that target specifically either transposed convolutions [17], [18], dilated convolutions [17], [18], or general sparsity [19]. We generalize, simplify, and significantly reduce the required architectural changes to support exactly the structured sparsity of these convolutional kernels. Our design goal is to avoid highly-specialized

- *Lois Orosa is with ETH Zurich and Galicia Supercomputing Center.*
- *Konstantinos Kanellopoulos, Juan Gómez-Luna and Onur Mutlu are with ETH Zurich.*
- *Skanda Koppula is with DeepMind.*
- *Yaman Umuroglu, Michaela Blott, and Kees Vissers are with AMD.*

accelerator architectures that are markedly different from common and well-understood spatial architectures (i.e., a matrix of processing elements working in a systolic array fashion) optimized for direct convolutions. Re-use of existing hardware architectural designs permits lower testing and manufacturing costs.

We make the following key contributions:

- We propose EcoFlow, a new set of dataflows and data mappings that enable efficient execution of transpose and dilated convolutions on CNN inference accelerators by introducing minimal hardware changes (Section 4).
- We develop a cycle-accurate spatial architecture simulator to evaluate EcoFlow. Our architectural simulator includes TPU [20], Eyeriss [21], and EcoFlow models, and it supports efficient execution of transposed, dilated, and direct convolutions (Section 5).
- We comprehensively evaluate the performance and energy efficiency of EcoFlow. Our evaluation shows that EcoFlow: 1) reduces end-to-end CNN training time between 7-85%, and 2) improves end-to-end GAN training performance between 29-42%, compared to state-of-the-art CNN dataflows.

2 Background

A deep convolutional neural network (CNN) is a neural network with one or more convolutional layers. A convolutional layer in a CNN applies a sliding filter to a 2D or 3D matrix that represents the input image or intermediate layer input. The input and output matrices to a convolutional layer are referred to as the *input feature map (ifmap)* and *output feature map (ofmap)*, respectively. The *filter* (or *kernel*) is the sliding filter that is applied to the ifmap to calculate the ofmap. In each convolutional layer, there are usually multiple filters applied in parallel to the ifmap, producing multiple output matrices that compose the ofmap. The stride of a convolution refers to the size of the step that the convolutional filter takes while sliding through the ifmap. The core operation of a CNN is a multiply-and-accumulate (MAC) operation. Modern CNNs may have up to 10^{18} MACs performed during one forward evaluation. Most of these MAC operations are performed in the convolutional layers.

Inference is the production phase where the CNN classifies unknown images. Before performing inference in production, the network is trained with and algorithm called backpropagation, that includes the calculation of input gradients and filter gradients. For a detailed treatment of CNN operation and gradient computation, we refer the reader to [22].

2.1 Different Types of Convolutions in CNNs

Figure 1 illustrates the three main types of convolutions we can find in convolutional neural networks. This examples shows the case for the CNN training phase of a convolutional neural network, where we find direct convolutions in the forward pass, and transposed and dilated convolutions in the backward pass.

2.1.1 Direct Convolution

A direct convolution (also known as convolution, standard convolution, or regular convolution) is one of the most

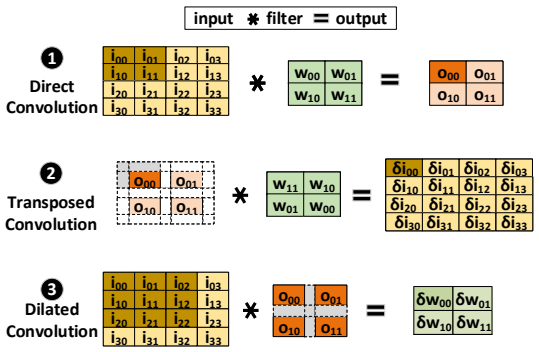


Fig. 1: Different types of convolutions with an example 4x4 input, 2x2 filter, and stride 2 used in the CNN training phase.

common operations in convolutional neural networks (in both inference and training), and other variants of CNNs [4], [9], [10]. A direct convolution is performed by sliding the filter (W_{xy}) over the input (i_{xy}) with a specific stride (stride 2 in the example ❶ in Figure 1), generally starting at the top left corner, so as to move the filter to the boundary of the input (❶ in Figure 1).

2.1.2 Transposed Convolution

A transposed convolution operation forms the same connectivity as a direct convolution but in the backward direction, which requires upsampling the input into an output of larger dimensions. Transposed convolutions are commonly used in CNN training and in emerging CNN workloads [11], [8], [9], [5]. Figure 1 ❷ shows an example that calculates the input gradients (δi_{xy}) in the backward propagation pass of CNN training. A transposed convolution is computed by convolving the error matrix (O_{xy}) with the forward pass filter (W_{xy}) rotated 180°. Transposed convolutions introduce zero padding into the error matrix to produce an output of larger dimensions, up-sampling the backpropagated errors. The error matrix might require zero-padding in the borders, as in Figure 1 ❷. If the stride is greater than one (the example Figure 1 ❷ has stride 2), the error matrix also require internal zero padding as zero-valued rows and columns.

2.1.3 Dilated Convolution

Dilated convolutions are commonly used in CNN training and in emerging CNN workloads [11], [8], [9], [5]. Figure 1 ❸ shows a dilated convolution example that calculates the filter gradients (δW_{xy}) with dilation rate = 2 (i.e., stride 2) in the backward propagation pass of CNN training. A dilated convolution is computed by convolving the input (i_{xy}) with a padded filter (O_{xy}) to augment its dimensions. This convolution inserts zero padding as rows and columns in the filter when the dilatation rate (i.e., the stride of the convolution when training a CNN) is greater than one. A dilation rate of 1 does not introduce any padding in the filter.

2.2 Spatial Architectures for CNN Inference

A spatial compute array is the key component in many popular low-cost CNN accelerators [21], [23], [20]. A spatial architecture consists of a matrix of simple processing

elements (PEs), interconnected with one or several internal networks. Each PE is able to perform a MAC operation. By orchestrating data into and out of the PE network, spatial architectures can efficiently implement either matrix multiplications or convolutions. Examples of spatial architectures include Eyeriss V1/V2 [21], [23], Google's TPU [20], NVIDIA's CUDA Tensor Cores [24], and SCNN [25].

Figure 2 illustrates the core elements of a common spatial architecture for CNN inference. At the core is an array of interconnected PEs. Data is cached on a global on-chip buffer, which utilizes various network-on-chips (NoCs) to exchange data with the PE array. In common designs [21], this network enables data transfer between vertically adjacent PEs, simultaneous broadcast to all PEs, and multicasting values to individual sets of PEs. On the left of Figure 2 we can see the off-chip memory that stores temporary data that overflows the global buffer, and the complete set of ifmaps, filters, and final ofmaps. The internal architecture of the PEs (right side of Figure 2) can differ slightly, based on the chosen dataflow, accelerator function (e.g. sparse/non-sparse CNN acceleration), and other optimizations (e.g. reduced precision, clock-gating). PEs generally store small amounts of weight or partial sum data which is reused during dataflow [21].

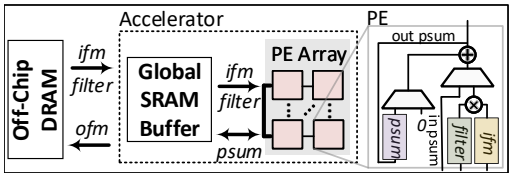


Fig. 2: Common Spatial architecture for CNN inference acceleration.

2.3 CNN Dataflows on Spatial Architectures

We describe the most widely-used dataflows for performing convolutions in spatial architectures, used to evaluate EcoFlow in Section 6. An in-depth discussion of each dataflow can be found in [21].

Convolution Dataflows. Row stationary (RS) [21] is a state-of-the-art dataflow for performing convolutions in spatial architectures. The RS dataflow attempts to minimize the overall energy consumed by off-chip data accesses by reusing the convolutional filters and ifmaps. RS minimizes data movement across all data types by effectively assigning each PE a 1D convolution to perform. The results of these 1D convolutions (or partial sums) are accumulated with other partial sums from other PEs to produce the final ofmap. RS has been shown to be the most energy efficient dataflow on spatial architectures [21], compared to Weight Stationary (WS) and Output Stationary (OS) dataflows.

Although previous works claim that the choice of dataflow is not critical for direct convolutions [26], in this work we demonstrate that this choice *does* matter for transposed and dilated convolutions. Using direct convolution dataflows for transposed and dilated convolutions can result into low performance and poor energy efficiency.

Matrix Multiplication Dataflows. Lowering a convolution into a matrix multiplication is a well known technique that is used today in many CNN frameworks and accelerators, i.e., TPUs [20]. For a detailed explanation of the lowering process, we refer the reader to [27]. After lowering, several dataflows can be used for the matrix multiplication. A common approach uses an output stationary dataflow in which partial sums are accumulated locally, and inputs are forwarded to adjacent rows [21]. The matrices are fed into the PE array from the top and left edges of the array [23]. This is the approach used in our reference implementation in Section 6.

3 Motivation and Goal

We describe the main inefficiencies of transpose and dilated convolutions, and how related works require a specialized accelerators to solve this problem (Section 3.1). Our goal in this paper is to introduce minimal changes to an existing DNN inference accelerator to perform tranpose and dilated convolutions (Section 3.2).

3.1 Inefficiencies of Transposed and Dilated Convolutions

To understand the mechanics and bottlenecks of transpose and dilated convolution, we analyze the backward pass of CNN training on representative convolutional layers with different strides from two common CNNs, ResNet-50 and AlexNet. Figure 3 shows the percentage of multiplications by zero required to compute both transposed and dilated convolutions. We observe that for strides larger than 1, the zero multiplications dominate utilization by large margins. For example, more than 70% of multiplications for 2-stride convolutions are zero. The larger the stride, the larger the number of zero multiplications.

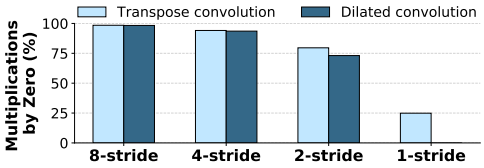


Fig. 3: Padding-induced zero multiplications in transpose and dilated convolutions during input and filter gradient calculation of representative CNN layers with different strides.

We make two observations. First, the PEs that execute zero operations *cannot* be used to perform useful operations, which causes resource under-utilization. Second, although the result of the multiplication is zero, inputs coming from other PEs might need to be accumulated and transmitted to the next node, which practically increases the latency of useful computations and reduce performance.

3.1.1 Analyzing Transpose Convolutions

Performing a transposed convolution in a spatial architecture designed for CNN inference requires significant padding to obtain the correct ofmap dimensions (i.e., up-sampling). Figure 4 shows two examples of the required padding in the input for obtaining the desired up-sampled ofmap[1]. In the example, layer Ⓐ requires 40 outer padding elements in the inputs (81% of the matrix), and layer Ⓑ requires 40 outer padding elements and 5 inner padding elements in the inputs (92% of the matrix).

1. The higher the stride, the higher the up-sampling.

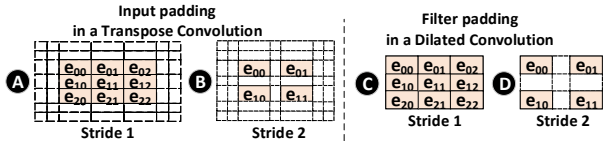


Fig. 4: Example of the zero-padding required to calculate transpose and dilated convolutions.

We can formulate the amount of padding required by a particular transposed convolution by considering ifmap, stride, filter sizes. For a $N \times N$ ifmap, $K \times K$ filter, and stride S, the number of inner padding elements is given by $[S\,(N-1)+1]^2 - N^2$. The number of outer padding elements is given by $4\,(K-1)\,[S\,(N-1)+1] + 4\,(K-1)^2$. The total number of zero-padding elements increases *linearly with the ifmap size*, and *quadratically with the stride*.

Transposed convolutions are used for upsampling a input to produce a high-resolution output feature map or media. For example, super-resolution [5] CNNs output images that are of the same or higher resolution than their input. Generative Adversarial Networks [28] use transposed convolutions for the same purpose.

Existing proposals. There are several works that propose to accelerate transposed convolutions with specialized GAN accelerators [17], [29]. Although these works achieve significant performance and energy improvements, they do it at the cost of designing a specialized accelerator for GANs instead of maintaining a simple, efficient, and more general spatial architecture optimized for CNN inference.

3.1.2 Analyzing Dilated Convolutions

Figure 4 illustrates two examples (C and D) of the required filter zero padding in a dilated convolution. Unlike in transposed convolution, the error matrix is only padded internally. In C, the stride is one, so the filter gradients can be calculated without padding. When the stride is larger than one, filter gradient calculation requires inner padding. D shows an example of this, with stride 2. 56% of the padded error matrix is zero. The amount of inner padding follows the same trend as above, increasing linearly with the ifmap size and quadratically with the stride.

Dilated convolutions are used in the forward pass of a handful of emerging, state-of-art classification networks [11].

Existing proposals. DT-CNN [18] proposes an specialized hardware accelerator to perform both transposed and dilated convolutions using delay cells. Unlike EcoFlow, DT-CNN is a specialized architecture customized for optimizing image segmentation workloads.

3.2 Goal

Our proposal builds on two key observations: (1) the padding required to perform transposed and dilated convolutions on spatial architectures has a very negative effect on efficiency, and (2) the padding is strictly determined by the characteristics of the convolution and the dimensions of the feature maps and kernel, and thus the location of zero-values is static and deterministic. Our goal in this work is to exploit these two observations in order to (1) eliminate zero padding to avoid low resource occupation, (2) minimize energy and memory requirements, (3) maximize throughput, and (4) introduce minimal changes to the spatial architecture of common CNN inference accelerators. To this end, we develop EcoFlow.

4 EcoFlow

We introduce EcoFlow, a new set of efficient dataflows and data mapping algorithms for calculating transpose convolutions (Section 4.1) and dilated convolutions (Section 4.2) in spatial architectures of CNN accelerators that are optimized for executing direct convolutions.

An inference CNN accelerator (e.g., Eyeriss [30]) executes only direct convolutions, thus, to perform a dilated or transposed convolution, it needs to make the data transformations illustrated in Section 2.1 (a.k.a., introducing zero padding) for adapting to the existing dataflow supported in the inference CNN accelerator, which causes bubbles and inneficiencies in the PE array. EcoFlow avoids introducing zero padding based on one key observation: dilated and transposed convolution data can be distributed more naturally in the inference CNN accelerator (a.k.a., without introducing zero padding) by just having a slightly improved multicast network that extends the original multicast controllers, which can support receiving data only from one multicast ID, for supporting receiving data from more than one multicast ID (5 IDs in our implementation, see Section 4.4) for more details). Based on this observation, we propose to 1) add support for multiple IDs in the multicast controllers (see Section 4.4), and 2) rebuild the dataflow and data mapping to adapt to this new architecture, which allows data distribution more adapted for transposed and dilated convolutions that prevents from introducing zero padding in the compilation time.

This section explains how EcoFlow meticulously orchestrates the dataflow and data mapping as to avoid zero padding and occupy PEs with only useful operations, asuming the minimal hardware changes proposed in Section 4.4. The dataflow and mapping onto hardware is computed at *compile time*. EcoFlow's mapping is more complex than other state-of-the-art dataflows, but this added complexity is a one-time cost during the initial compilation step. The compiler calculates a Finite State Machine (FSM) that is loaded into the PEs to perform the convolutions at runtime. We explain the details of the hardware architecture in Section 4.4.

4.1 Transpose Convolutions

In this section, we explain the steps EcoFlow takes during compilation time (Section 4.1.1) and runtime (Section 4.1.2) to perform transposed convolutions.

Without loss of generality, we use an example of the transposed convolution that calculates the input gradients in the CNN training algorithm. In this context, the input of the convolution is the padded error (the amount of padding depends on stride in the forward pass), the filter corresponds to the rotated filter from the forward pass, and the output of the convolution are the calculated input gradients.

4.1.1 Compilation Time

The EcoFlow compiler determines (1) the computation scheduling required to compute the (transposed) convolution and (2) the mapping of computations onto the architecture's PEs array.

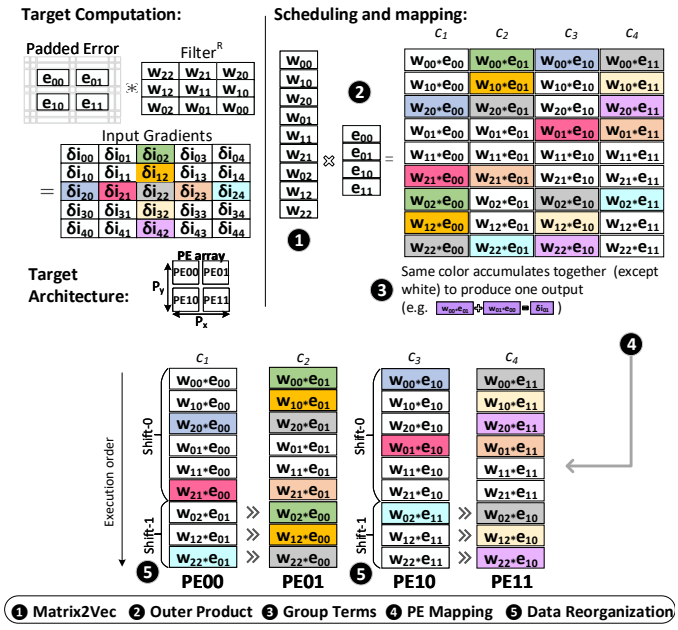


Fig. 5: Example of transposed convolution for calculating the input gradients on CNN training algorithm using EcoFlow. The symbol >> represents a column element `shift` by one.

EcoFlow follows five steps to calculate the **computation scheduling** and **mapping**. To improve clarity, we walk through each step using the example in Figure 5: a transposed convolution with stride 2, 5×5 output (i.e., input gradients), 3×3 filter (i.e., rotated filter), and 7×7 input (i.e., padded error) reshaped using padding from the original 2×2 error):

❶ The EcoFlow compiler converts the rotated filter and the error matrix into symbolic vectors. In Figure 5, these vectors have dimensions 9×1 and 4×1, respectively.

❷ The compiler performs the symbolic outer product of both vectors by multiplying all elements of the filter by all elements of the error matrix. The resulting matrix contains all multiplications required to perform the transposed convolution for input gradient calculation. Each gradient is the sum of some subset of these products. Notably, this matrix does *not* contain any zero multiplication due to padding. In our example, this matrix has dimension 9×4.

❸ EcoFlow determines which matrix elements have to be accumulated together to produce a single input gradient, and marks them with the same *label*. The labels are determined by doing a transposed convolution with placeholder symbols. In the example, cells with the same color represent matrix elements with the same label. The exception to this are the white cells, which produce a single gradient by themselves; white cells do not need to be accumulated with other values. Note that at this step, each PE is assigned with data from different colors.

❹ The compiler assigns each column of symbolic computations to a different PE. The mapping assigns consecutive columns to consecutive PEs, from left to right and from top to bottom in the PE array. The number of PEs used by EcoFlow is equal to the dimensions of the error matrix. In the example, the PE array is composed by 2×2 array, shown in the bottom left. This mapping can be reorganized to reduce the number of required PEs (see *Grouping*).

❺ The multiplications are reorganized with the goal of leveraging local point-to-point network to accumulate partial sums across connected, vertically-adjacent PEs. EcoFlow maps multiplications that must accumulate together either into the same PE, or across vertical PEs. The reorganization consists of *circular shifting* of these multiplication blocks across horizontal PEs. Each block shifts $\lfloor \frac{w\_idx}{W_x \times stride} \rfloor$ PEs over, where W_x is one dimension of the filter ($W_x = 3$ in the example) and w_idx is the index of the computation in the order of execution in each PE (e.g., $w_{00} * e_{00}$ has $w\_idx = 0$, $w_{10} * e_{00}$ has $w\_idx = 1$, etc.). Since the shifting is circular across horizontal PEs, computation blocks in the upper row of PEs shift from PE00 to PE01 and from PE01 to PE00 in the example. In the lower row, computation blocks shift between PE10 and PE11.

The formula is derived directly from the transposed convolution algorithm algorithm, and adapted to the the EcoFlow network that connects the PEs. The goal of shifting is to place all operations that need to accumulate together (same color) vertically in the PE array. Intuitively, a larger stride would make the filter to slide through the input in larger steps, so the shift formula is used to correctly map this particularity of the algorithm to the PE arquitecture.

In Figure 5, the first six computation blocks are not shifted ($\lfloor \frac{w\_idx}{3 \times 2} \rfloor = 0$ for $0 \leq w\_idx < 6$), but the next three blocks are shifted over to the horizontally adjacent PE ($\lfloor \frac{w\_idx}{3 \times 2} \rfloor = 1$ for $6 \leq w\_idx \leq 9$). As a result of this reorganization, all the data that needs to be accumulated together is placed vertically. For example, the light blue multiply operations ($w_{22} * e_{01}$ and $w_{02} * e_{11}$) are shifted so they accumulate across on vertically adjacent PEs, PE00 and PE10.

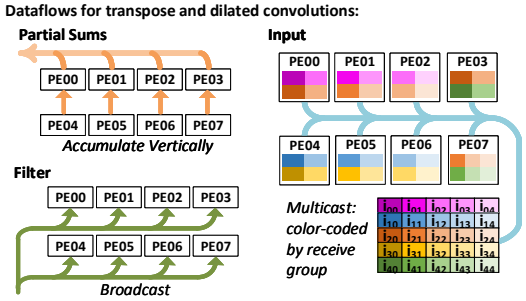


Fig. 6: Dataflow for each data type for transpose and dilated convolutions used in CNN training to calculate the input and filter gradients.

The EcoFlow compiler also performs optimization techniques, called *grouping* and *expansion*, that allows to group high-dimension convolutions into a small PE array, or to expand small-dimension convolutions into a large PE array.

Our proposed mapping and dataflow algorithms guaranty that the accumulation between PEs happens in the same column, so it avoids complex routing of data within the PE array, or the introduction of a more complex on-chip network. Note that the original Eyeriss accelerator is designed for direct convolutions, so to execute a dilated convolution Eyeriss needs a transformation that introduces padding in the data (see Section 2.1).

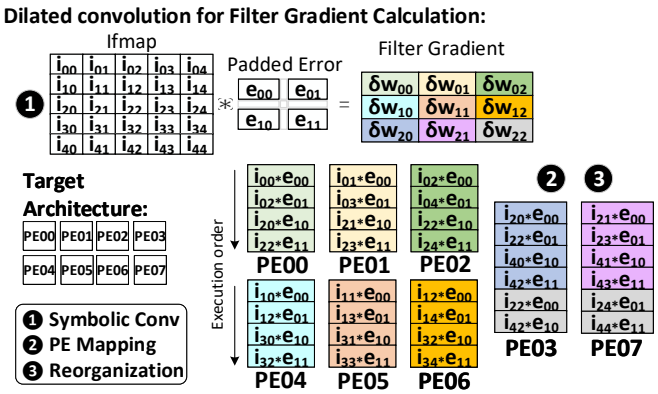


Fig. 7: Dilated convolution using EcoFlow to calculate the filter gradients in CNN training.

To validate that the proposed mapping and dataflow algorithms work, we empirically test all combinations of input images size (from 3x3 up to 224x224) and filter sizes (from 1x1 up to 15x15) in our SASIM simulator (Section 5) to validate the obtained results against the expected results.

4.1.2 Runtime.

The **dataflow** in EcoFlow leverages existing connections between vertically adjacent PEs and the on-chip multicast network present in spatial architectures. In this section, we describe data feeding and flow of the *partial sums*, *weights*, and *error maps* through the PE array. Figure 6 summarizes the dataflow of the three data types through the PE array.

Partial sums are accumulated locally and passed upward. Each filter-error product is added to a PE-local accumulation register. If a multiplication is the last one for a particular label (i.e., a color group) in one PE, the accumulated result is passed upward to the next PE in the same column. In Figure 5, the calculation of gradient element δi_{22} needs three steps. First, PE11 and PE01 compute $w_{00} \times e_{11}$ and $w_{20} \times e_{01}$, respectively. The results are stored in their internal accumulation registers. Next, the PE's compute $w_{02} \times e_{10}$ and $w_{22} \times e_{00}$, adding the result to the accumulation register. Third, PE11 passes the value in its accumulation register to PE01, and PE01 adds the received value to its accumulation register. The result is δi_{22}, which is then stored into the off-chip memory.

Filter weights are sequentially broadcast to all PEs and consumed every cycle. In Figure 5, the first set of multiplications use w_{00}, which is used by all PEs ($w_{00} \times e_{00}$, $w_{00} \times e_{01}$, $w_{00} \times e_{10}$, $w_{00} \times e_{11}$). The next broadcast weight is w_{10}, and so on.

Error matrix elements are sequentially multicast to the PE array. Each PE maintains a list of multicast groups to which it is subscribed, and receives the error elements required. For example, in Figure 5, PE00 receives the multicast groups $\{e_{00}, e_{01}\}$. Multicast groups are determined at compile time and loaded into each PE as part of an FSM.

4.2 Dilated Convolution

A dilated convolution is a direct convolution with a modified kernel (i.e., padded kernel) to match the desired output dimensions. Without loss of generality, we use an example of dilated convolution that calculates the filter gradients in the CNN training algorithm. In this context, the input of the convolution is the ifmap from the forward pass, the filter corresponds to the padded error (the amount of padding depends on stride in the forward pass), and the output of the convolution are the calculated filter gradients. Generally, a dilated convolution has a more simple and regular data access pattern than a transposed convolution, which results into a more intuitive and simple EcoFlow mechanism.

4.2.1 Compilation Time

For a dilated convolution, EcoFlow performs computation scheduling and data mapping using a three-step process. For clarity, we walk through compilation using Figure 7 which illustrates filter gradient calculation with a 5×4 ifmap, 3×3 filter, and stride 2 convolutional layer.

❶ EcoFlow performs a symbolic convolution between the ifmap and the padded errors, determining the symbolic computations required to produce the filter gradients. During this step, the compiler forms groups that accumulate together to produce a single gradient element. In Figure 7, multiplications of the same color accumulate together.

❷ EcoFlow provisionally assigns the calculation of each filter gradient to one PE, eliminating inter-PE communications. Based on the necessity to parallelize channels in a filter, and to avoid potential slowdowns associated with large error maps, the compiler automatically reorganizes and re-distributes the compute schedule using assignment expansion, as explained in Section 4.2.2.

❸ Finally, the compiler determines multicast groups for the ifmap for use during execution. In the next section, we describe the dataflow for the partial sums, error matrix, and ifmap.

4.2.2 Runtime

EcoFlow uses a straightforward dataflow for calculating the filter gradients. Similar to the calculation of the input gradients, the calculation of the filter gradients adapts well to the underlying on-chip network in state-of-art spatial architectures. Figure 6 describes the three main dataflows.

Error matrix elements are broadcast to each PE simultaneously. In Figure 6, e_{00} is used by all PEs in their first cycle, and is the first error to be broadcast.

The *input matrix* is distributed to the PEs using a multicast pattern determined by the compiler. Figure 6 illustrates the input matrix multicast used for the convolution described in Figure 7. In Figure 6, we see that each PE is part of at least four receive groups, corresponding to the number input matrix elements required in its computation schedule.

Finally, *partial sums* are accumulated within the PE. Each PE is responsible for multiplying and accumulating all the data it receives, and for storing the resulting filter gradient into off-chip memory.

Expansion. The computation of a single convolution step can be distributed across vertical PEs, so the partial SUMs are accumulated vertically in the array. The top PE, after performing all the local operations, accumulates any passed-in results, and writes the final gradient to memory. For example, in Figure 7, the partial sum of the gray data in PE07 are transmitted vertically to PE03, which accumulates this partial sum from PE07 with its local operations.

4.3 Memory Management

We describe EcoFlow's data reuse using two concepts from [21]. First, a *PE set* is the subset of PEs used to run a 2D convolution. If the physical array is large enough, several PE sets can be mapped concurrently in the array. Second, a *processing pass* is the contained, simultaneous execution of 2D convolutions in the PE array. In a single transpose convolution processing pass, each input element is read once from the global buffer, and the partial sums are stored back to the global buffer only once.

For a transpose or a dilated convolution, EcoFlow has three types of reuse: 1) it reuses the input values by storing them in the global buffer using them with different filters, 2) it reuses the filters by broadcasting and using them across multiple PEs, and 3) it accumulates the partial sums within the PE and across vertical PEs. The filters are streamed from DRAM directly to the PE registers, and the inputs and partial sums are stored in the global buffer for reuse between processing passes. Algorithm 1 shows a nested loop describing the mapping of a processing pass into PE sets.

Algorithm 1 Mapping a Processing Pass into PE Sets

```
1  for cg = 0 to Q/R: # Channel group
2    for c = (cg × R) to (cg + R) # Channel
3      for fg = 0 to P/T # Filter Group
4        for i = 0 to N # Input
5          for f = (fg × T) to (fg + T) # Filter
6            # 2D Conv mapped to each PE set
7            out[i][f] += input[c][i] *
                 filter[c][f]
```

To map PE sets into a processing pass, EcoFlow uses five parameters: N, R, T, Q and P. EcoFlow fits $R \times T$ PE sets. Every T PE sets share the same inputs with T filters, and every R PE sets that run on R channels accumulate their partial sums within the PE array. Also, a processing pass can process N inputs, P filters and Q channels at the same time. These parameters depend on the size of the internal PE registers. EcoFlow exhausts reuse opportunities of inputs and partial sum across different processing passes.

To optimize these parameters and allocate global buffer space for inputs and partial sums, our compiler pass runs an optimization procedure that finds parameters that minimize energy consumption for a given hardware configuration.

Convolutions from different channels are accumulated and stored in the local registers. In case there is no space in the local registers, partial sums can be stored in the global buffer. In the workloads we evaluate (Section 6), partial sums are accumulated and stored in the local registers.

4.4 Hardware Architecture

EcoFlow targets spatial architectures similar to those described in Section 2.2. We use Eyeriss [21] as the baseline architecture, and we incorporate changes to the on-chip network and PE array to support EcoFlow.

On-Chip Network Requirements. The baseline architecture uses four on-chip networks: 1) a filter broadcast network to send filter weights to the PE, 2) an ifmap multicast network to send a unique ifmap element to each PE (i.e., one multicast group per PE), 3) an ofmap network that delivers partial sums to the global buffer, and 4) a network of local unidirectional point-to-point links that transmit partial sums through PEs in a column.

EcoFlow requires an expansion of the multicast network, so that each PE in the array can belong to several multicast groups. For example, in Figure 7, PE02 belongs to these four multicast groups: $\{i_{02}, i_{04}, i_{22}, i_{24}\}$. The multi-cast group i_{02} consists of PE00 and PE02, and likewise for other groups. To support this, we extend the original multicast network of Eyeriss [21]. Figure 8 shows the EcoFlow PE array architecture, and the details of the multicast controller, which include five IDs to support five multicast groups. To support an $R \times C$ array of PEs, Eyeriss has a vertical Y-bus consisting of R horizontal X-buses. Each X-bus has a row ID, and each PE has a column ID. These IDs are reconfigurable, allowing different layers to map onto the same array.

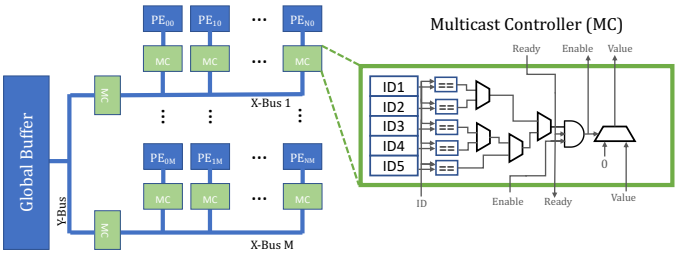


Fig. 8: EcoFlow NoC and Multicast Controller.

We extend this network to have several row IDs per X-bus, and several column IDs per PE. For a $N \times N$ filter with stride S, the total number of row IDs that each X-bus needs to store is given by $\lceil \frac{N}{S} \rceil$. The number of bits needed by each row ID is $\lceil (\log_2 2N - S) \rceil$. $2N - S$ quantifies the total number of groups in a row. The equations to calculate the column ID requirements are exactly the same. We size the ID registers to support the largest layers in the CNN. For example, AlexNet requires five 5-bit row IDs per bus, while ResNet-50 requires four 4-bit row IDs per bus.

We estimate the area overhead of our NoC modifications by accounting for the additional logic gates and storage elements required to support the worst case CNN evaluated in this work. The extra IDs and comparison logic affect all the PE multicast controllers within the PE array. Our results show that the additional changes in the NoC introduce a 2.9% area overhead in the PE array.

EcoFlow also uses larger bandwidth to keep all PEs continuously utilized. Table 1 shows the maximum bus width required by EcoFlow in the three networks to run at maximum throughput on all evaluated CNNs. First, EcoFlow requires a 64+16 bits wide multicast global input network (GIN) for filters+ifmaps (forward pass), for errors+filters (input gradient calculation), and for ifmaps+errors (filter gradient calculation). Second, EcoFlow requires a 64 bits wide global output network (GON) for ofmaps (forward pass), input gradients (input gradient calculation), and filter gradients (filter gradient calculation). Third, EcoFlow requires a 64 bits wide local network (Local) for transmitting psums between vertical PEs.

We observe that EcoFlow does not require additional bandwidth for GON and Local networks, and it requires 40% more bandwidth for the GIN network.

Fitting Different CNNs. In case the dimensions of a convolutional layer do not fit the size of the PE array, EcoFlow executes the convolution pass in two (or more) sequential

	GIN	GON	Local
Eyeriss	64 + 16 bits	64 bits	64 bits
EcoFlow	80 + 32 bits	64 bits	64 bits

TABLE 1: Bus bit width of the multicast global input (GIN), global output (GON), and local (Local) networks, for a 13×15 PE array.

steps, similar to prior works [30], at the cost of decreasing performance. Partial results across sequential steps are maintained in local registers. Also, if the dimensions of the PE array are larger than the dimensions of the convolutional layer, EcoFlow can increase parallelism (e.g., processing several channels in parallel) to use as many resources as possible and improve performance.

Network Scalability. The multicast bandwidth required when EcoFlow scales depends on the data reuse of the particular DNN network executed on the accelerator. For the DNN networks tested on our evaluation, if we increase the number of PE elements in the array (e.g., double), the bandwidth of multicast network would have to increase (e.g., double) to maintain a high utilization. Generally, the bandwith of the multicast network would have to increase linearly with the number of PE elements to keep a similar level of utilization.

PE Requirements. Like a typical PE design, EcoFlow needs an FSM to orchestrate loads and stores to registers, accumulations, stores to the global buffer, and communication with its neighboring PE. The compiler generates these FSMs. EcoFlow accumulates in each PE a variable amount of partial sums before the PE sends the result to the above PE or to memory, and it needs to accumulate the corresponding partial sums together (e.g., the same colors needs to accumulated together in Figure 5). This requires a slightly more complex FSM in the PE, compared to row-stationary dataflow.

Memory Requirements. EcoFlow does not require a different memory hierarchy than other spatial architecture accelerators. We use commodity DRAM chips and a highly banked global buffer.

5 SASIM: The Spatial Architecture Simulator

To evaluate EcoFlow, we develop SASIM, a new cycle-accurate simulator that mimics the hardware of a spatial architecture. SASIM models all the components of the PEs, the network, and the memory hierarchy. Each component of SASIM can be fully microprogrammed, and all the latency and energy parameters are fully parametrizable. SASIM can estimate the latency and energy consumed by direct, transpose and dilated convolutions of a particular layer; many other metrics such as PE utilization and bandwidth can be measured. We also develop a new compiler that automatically generates the signals required by SASIM to execute a particular CNN layer. We will *open-source* both the simulator and the compiler to help enable the development of new dataflows and high-accuracy simulation of new spatial architectures and dataflows.

5.1 The Simulator

SASIM models the on-chip hardware of a spatial architecture and off-chip DRAM memory. SASIM contains architecture models for Eyeriss [21] and TPU [20]. SASIM is extensible and fully programmable. The level of abstraction of SASIM is similar to RTL: we model a synchronous digital circuit in terms of the flow of digital signals (data) between hardware registers, and the logical operations performed on those signals. In addition to a timing simulator, SASIM is a functional simulator that propagates the input values through the PE array to get the output, which allows to validate that the implementation of the dataflow at microprogramming level is correctly implemented.

The simulator has three main components: (1) a PE array, each of which has a global buffer, local registers, pipelined multiply-and-accumulate unit, and input/output queues connected to neighbouring PEs (2) a network on chip that interconnects neighboring PEs and PEs to the global buffer, and (3) a highly banked global buffer (e.g., 27 banks in our evaluation in Section 6). All components update their state at every clock cycle.

The basic organization of the simulator is simple: 1) the components connect together according to the specific design of the PEs and networks, 2) all components are controlled through input and output signals that are microprogrammed, and 3) all components update their state cycle by cycle. All components of SASIM are configurable, including memory sizes, network bandwidth, energy parameters. We support two variants of PEs, one tailored for convolutions (e.g., Eyeriss) and one for tailored for matrix multiplications (e.g., TPUs).

5.2 The Compiler

For simplifying the generation of the microprogramming control signals for SASIM, we implement a compiler. The inputs to the compiler are all the characteristics of the hardware and the CNN layers (e.g., feature map and filter dimensions). SASIM can perform inference and training with row-stationary, TPU, or EcoFlow dataflows.

5.3 Validation

We validate SASIM by analyzing that the output values match the expected golden results, and that the timings and power consumption are similar to the results reported by a real chip Eyeriss accelerator [21]. For validation purposes, we use inference workloads, and we configure SASIM with PE integer units. In our evaluation (Section 6) we use training workloads, and we configure SASIM with PE floating point units. We configure SASIM with the same row-stationary dataflow parameters and the same accelerator configuration as reported in [21]. Table 2 shows the execution time, power, total size of global buffer accesses, and total size of all DRAM accesses for both Eyeriss and SASIM while running inference on AlexNet. We expect some variations because the Eyeriss paper [21] does not provide full detail about their exact procedure for measuring timing, and about their memory management mechanisms for convolutions with high filter/channel count that overflow the global buffer. We calculate the power based on the energy parameters for a 45nm technology node reported by Horowitz [31]. There are two challenges for validating the power. First, the technology node of the Eyeriss chip is 65nm, not 45nm. We

address this by scaling the energy consumption up by a factor of 1.4, based on estimations obtained from previous studies [32]. Second, SASIM does not model the energy of many details that have a large influence in the energy consumption, such as the clock network, which consumes between 33-45% of the power [21]. To estimate the total power consumption, we simulate the power consumption of the components included in SASIM, and we estimate the power consumption from the remaining components by using the data from a real chip reported in [30].

		CONV5	CONV4	CONV3	CONV2	CONV1
SASIM	Exec. Time	12.5ms	18.8ms	25ms	39.5ms	15.2ms
	Power	207mW	*	*	*	273mW
	GB acc.	23.8MB	35.6MB	66MB	74MB	16.8MB
	DRAM acc.	1.5MB	2.1MB	2.6MB	4.11MB	3.6MB
Eyeriss	Exec. Time	11ms	16ms	21.8ms	39.2ms	16.5ms
	Power	236mW	235mW	266mW	288mW	332mW
	GB acc.	24.9MB	37.4MB	50.2MB	77.6MB	18.5MB
	DRAM acc.	1.3MB	2.1MB	3.0MB	4.0MB	5.0MB

* Eyeriss [23] does not report the detailed power breakdown of these layers, so it is not possible to cross-verify these particular results.

TABLE 2: Comparison of execution time, global buffer (GB) accesses and DRAM accesses of SASIM and Eyeriss [21].

We observe that the results of SASIM are similar to the real chip Eyeriss measurements, and follow the same trends across layers. We make three key observations. First, the reported SASIM execution time is within 0.07% to 10% of the real Eyeriss accelerator. Second, the amount of data accessed in memory (GB and DRAM) by SASIM has a deviation of 0% to 24% from real measurements. Third, the power consumption reported by SASIM shows a good approximation, and the results are relatively accurate, despite the fact we could not model many details that are missing in the real Eyeriss chip paper.

We conclude that SASIM is an cycle accurate simulator that allows to model different spatial arrays with different NoCs and PE configurations at a microprogramming level of detail, which enables to functionally verify the correctness of dataflows and its implementation.

6 Evaluation

We evaluate transpose and dilated convolutions using workloads that contain both types of convolutions: CNN training (Section 6.2) and GAN training (Section 6.3).

6.1 Experimental Setup

We use the SASIM simulator and the SASIM compiler (Section 5) to evaluate EcoFlow. We model the energy of the accelerator with values obtained from a 45nm process [31]. We model DRAM energy using DRAMPower [33]. We compare EcoFlow to the row-stationary (RS) dataflow [21] used in Eyeriss and to a lowering-based convolutional dataflow used in TPUs [20]. Table 3 shows the configuration of the target architecture used in evaluation. We choose an array of 13×15 PE elements, matching prior work and tuned with RS and TPU dataflows to fit the dimensions of the evaluated layers.

We implement a clock-gating mechanism that activates when the PE receives a zero value [21]. This is included in all our baselines. The NoC of Eyeriss and EcoFlow are similar,

PE Array	13 x 15 PEs
PE Array Clock	200 MHz
PE Register File (ifmap, filter, psum)	75, 224, 24
PE Register Latency	1 cycle
Global Buffer	108KB / 27 banks
DRAM	4GB DDR4 1866MHz
Clock Gating	Zero Operations
Multiplier/Accumulator	2-stage/1-stage
I/O Queues	8 entries
On-chip Network Latency	1 cycle

TABLE 3: Configuration of the base CNN accelerator.

implementing dedicated networks for each data type. We use the on-chip networks described in Table 1. The TPU uses a much simpler NoC with only two uni-directional connections between neighbour PEs (for propagating input and filter values), while the partial sums are accumulated locally. We evaluate CNN training in Section 6.2 and GANs in Section 6.3.

Note that EcoFlow is a improved version of the Eyeriss accelerator that adds some small changes (see Section 4.4) that enable performing dilated and transposed convolutions very efficiently, but they do not affect direct convolutions. Thus, EcoFlow executes direct convolutions with the same performance as the original Eyeriss accelerator.

To estimate the execution time of the end-to-end CNN training algorithm (i.e., execution time of all layers), we first profile the evaluated models in GPU and CPU to get the average breakdown of the execution time per layer. To estimate the total performance gains, we use the latency results from a real chip [30] to infer the execution time of the components that are not simulated with SASIM.

6.1.1 Optimizing CNN Training for EcoFlow

To get the maximum benefit from EcoFlow on CNN training, we need to replace pooling layers with larger strides when possible. Prior work demonstrates that pooling can be replaced by a convolutional layer with increased stride *without loss in accuracy* [34]. The authors show that for the tested CNNs, when they replace pooling with a convolutional layer with 2-stride, there is no accuracy loss. We corroborate and extend these results with experiments of our own on six larger, more recent CNNs. We train two variants of each CNN topology: one with pooling layers and one with pooling layers replaced with larger stride. We use the CIFAR-10 and ImageNet training and test datasets, and retain the default learning hyper-parameters.

Table 4 summarizes our results. We observe that using a larger stride (Stride) instead of pooling layers marginally reduces accuracy (<2%), and in some cases, improves accuracy. This can be an acceptable trade-off in some applications, given the performance advantages.

	CIFAR-10			ImageNet		
CNN	**Original**	**Stride**	**Diff.**	**Original**	**Stride**	**Diff.**
ResNet-18	94.6%	94.2%	-0.4%	69.6%	69.5%	-0.1%
ResNet-101	94.6%	93.7%	-0.9%	77.6%	76.9%	-0.7%
DenseNet-201	94.0%	93.7%	-0.3%	78.6%	76.8%	-1.8%
VGG-19	92.5%	92.1%	-0.4	74.5%	74.6%	+0.1%
MobileNet-v2	90.7%	90.7%	+0.0%	74.7%	73.14%	-1.56%

TABLE 4: Accuracy comparison of CNNs that downsample using pooling layers (original) versus a larger stride (Stride).

6.2 CNN Training Evaluation

This section evaluates CNN training workloads. Table 5 details characteristics of 8 sample layers that we evaluate from six representative and widely-used CNNs, namely AlexNet, ResNet-50, Shufflenet, Inception, Xception, and MobileNet.

Our complete evaluation tests 72 layers in total. These layer topologies and networks encompass most of the layers used in popular networks, and include recent winning topologies of the ILSVRC competitions. We use a batch size of four in our evaluations. We also evaluate the variant of each layer that includes the larger stride optimization described in Section 6.1.1. We denote these layers with a suffix of `opt`.

CNN	Layer#	IFM	OFM	Filter	# Filts	Str.	Opt.
AlexNet	CONV1	3x224x224	55x55	11x11	64	4	Yes
AlexNet	CONV2	64x31x31	27x27	5x5	192	1	Yes
ResNet-50	CONV3	128x57x57	28x28	3x3	128	2	No
ShuffleNet	CONV2	58x57x57	28x28	3x3	58	2	No
ShuffleNet	CONV5	232x7x7	7x7	1x1	232	1	No
Inception	CONV3	192x17x17	8x8	3x3	320	2	No
Xception	CONV3	728x29x29	14x14	3x3	1	2	No
MobileNet	CONV5	512x15x15	7x7	3x3	1	2	No

TABLE 5: Eight of the 72 evaluated layers from three CNNs.

We train using 16 bits instead of the 32 bits used in typical training algorithms. A previous work [35] demonstrates, training with BFLOAT16 can achieve the same accuracy as training with FP32.

6.2.1 Performance results

Figure 9 shows the speedup of input gradient calculation through each layer in TPU, RS and EcoFlow dataflows, normalized to TPU. Similarly, Figure 10 shows the speedup of the filter gradient calculation for the three dataflows. The numbers on top of the TPU bars indicate the absolute execution time of TPU in *milliseconds*. The layers starting with the letter "o" (e.g., Alexnet o-CONV1) are the optimized versions of the layers (Section 6.1.1).

We make two main observations. First, the speedup of EcoFlow for calculating the input gradients compared to TPU and RS is very high for strides larger than 1. As shown in the figure, the speedup is close to 4x for stride 2 (e.g, resnet50 CONV3), 11x for stride 4 (Alexnet CONV1), and 52x for stride 8 (Alexnet opt CONV1). For stride 1, the speedup is from 0% (e.g., resnet50 CONV2) to 10% (Alexnet CONV3). Second, the speedup of EcoFlow for calculating the filter gradients compared to TPU and RS is also very large for stride larger than 1. The speedup is more than 3x for stride 2 (e.g., resnet50 CONV3), 15.6x for stride 4 (Alexnet CONV1), and 60.1x for stride 8 (Alexnet o-CONV1). We conclude that EcoFlow performs the backward pass much more efficiently than RS and TPUs in strides larger than 1, because both TPU and RS need to introduce padding for performing transpose and dilated convolutions, which cause a significant waste of resources.

Table 6 shows the speedup of the evaluated end-to-end CNN networks.

We make two observations. First, Alexnet greatly benefits from EcoFlow, because more than 80% of the execution time is dedicated to execute convolution layers following by pooling layers, or convolutional layer with stride larger than one. Second, ResNet-50, ShuffleNet, Inception, Xception and Mobilenet have smaller benefits because many of their convolutional layers have stride 1. We conclude that EcoFlow has very significant end-to-end benefits in networks that use strides in convolutional or pooling layers. Notice that other modern networks with larger strides, like EfficientNet, would also greatly benefit from EcoFlow.

	Speedup			Energy savings		
	TPU	Eyeriss	EcoFlow	TPU	Eyeriss	EcoFlow
Alexnet	1	0.94	1.83	1	0.97	1.38
ResNet-50	1	0.99	1.07	1	1.02	1.06
ShuffleNet	1	0.98	1.08	1	1	1.07
Inception	1	1.01	1.08	1	0.99	1.08
Xception	1	1.01	1.11	1	1.00	1.10
Mobilenet	1	1.01	1.09	1	1.00	1.08

TABLE 6: Speedup and energy savings of end-to-end CNN training of convolutional layers in different architectures, normalized to TPU (larger is better).

6.2.2 Energy Results

In this section, we evaluate the energy consumption of EcoFlow. PEs are clock gated when idle, and all other parameters are defined in Table 3.

Figure 11 shows the energy comparison of TPU, RS and EcoFlow for the input gradient and filter gradient calculation. The breakdown of the energy includes DRAM (DRAM), global buffer (GBUFF), internal scratchpad memories (SPAD), the multipliers and the adders (ALU), and all on-chip networks (NoC). We make four main observations. First, the energy consumption of EcoFlow is much lower than TPU and RS for strides larger than 1. For example, the maximum energy savings of EcoFlow is 26x for Alexnet-opt-CONV1 compared to TPU. For the filter gradients, EcoFlow saves up to 8.3x energy. Second, the energy savings of EcoFlow are coming mainly from SPAD and NoC, whereas the energy consumed by DRAM is maintained (e.g., Alexnet convolutional layers). Third, for some layers with stride 1 (e.g., resnet50-CONV4), EcoFlow consumes more energy than TPU and RS, because EcoFlow needs more DRAM accesses in these cases. Four, the energy of the filter gradient calculation is dominated by DRAM in some layers, e.g., resnet50-CONV4, resnet50-CONV2, since the errors in these layers have little reuse and are memory bound. This happens when the kernel size is small. EcoFlow is most energy efficient for layers that have stride and kernel larger than one.

We conclude that the additional padding required by TPU and RS training dataflows cause large energy inefficiencies due to 1) resource underutilization, 2) increased data movement (data has to go through PEs while not performing any useful operations) and 3) increased execution time (a.k.a., increased static energy).

6.3 GAN Training Evaluation

This section evaluates GAN training workloads. We use the GAN convolutional layers described in Table 3. We compare EcoFlow to GANAX [29], a hardware GAN accelerator that optimizes the execution of GANs by avoiding unnecessary zero computations. The key idea introduced by GANAX is to identify repeated patterns in the GAN computation and create different microprograms to execute each of this

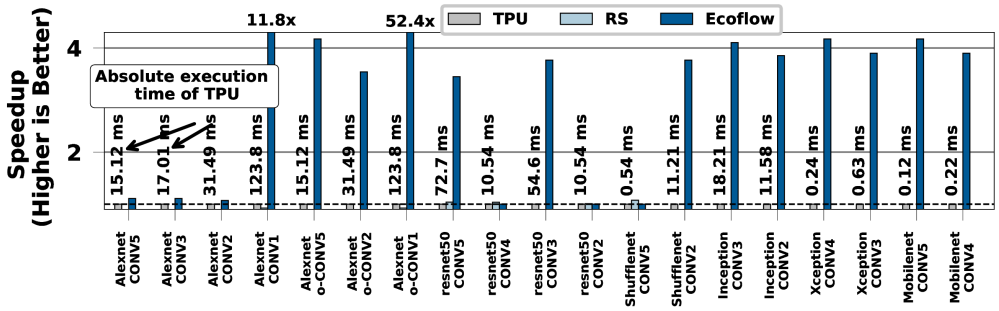


Fig. 9: Speedup of input gradient calculation, normalized to the TPU dataflow, and absolute TPU execution time.

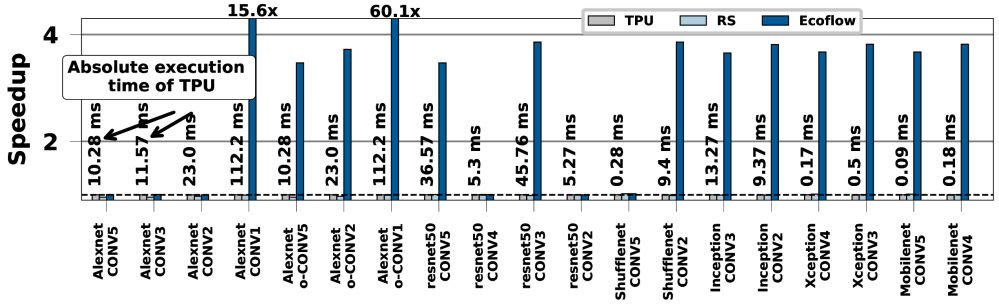


Fig. 10: Speedup of the filter gradient calculation, normalized to the TPU dataflow.

patterns. GANAX requires significant changes over an Eyeriss architecture, a new SIMD-MIMD execution model, a new ISA, a new global buffer to store instructions, and decoupling of the PEs into execution units and access units.

Table 7 shows the properties of the evaluated GAN layers. The layers are used by two representative GANs, namely CycleGAN, and pix2pix. The layers of the discriminator (Disc) are regular convolutional layers, and the layers of the generator (Gen) are transposed convolutions. EcoFlow accelerates the backward pass of the discriminator and the forward pass of the generator.

CNN	Layer#	IFM	OFM	Filter	# Filts	Str.
CycleGAN	Disc-CONV3	64x114x114	56x56	4x4	128	2
CycleGAN	Gen-TCONV1	256x56x56	113x113	3x3	128	2
pix2pix	Disc-CONV6	128x130x130	64x64	4x4	256	2
pix2pix	Gen-TCONV41	512x64x64	130x130	4x4	128	2

TABLE 7: Evaluated layers from two widely-used GANs.

6.3.1 Performance Results

Figure 12 shows the speedup of the backward (Input, Filter) and the forward passes of selected GAN layers, for RS, TPU, GANAX, and EcoFlow dataflows, normalized to RS. We make two observations. First, EcoFlow performs on the order of 4x better than RS and TPU. Because GANs use strides larger than 1 instead of pooling layers, EcoFlow accelerates most convolutional layers. Second, EcoFlow performs 3-4x times better than GANAX in the filter gradient calculations, because GANAX does not provide a dataflow to accelerate gradient calculation. However, GANAX performs very similar to EcoFlow in the forward pass of the generative layers, and in the calculation of the input gradients.

Table 8 shows the speedup of the evaluated end-to-end GAN networks.

	Speedup				Energy savings			
	TPU	Eye.	GANAX	EcoFlow	TPU	Eye.	GANAX	EcoFlow
pix2pix	1	0.95	1.34	1.39	1	0.93	1.11	1.29
CGAN	1	0.94	1.37	1.42	1	1.04	1.32	1.37

TABLE 8: Speedup and energy savings (higher is better) of end-to-end training of two GANs, normalized to TPU.

We make the key observation that EcoFlow has large benefits in end-to-end training of GAN networks. The training performance of EcoFlow outperforms even specialized GAN architectures like GANAX, because EcoFlow can accelerate filter gradient calculations.

6.3.2 Energy Results

Figure 13 shows the energy breakdown of the backward (Input,Filter) and the forward passes of selected GAN layers, for TPU, RS and EcoFlow dataflows, in absolute values. We could not compare to GANAX because some implementation details are missing in the paper (e.g., data reuse in each memory).

We make two main observations. First, the energy consumption of EcoFlow is much lower than the energy consumption of TPU and RS. For example, for the cyclegan-disc-CONV3 layer the energy savings of EcoFlow are in the order of 4x compared to TPU and RS. Second, similar to the results in CNN training (Section 6.2), the energy savings of EcoFlow are coming from reducing the energy in the SPADs, NOC and ALUs, whereas the DRAM energy consumption is very similar in all dataflows.

This article has been accepted for publication in IEEE Transactions on Computers. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TC.2023.3272282

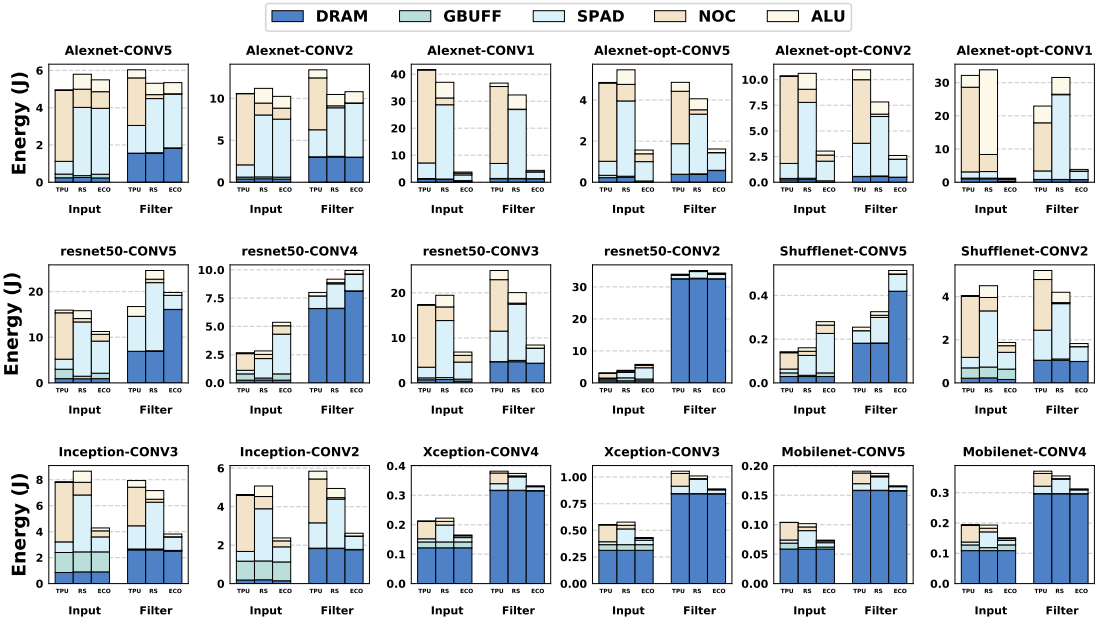


Fig. 11: Energy breakdown of the evaluated CNN layers.

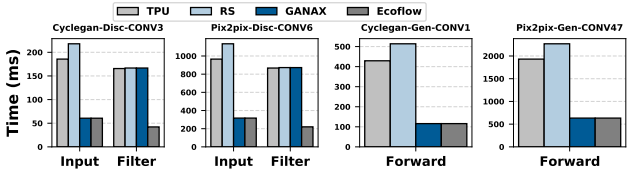


Fig. 12: Execution time of the evaluated GAN layers.

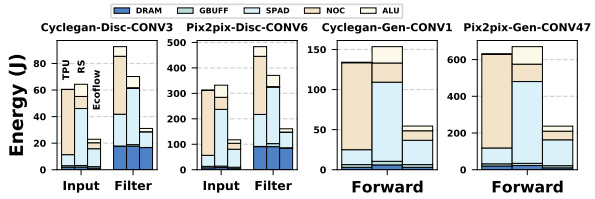


Fig. 13: Energy breakdown of the evaluated GAN layers.

A key property of GANs is that they use larger strides instead of pooling layers, so most of the layers of state-of-the-art GANs benefit from EcoFlow. We conclude that EcoFlow provides very significant performance gains and energy savings when training GAN worloads.

7 Related Work

To our knowledge, this is the first work to design an efficient dataflows to perform transpose and dilated convolutions on CNN inference spatial architectures. We have already extensively compared EcoFlow to TPU [20], Eyeriss [21] and GANAX [29]. In this section, we describe other related works.

Specialized Inference Accelerators. Most existing specialized CNN accelerators are optimized for direct convolutions commonly used on CNN inference (e.g. Eyeriss [21]). WaveCore [36] and Google's TPUv2 [20] support CNN training, but suffer from challenges highlighted in Section 3. EcoFlow solves these issues, while introducing minimal changes to the CNN inference accelerator architecture.

Sparse Accelerators. Sparse accelerators [19], [37], [38], [39], [25], [**?**], [40] address the inefficiencies caused by zeros contained in sparse matrices, which is a fundamentally different problem than padding introduced by transpose and dilated convolutions. EcoFlow can be incorporated to these accelerators to obtain aggregated benefits.

GAN Accelerators. Prior works focus on accelerating GANs by performing transposed convolutions on new memory technologies [41], FPGAs [42], and significantly modified spatial architectures [29]. Our work is unique in that 1) it focuses on both transposed and dilated convolutions, 2) it requires fewer hardware changes, 3) it proposes a multicast network that is able to effectively distribute the input data into the corresponding PEs, and 4) it evaluates GAN training and CNN training.

Winograd and Frequency-Domain Algorithms. Winograd is an alternative algorithm to perform matrix multiplications [43] that reduces the number of computations in CNNs via a series of data transformations. EcoFlow, however, targets the orthogonal problem of the zero padding introduced by the training algorithm to upscale and back-propagate the errors through the network. Frequency domain backpropagation [44] replaces convolutions with simple point-wise multiplications, which avoids the inefficiencies of transposed and dilated convolutions. However, this approach requires computationally-intensive Fast Fourier Transforms (FFTs) and Inverse FFTs (IFFTs) at the boundary of every layer, and it requires a larger memory footprint.

Other Algorithms. Direct convolutions [45] can avoid zero padding in the backward pass of some layers that meet some specific and restricted parameters. In contrast, EcoFlow is a general dataflow that can apply to the backward pass of any convolutional layer.

8 Conclusion

In this work, we aim to accelerate transpose and dilated convolutions in energy-efficient CNN inference spatial architectures. We observe that a main source of inefficiencies of state-of-the-art CNN inference accelerators when executing transpose and dilated convolutions is the large amount of required zero padding, which diminishes the overall energy efficiency and performance.

To address this issue, we propose EcoFlow, a new set of mapping and dataflow algorithms for transpose and dilated convolutions. EcoFlow eliminates zero-padding by meticulously orchestrating the scheduling, dataflow, and data mapping to fit the characteristics of the target CNN inference accelerator. We show that, by introducing minimal changes to the CNN inference hardware, EcoFlow can significantly improve the energy efficiency and performance of common CNN workloads.

References

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *CVPR*, 2016.

[2] N. Ramachandran, S. C. Hong, M. J. Sime, and G. A. Wilson, "Diabetic retinopathy screening using deep neural network," *Clinical & experimental ophthalmology*, 2018.

[3] C. Han, K. Murao, S. SATOH, and H. Nakayama, "Learning more with less: Gan-based medical image augmentation," *Medical Imaging Technology*, 2019.

[4] 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.

[5] C. Dong, C. C. Loy, K. He, and X. Tang, "Learning a deep convolutional network for image super-resolution," in *European conference on computer vision*, 2014.

[6] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," in *NIPS*, 2015.

[7] A. Van Den Oord and S. Dieleman, "Heiga zen, karen simonyan, oriol vinyals, alex graves, nal kalchbrenner, andrew senior, and koray kavukcuoglu. wavenet: A generative model for raw audio," *arXiv preprint arXiv:1609.03499*, 2016.

[8] K. J. Han, R. Prieto, and T. Ma, "State-of-the-art speech recognition using multi-stream self-attention with dilated 1d convolutions," in *ASRU*, 2019.

[9] J. Pons, S. Pascual, G. Cengarle, and J. Serrà, "Upsampling artifacts in neural audio synthesis," *arXiv preprint arXiv:2010.14356*, 2020.

[10] S. Srivastava, P. Agarwal, G. Shroff, and L. Vig, "Hierarchical capsule based neural network architecture for sequence labeling," in *IJCNN*, 2019.

[11] F. Yu, V. Koltun, and T. Funkhouser, "Dilated residual networks," in *CVPR*, 2017.

[12] L. Liu, H. Li, and M. Gruteser, "Edge assisted real-time object detection for mobile augmented reality," in *MobiCom*, 2019.

[13] R. Andri, L. Cavigelli, D. Rossi, and L. Benini, "YodaNN: An architecture for ultralow power binary-weight CNN acceleration," *TCAD*, 2017.

[14] S. Choi, J. Sim, M. Kang, and L.-S. Kim, "Trainware: A memory optimized weight update architecture for on-device convolutional neural network training," in *ISLPED*, 2018.

[15] J. Donahue and K. Simonyan, "Large scale adversarial representation learning," in *NIPS*, 2019.

[16] Z. Zhang, X. Wang, and C. Jung, "DCSR: Dilated convolutions for single image super-resolution," *IEEE Transactions on Image Processing*, 2018.

[17] D. Im, D. Han, S. Choi, S. Kang, and H.-J. Yoo, "DT-CNN: An energy-efficient dilated and transposed convolutional neural network processor for region of interest based image segmentation," *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2020.

[18] ——, "DT-CNN: Dilated and transposed convolution neural network accelerator for real-time image segmentation on mobile devices," in *High PerS*, 2019.

[19] C. Zhu, K. Huang, S. Yang, Z. Zhu, H. Zhang, and H. Shen, "An efficient hardware accelerator for structured sparse convolutional neural networks on fpgas," *arXiv preprint arXiv:2001.01955*, 2020.

[20] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, and A. Borchers, "In-datacenter performance analysis of a tensor processing unit," in *ISCA*, 2017.

[21] Y. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *JSSC*, 2017.

[22] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep learning*. MIT press Cambridge, 2016.

[23] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss V2: A flexible accelerator for emerging deep neural networks on mobile devices," *JETCAS*, 2019.

[24] S. Markidis, S. W. Der Chien, E. Laure, I. B. Peng, and J. S. Vetter, "Nvidia tensor core programmability, performance & precision," in *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2018.

[25] 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.

[26] X. Yang, M. Gao, J. Pu, A. Nayak, Q. Liu, S. E. Bell, J. O. Setter, K. Cao, H. Ha, and C. Kozyrakis, "Dnn dataflow choice is overrated," *arXiv preprint arXiv:1809.04070*, 2018.

[27] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cuDNN: Efficient primitives for deep learning," *arXiv preprint arXiv:1410.0759*, 2014.

[28] C. Vondrick, H. Pirsiavash, and A. Torralba, "Generating videos with scene dynamics," in *NIPS*, 2016.

[29] A. Yazdanbakhsh, K. Samadi, N. S. Kim, and H. Esmaeilzadeh, "GANAX: A unified MIMD-SIMD acceleration for generative adversarial networks," in *ISCA*, 2018.

[30] Y. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *High Per*, 2016.

[31] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)," in *ISSCC*, 2014.

[32] S. Rodriguez and B. Jacob, "Energy/power breakdown of pipelined nanometer caches (90nm/65nm/45nm/32nm)," in *ISLPED*, 2006.

[33] K. Chandrasekar, C. Weis, Y. Li, S. Goossens, M. Jung, O. Naji, B. Akesson, N. Wehn, and K. Goossens, "DRAMPower: Open-source DRAM power & energy estimation tool," *URL: http://www. drampower. info*, 2012.

[34] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller, "Striving for simplicity: The all convolutional net," *arXiv preprint arXiv:1412.6806*, 2014.

[35] D. Kalamkar, D. Mudigere, N. Mellempudi, D. Das, K. Banerjee, S. Avancha, D. T. Vooturi, N. Jammalamadaka, J. Huang, H. Yuen *et al.*, "A study of bfloat16 for deep learning training," *arXiv preprint arXiv:1905.12322*, 2019.

[36] S. Lym, A. Behroozi, W. Wen, G. Li, Y. Kwon, and M. Erez, "Mini-batch serialization: CNN training with inter-layer data reuse," *arXiv preprint arXiv:1810.00307*, 2018.

[37] 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.

[38] 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.

[39] 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.

[40] K. Hegde, H. Asghari-Moghaddam, M. Pellauer, N. Crago, A. Jaleel, E. Solomonik, J. Emer, and C. W. Fletcher, "Extensor: An accelerator for sparse tensor algebra," in *MICRO*, 2019.

[41] F. Chen, L. Song, H. H. Li, and Y. Chen, "ZARA: A novel zero-free dataflow accelerator for generative adversarial networks in 3D ReRAM," in *DAC*, 2019.

[42] A. Yazdanbakhsh, M. Brzozowski, B. Khaleghi, S. Ghodrati, K. Samadi, N. S. Kim, and H. Esmaeilzadeh, "FlexiGAN: An end-to-end solution for FPGA acceleration of generative adversarial networks," in *FCCM*, 2018.

[43] M. Kim, C. Park, S. Kim, T. Hong, and W. W. Ro, "Efficient dilated-winograd convolutional neural networks," in *ICIP*, 2019.

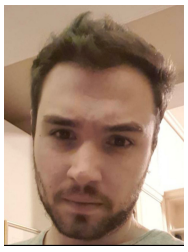
[44] J. H. Ko, B. Mudassar, T. Na, and S. Mukhopadhyay, "Design of an energy-efficient accelerator for training of convolutional neural networks using frequency-domain computation," in *DAC*, 2017.

[45] J. Zhang, F. Franchetti, and T. M. Low, "High performance zero-memory overhead direct convolutions," *arXiv preprint arXiv:1809.10170*, 2018.

Lois Orosa is the Director of the Galicia Supercomputing Center (CESGA). Previously, he was a senior researcher at SAFARI Research group @ ETH Zürich, Switzerland. He received his PhD degree from the University of Santiago de Compostela, Spain, in 2013. His current research interests are in computer architecture, hardware security, reliability, memory systems, and machine learning (ML) accelerators. For more information, please see his webpage at https://loisorosa.github.io/

Skanda Koppula is currently a research engineer at DeepMind. Previous to this, he worked at ETHZ in the SAFARI research group on memory systems, machine learning acceleration, and computer architecture. He completed his MEng and BSc from MIT in 2018.

Yaman Umuroglu received the PhD degree from the Norwegian University of Science and Technology (NTNU), Norway and a joint European MSc on Embedded Systems from the Erasmus Mundus EMECS programme. He is a research scientist at Xilinx Research Labs, Ireland. His research takes a full-stack view of machine learning with neural networks with a focus on highefficiency and high-performance implementations and spans hardware-network codesign, techniques for efficient arithmetic, sparsity, and quantization.



Konstantinos Kanellopoulos is currently pursuing his PhD at ETHZ in the SAFARI research group. He completed his MEng and BSc at NTUA. His research interests lie at the intersection of software and hardware.

Juan Gómez-Luna is a senior researcher and lecturer at SAFARI Research Group @ ETH Zürich. He received the BS and MS degrees in Telecommunication Engineering from the University of Sevilla, Spain, in 2001, and the PhD degree in Computer Science from the University of Córdoba, Spain, in 2012. Between 2005 and 2017, he was a faculty member of the University of Córdoba. His research interests focus on processing-in-memory, memory systems, heterogeneous computing, and hardware and software acceleration of medical imaging and bioinformatics. He is the lead author of PrIM (https://github.com/CMU-SAFARI/prim-benchmarks), the first publicly-available benchmark suite for a real-world processing-in-memory architecture, and Chai (https://github.com/chai-benchmarks/chai), a benchmark suite for heterogeneous systems with CPU/GPU/FPGA.

Michaela Blott received the master's degree from the University of Kaiserslautern in Germany and brings more than 25 years of computer architecture, FPGA and board design, in research institutions (ETH Zurich and Bell Labs) and development organizations. She is a distinguished engineer at Xilinx Research, Dublin, Ireland, where she heads a team of international scientists driving exciting research to define new application domains for Xilinx devices, such as machine learning. She is heavily involved with the international research community serving as the technical co-chair of FPL'2018, workshop organizer (H2RC, ITEM'2020), and member of numerous technical program committees (FPL, ISFPGA, DATE, etc.).

Kees Vissers graduated from Delft University in the Netherlands. He worked at Philips Research in Eindhoven, The Netherlands, for many years. The work included Digital Video system design, HW–SW co-design, VLIW processor design and dedicated video processors. He was a visiting industrial fellow at Carnegie Mellon University, where he worked on early High Level Synthesis tools. He was a visiting industrial fellow at UC Berkeley where he worked on several models of computation and dataflow computing. He was a director of architecture at Trimedia, and CTO at Chameleon Systems. For more than a decade he is heading a team of international researchers at Xilinx in the CTO office. The research topics include machine learning applications and architectures, wireless applications, image processing applications and new datacenter applications. These applications drive next generation programming environments and architectures. He is a Fellow at Xilinx.

Onur Mutlu is a Professor of Computer Science at ETH Zurich. He is also a faculty member at Carnegie Mellon University, where he previously held the Strecker Early Career Professorship. His research interests are in computer architecture, systems, hardware security, bioinformatics. A variety of techniques he, along with his group and collaborators, has invented over the years have influenced industry and are employed in commercial microprocessors and memory/storage systems. He started the Computer Architecture Group at Microsoft Research (2006-2009), and held various product and research positions at Intel Corporation, AMD, VMware, and Google. He is an ACM Fellow, IEEE Fellow, and an elected member of the Academy of Europe. His computer architecture and digital design course lectures are freely available (https://www.youtube.com/OnurMutluLectures). His research group makes a wide variety of software and hardware artifacts freely available online (https://safari.ethz.ch/).