MAERI: Enabling Flexible Dataflow Mapping over DNN Accelerators via Reconfigurable Interconnects

Hyoukjun Kwon
Georgia Institute of Technology
Atlanta, Georgia
hyoukjun@gatech.edu

Ananda Samajdar
Georgia Institute of Technology
Atlanta, Georgia
anandsamajdar@gatech.edu

Tushar Krishna
Georgia Institute of Technology
Atlanta, Georgia
tushar@ece.gatech.edu

Abstract

Deep neural networks (DNN) have demonstrated highly promising results across computer vision and speech recognition, and are becoming foundational for ubiquitous AI. The computational complexity of these algorithms and a need for high energy-efficiency has led to a surge in research on hardware accelerators. To reduce the latency and energy costs of accessing DRAM, most DNN accelerators are *spatial* in nature, with hundreds of processing elements (PE) operating in parallel and communicating with each other directly.

DNNs are evolving at a rapid rate, and it is common to have convolution, recurrent, pooling, and fully-connected layers with varying input and filter sizes in the most recent topologies. They may be dense or sparse. They can also be partitioned in myriad ways (within and across layers) to exploit data reuse (weights and intermediate outputs). All of the above can lead to different dataflow patterns *within* the accelerator substrate.

Unfortunately, most DNN accelerators support only fixed dataflow patterns internally as they perform a careful co-design of the PEs and the network-on-chip (NoC). In fact, the majority of them are only optimized for traffic within a convolutional layer. This makes it challenging to map arbitrary dataflows on the fabric efficiently, and can lead to underutilization of the available compute resources.

DNN accelerators need to be programmable to enable mass deployment. For them to be programmable, they need to be configurable internally to support the various dataflow patterns that could be mapped over them. To address this need, we present MAERI, which is a DNN accelerator built with a set of modular and configurable building blocks that can easily support myriad DNN partitions and mappings by appropriately configuring tiny switches. MAERI provides

DNN	Year	Num CNV	Num RNN	Num POOL	Num FC	Filter Sizes	Input Sizes
Alexnet [9]	2013	6	0	1	1	11x11, 5x5, 3x3	224x224
Googlenet [10]	2014	59	0	16	5	1x1, 3x3, 5x5	224x224
Resnet-50 [11]	2014	49	0	2	0	1x1, 3x3	224x224
VGGnet-16 [12]	2015	13	0	5	3	1x1, 3x3	224x224
DeepSpeech2 [2]	2016	2	7 (GRU)	0	1	41x11, 21x11	13x41x11
Deep voice [13]	2017	0	40	0	3	-	28x29

Table 1. Parameters of recent DNNs.

8-459% better utilization across multiple dataflow mappings over baselines with rigid NoC fabrics.

1 Introduction

The recent resurgence of the AI revolution has transpired because of synergistic advancements across big data sets, algorithms, and hardware. As a result, deep neural networks (DNNs) are being deployed at an increasing scale - across the cloud and IoT platforms - to solve complex regression and classification problems in image [1] and speech recognition [2] with accuracies surpassing those of humans.

The microarchitecture of DNN inference engines is currently an area of active research in the computer architecture community. GPUs are extremely efficient for training due to the mass parallelism they offer, multi-core CPUs continue to provide platforms for algorithmic exploration, and FPGAs provide power-efficient and configurable platforms for algorithmic exploration and acceleration; but for mass deployment across various domains (smartphones, cars, etc), specialized DNN accelerators are needed to maximize performance/watt. This observation has led to a flurry of ASIC proposals for DNN accelerators over the recent years [3–8].

This work is motivated by a key practical and open challenge for DNN accelerator ASIC design going forward: programmability. DNNs have been evolving at a massive rate. Table 1 lists some of the popular ones in use today. Within
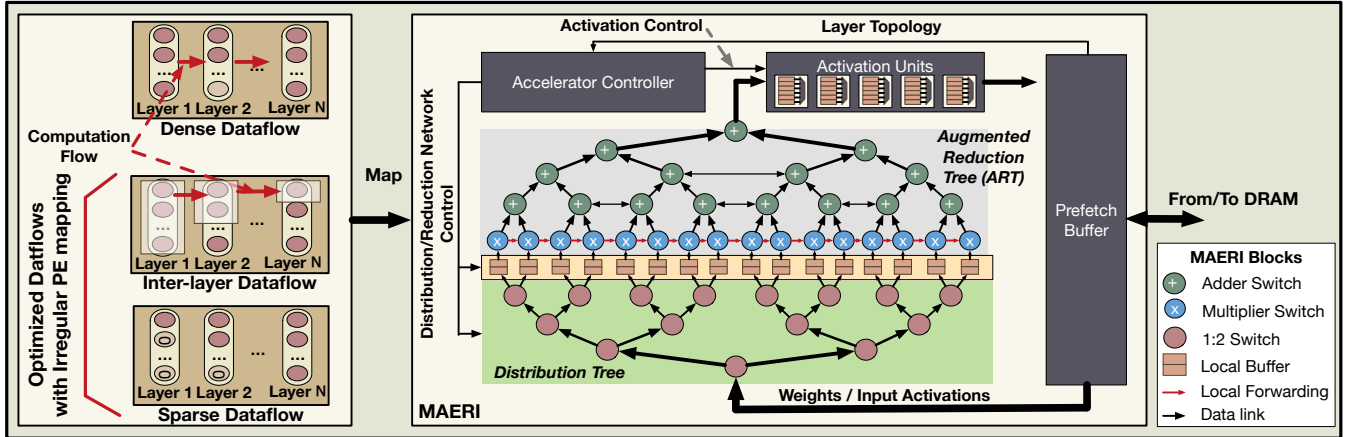
Figure 1. An overview of MAERI. MAERI is designed to efficiently handle CONV, LSTM, POOL and FC layers. It can also handle cross-layer and sparse mappings. We implement this flexibility using a novel configurable interconnection network topology within the accelerator.

this short list itself, we can observe heavy variability in terms of layer types (convolution, recurrent, pooling, and fully-connected) and input/filter sizes. Some DNNs such as GoogLeNet [10] have additional "reduction" layers to reduce image dimensions. An efficient accelerator is one that can be programmed to run all these DNNs efficiently.

We can consider programming an accelerator as a dataflow graph (DFG) partitioning and mapping problem to target accelerator substrates (analogous to the role of a compiler in programmable CPUs). The classic way of partitioning DFGs is layer by layer, and then slicing the layer into blocks that can be mapped over the underlying hardware substrate. This is the approach most of the early DNN accelerators took [6, 14, 15]. However, recently new partitioning approaches that explore additional data reuse opportunities by changing the partition have emerged - across layers [16], kernels [17] and hybrid (kernel, channel, output) [18]. Among these, for e.g., Fused CNN [16] retains intermediate outputs of a layer and calculate the next layer output using the retained outputs, not iterating over within a layer. This reduces data transfer between memory and processing elements (PEs), thereby increasing power efficiency. In addition to partitioning, the DFG can also be transformed by removing some of the edges whose weights are zero or close to zero. This is the idea of sparsity [7, 8, 19, 20] and has been popular recently to reduce power consumption inside DNN accelerators. Each of these promising approaches lead to a unique dataflow between neurons in the DFG, which in turn translates to irregular data movement patterns on-chip between the processing and memory elements. Unfortunately, most ASIC DNN accelerators proposed to date are fairly rigid in terms of their internal implementations, as the PEs and network-on-chip (NoC) are tightly coupled, as we discuss in Section 2, making it infeasible for them to support all these dataflows within the same substrate. As a result, each new optimization has resulted in a new accelerator proposal optimized for the given optimization goal [7, 8, 16, 20]. This

makes the hardening of DNN accelerators into an IP or a discrete chip impractical.

How do we design a single accelerator substrate that can handle the growing number of dataflows resulting from multiple kinds of layers, dense and sparse connections, and various partitioning approaches? Our proposed design philosophy is to make the interconnects within the accelerator reconfigurable. The DNN DFG is fundamentally a multi-dimensional multiply-accumulate calculation. Each dataflow is essentially some kind of transformation of this multi-dimensional loop [21, 22]. We propose to design DNN accelerators as a collection of multiply and adder engines, each augmented with tiny configurable switches that can be configured to support different kinds of dataflows. Our design is called MAERI (Multiply-Accumulate Engine with Reconfigurable Interconnect) [1]. MAERI can be viewed as a design methodology rather than a fixed design by itself, that makes a case for building accelerators using a suite of plug-and-play building blocks rather than as a monolithic tightly-coupled entity. These building blocks can be tuned at runtime using our novel tree-based configurable interconnection fabrics to enable efficient mapping of myriad dataflows.

We demonstrate MAERI with multiple case studies, demonstrating how it handles convolutions, recurrent layers, irregular filter sizes, and sparsity, providing 8-459% better utilization compared to a tightly-coupled rigid accelerator while adding 6.5% power overhead and reducing 36.8% area overhead over a state-of-the-art baseline like Eyeriss [6], and adding 47.0% area and increase throughput by 49.0% over a systolic array [23].

The rest of the paper is organized as follows. Section 2 provides background on DNNs and internal dataflows. Section 3 presents the MAERI design. Section 4 demonstrates various

---

[1] MAERI is a Korean word that means echo. We selected this name because a DNN accelerator's dataflow involves sending input data toward computation units and receiving a weighted-accumulation (echo) back.

dataflow mappings over MAERI. Section 5 presents implementation results. Section 6 evaluates MAERI with myriad case studies. Section 7 presents related work, and Section 8 concludes.

2 Background and Motivation

2.1 Deep Neural Networks

Neural networks are a rich class of algorithms that can be trained to model the behavior of complex mathematical functions. Neural networks models human brain with a large collection of "neurons" connected with "synapses". Each neuron is connected with many other neurons and its output enhances or inhibits the actions of the connected neurons. The connection is based on weights associated with synapses. Deep neural networks (DNNs) have multiple internal (called hidden) neuron layers before the final output layer that performs the classification.

At present, there are three popular flavors of DNNs [23]: *Convolution Neural Networks (CNN)*, *Recurrent Neural Networks (RNN)*, and *Multi-Layer Perceptrons (MLP)*. CNNs are feed-forward DNNs that have demonstrated a remarkable degree of accuracy in recognition and classification of images, exceeding human capabilities [11, 24]. Each convolution layer receives input in form of a raw image or input activations (the output of a previous convolution layer) and produces output activations by sliding a set of filters over input images [25]. Convolutions account for almost 90% of the computations in a CNN [26]. RNNs add feed-back connections inside the neural network to reflect past context during the feed-forward computation and are used for tasks involving signals with temporal correlation, such as speech recognition, transliteration and so on. The most popular RNN is long short-term memory (LSTM) [27], where the output at a certain time depends on the current input value, three gate values (forget, input, and output), and one state value. RNNs account for 29% of Google's inference traffic [23].

All DNNs typically use one or more *fully-connected (FC)* layers at the end to perform the actual data classification. MLPs are built using multiple fully-connected (FC) layers and are the most general forms of DNNs. DNNs also employ other layers such as *pooling (POOL)* to reduce the outputs of multiple neurons into one. Table 1 lists some of the most popular DNNs in use today.

2.2 Dataflows in a DNN

In this work, we define "dataflow" as the data communication pattern within a DNN accelerators between the compute and memory elements. Dataflow affects the data reuse pattern, which is critical to throughput and energy efficiency of the accelerator, and has thus been the subject of active research recently. We identify three factors that affect the dataflow.
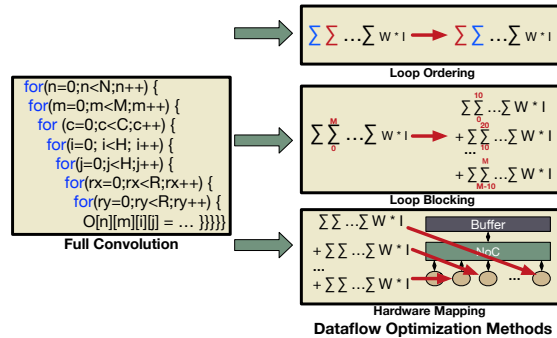

Figure 2. Techniques to optimize dataflows in DNN accelerators

Dataflows due to DNN Topology. DNN topologies have been evolving at timescales of a few months due to the proliferation of deep learning frameworks [28–31] and the availability of massive datasets. The DNNs listed in Table 1 can beat humans at vision/speech level tasks. The most accurate DNNs employ a multitude of layers, and input/filter sizes. Naturally, these lead to different dataflow patterns based on the type of layer, and the size, as Section 2.1 alluded to.

Dataflows due to DNN Topology Mapping. The DNN DFG can be partitioned in multitude ways in computation order, partitioning size, and hardware mapping of each partition, as Figure 2 shows. For example, Fused-layer [16] modified the computation order (loop order) in CNNs to maximize data reuse that minimizes energy-consuming global communication between the PE array and the prefetch buffer. Ma *et al.* [32] leverage loop blocking, unrolling, ordering to minimize storage in DRAM. All these loop optimizations change the dataflow within an accelerator fabric. For instance, a loop ordering optimization in a CNN like AlexNet [9] across the first three layers would lead to a 11x11 filter, 5x5 filter, and 3x3 filter being simultaneously mapped over an accelerator array which is quite difficult to achieve in most accelerator implementations, as Section 2.3 discusses.

Dataflows due to DFG optimizations (e.g., pruning zero edges; sparsity) The DNN DFG can also be changed by removing redundant parts, especially zero weights or inputs, because a multiplication with zero is always zero, and does not affect the result of the accumulation. EIE [7] reported that the percentage of zeros in weights from practical benchmarks ranges from 4% to 25%, which implies that we can reduce the number of computation from 4% to 25%. This in turn has led to accelerator implementations optimized for handling the unique dataflows stemming from sparsity [7, 8, 20].

In summary, various DNN topology and DFG transformations/pruning lead to different dataflow patterns. In particular, to exploit the benefits of DFS optimization and pruning, hardware support is essential. Therefore, we need DNN accelerator designs that can efficiently handle myriad dataflows.

2.3 Challenges with supporting flexible dataflows

The big challenge with DNN accelerator ASICs today is that the design process tightly couples PEs and the NoC. A generic

Distribute	Sending input activations and filter weights for each neuron to the various PEs from the PB. Distributes can be unicasts, multicasts or broadcasts, depending on the dataflow mapping.
Compute	The *multiplication* of input activations with the weight to generate a partial sum (psum) in each PE.
Reduce	*Adding* the partial sums generated by PEs to generate the output activation of each neuron.
Collect	Transferring the final output activations from multiple PEs to the PB, to be used as inputs to the next layer. Collect can also be used to store psums in case the entire neuron does not fit over the PE array.

Table 2. Taxonomy of On-Chip Communication Flows.

all-to-all NoC like a crossbar or a mesh is extremely area and power inefficient [6, 33] for an array of 100s of tiny PEs, and as a result almost every DNN accelerator has used a hierarchy of buses [6, 14, 34] and/or trees [34, 35]. For instance, Eyeriss [6] uses buses to connect 12 PEs together in a row, and 14 rows are connected together by another bus. SCNN [8] creates clusters 4x4 PEs with adder trees internally, and an external bus connecting to on-chip SRAM. The size of each cluster is often determined by the nominal size of the filters (3×3 or 4×4), and the buses/trees optimized for data distribution and collection from these. This rigid structure restricts arbitrary filter sizes or cross-layer dataflows from being supported in these designs. Moreover, when optimizations such as sparsity are introduced, the filter sizes can vary dramatically while the size of the PE clusters is fixed, leading to inefficient utilization as we demonstrate in this work.

FPGAs have been popular substrates to evaluate various dataflows [35–37] as they provide the flexibility of running different dataflows based on their reconfigurable substrate. A recent work explored FPGA design optimization for DNN [35] demonstrates that the nominal tiling factor for CNN computation can vary layer by layer and generates optimized RTL for every layer. Fused CNN [36] augments this approach to support cross-layer dataflows on FPGAs. However, while FPGAs vs. ASICs for DNNs is an ongoing debate, with the flexibility being touted as a reason to prefer the former, the area, timing, and power-efficiency of ASICs is still the strong case for ASICs.

The goal of this work is to provide the flexibility afforded by FPGAs today in terms of flexible dataflow support to ASIC accelerators. Our key idea is to use a homogeneous, rather than hierarchical, design and provide flexibility within the NoC topology to create virtualized clusters of arbitrary sizes at runtime.

2.4 Taxonomy of communication flows

Without loss of generality, we assume a DNN accelerator implementation comprising of an on-chip memory buffer (that we call a prefetch buffer (PB)) to prefetch inputs/weights/intermediate values from DRAM), and a multitude of PEs. Any dataflow pattern mapped over these PEs can essentially be
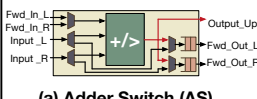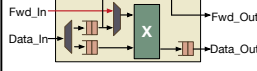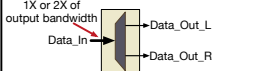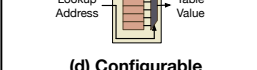

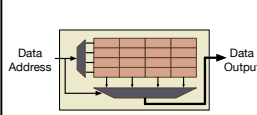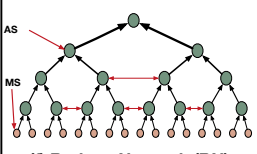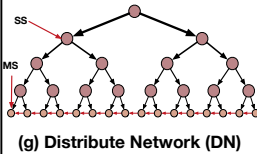| Building Block	Description
(a) Adder Switch (AS)	The AS is an adder augmented with a tiny switch that enables us to map arbitrary adder trees over our non-blocking reduce network called ART. It also contains comparators for pooling operations.
(b) Multiplier Switch (MS)	The MS is a multiplier augmented with a tiny switch that is used for local data forwarding. It is used by CNNs for generating partial sums from weights and input activation values, and by RNNs for generating gate values, input activations, and previous output activations.
(c) Simple Switch (SS)	The SS provides 1:2 switching functionality in the distribute network's chubby tree nodes.
(d) Configurable Look-up Table (LT)	LTs implement activation functions such as *sigmoid* or *tanh*. We load all the necessary functions for a neural network and change its target function in run time based on the configuration generated by *MAERI*.
(e) Prefetch Buffer (PB)	The PB works like a cache memory between DRAM and the computation units. We implement this as a private scratchpad. Because the characteristics of a prefetch buffer would differ based on SRAM technology library and they are usually commercial libraries, we provide a default multi-banked implementation using flip-flops.
(f) Reduce Network (RN)	The RN is a network structure for reduce and collection operations. It is based on a new adder tree structure, augmented reduction tree (ART) we propose in this paper. ART facilitates mapping multiple configurable non-blocking adder trees and minimizing inactive multiplier switches.
(g) Distribute Network (DN)	The DN is based on a chubby-tree structure, which is a tree-based network with wider link bandwidth in higher levels of a tree. We exploit abandunt bandwidth at high level links to enable multicast functionality, which is one of the most common traffic patterns in DNN accelerators.

Figure 3. The microarchitecture of building blocks used in MAERI and description of them.

abstracted to perform one of four operations listed in Table 2. We use this taxonomy throughout this work.

3 MAERI Building Blocks

Figure 1 provides an overview of the MAERI microarchitecture. We use a homogeneous design with plug-and-play building blocks that are listed in Figure 3. A prefetch buffer (PB) serves as a cache of DRAM, and stores input activations, weights, intermediate partial sums that could not be fully accumulated, and output activations. Lookup Tables (LTs) implementing activation functions are located between the root of the reduction-tree and the PB. The secret sauce that enables our flexibility is two-fold:

(i) We augment the multipliers and adders with configurable switches, calling them *multiplier switch (MS)* and *adder switch (AS)* respectively, enabling MAERI to optimize for the collective communication patterns.

(ii) We use two configurable interconnection networks - a distribution network, built using tiny *simple switches (SS)* sends inputs to the MSes from the PB. A reduce+collect network, built using ASes, sends outputs back to the PB via activation units implemented with look-up tables.

The entire accelerator is controlled by a programmable controller which manages reconfiguration of all three sets of switches (MS, AS, and SS) for mapping the target dataflow.

We design two novel interconnect topologies specialized for the distribution and reduction+collection flows, which we describe next. These networks can be tuned to provide full non-blocking bandwidth to the compute blocks, but can be pruned to reduce the bandwidth if required.

3.1 Data Distribution Network

The data distribution network of MAERI sends input activations and weights from the PB to the MSes. It ensures full non-blocking bandwidth to all the multipliers.

3.1.1 Topology

We use a binary-tree as our base topology for distributions as it is multicast friendly, and augment it with (a) chubby links for supporting higher-bandwidth from the PB, and (b) forwarding links, to implementing store-and-forward multicasts for CNNs. We describe these next.

Chubby Links from Root. A fat-tree supports 2x bandwidth at every level from the leaves up to the root, and is a classic topology for providing non-blocking bandwidth, and is used extensively in datacenter networks [38]. However, such a topology is infeasible to build on-chip since the bandwidth requirement at the root would require too many wires and ports at the PB, resulting in significant area and power overheads[2]. To address this design-challenge, we propose a **Chubby tree**, where the bandwidth at the upper levels is either 2x or the same as that at the lower levels. For example, in Figure 4, the bandwidth of link level 0 is twice of that of link level 1, but the bandwidth of link level 2 and 3 is 1x.

We size the bandwidth at the root to equal that of the PB, and taper the width going down the tree, providing non-blocking flows till that level. Once bandwidth becomes 1x, the bandwidth for the rest of the levels are compensated for by adding local buffers at the MSes to hide the communication delay due to multiplexing of links at the 1x levels.

Forwarding Links at Leaves. MAERI provides local data forwarding links (FL) between the leaves (i.e., adjacent MSes) in the distribution network, as shown in Figure 5 to provide store-and-forward multicast for input activation values. We
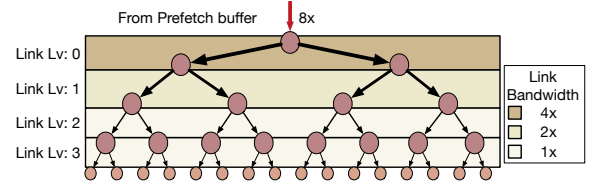
[2]For a fat-tree network, 256 MSes would require a 256 ported SRAM.

Figure 4. Example of chubby distribution tree. Leaves are multiplier switches. Other nodes are simple switches without compute units.
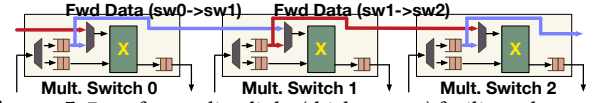

Figure 5. Data forwarding links (thick arrows) facilitate data reuse between adjacent multiplier switches.

highlight their use for CNNs in Section 4. The FL's are unidirectional since MAERI maps data over the multiplier switches in order, so input activation values flow in one direction.

3.1.2 Microarchitecture, Routing and Flow-Control

The distribution tree nodes use a *simple switch (SS)* whose microarchitecture is shown in Figure 3. SSes are bufferless demuxes with a select line that is set by the input data directly. SSes to chubby links are direct connections, since no bandwidth sharing is being done.

Since the topology is binary-tree based, input data is source routed, with a bit to choose between the left and right paths at each switch. Since the SSes are bufferless, the flow-control is end to end between FIFOs at the MSes and the PB. Also, *we provide single-cycle traversals from the PB to the leaves (MS)* for every piece of data.

3.2 Data Reduction and Collection Network: ART

Binary trees are well-suited for performing reductions and have been used in prior DNN implementations [3, 4, 17, 19, 21] to implement adder trees within the PE clusters described in Section 2.3. However, they have a key inefficiency: the fixed topology of a tree is inherently inefficient whenever the number of partial sums to be accumulated is smaller than the width of the adder tree. Figure 6(a) illustrates this. Suppose there are 16 multipliers, all connected via a 16-node binary reduction tree. Each node in the reduction tree is an adder. This tree is perfect for performing a reduction for a 16-input neuron. However, suppose we map three neurons over these multipliers, each generating five partial sums, as Figure 6(a) shows. Each neuron requires four additions to generate an output, so the total additions required is 12. The reduction tree contains 16 adders, which should be sufficient to perform the additions for all neurons in parallel. However, the four links in red are shared by multiple neurons, limiting the tree from generating all three outputs simultaneously.

More formally, the challenge pertains to mapping arbitrary number of reduction trees over an underlying fixed topology. To provide flexibility to support any mapping, this needs to be addressed, otherwise there would be a drop in utilization.
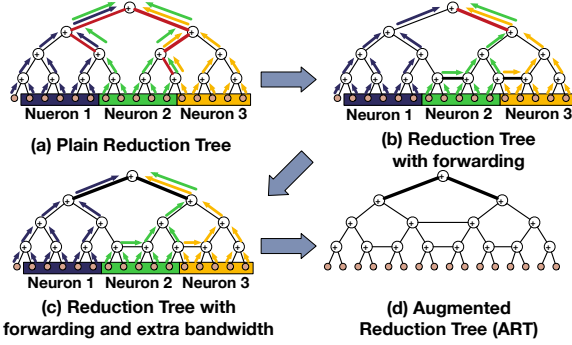
Figure 6. Motivation for Augmented Reduction Tree (ART). Three neurons are mapped over five multipliers each. Each neuron generates one output using the adder tree for reduction. Red links in (a) and (b) represent congested links, thick links in (c) and (d) represent links with double bandwidth. The forward links (FL) in the ART are bi-directional.

We solve this challenge by proposing a new tree topology called an Augmented Reduction Tree (ART).

3.2.1 Topology of ART

The ART is a binary-tree augmented with additional links to enable multiple arbitrary sized reductions to run in a non-blocking manner.

Forwarding Links at Intermediate Levels. Figure 6(b) shows a binary tree with additional links for forwarding the adder outputs to other adders at the same level, instead of to the parent. This removes contention from three out of the four links, and only one link is shared by Neuron 2 and Neuron 3.

Chubby Links from Root. Physical bandwidth limitations at the root node can limit the number of parallel reductions (collects) as Figure 6(b) showed. To address this, we augment the ART with chubby links, just like the one in Section 3.1. This eliminates contention completely, as shown in Figure 6(c).

3.2.2 Formal Definition and Properties of ART

Figure 6(d) presents an example of an ART. We formally define it and present two key properties.

Definition. *Augmented Reduction Tree is an undirected graph that consists of a complete binary tree and additional links that connects adjacent tree nodes in the same level with different parents except between leaves.*

Property 1: Configurability. *An ART with N leaves can map any adder tree onto its substructure when the adder tree accumulates values from k consecutive leaves and k < N.*

Property 2: Non-Blocking. *An ART can map multiple of such adder trees mentioned in (1) without any sharing any link if the sets of leaves of each adder tree are all disjoint.*

Property 1 guarantees any sized reduction operation can be mapped over an ART. Property 2 guarantees that multiple non-blocking reduction operations can be mapped over an ART. For example, if an ART has 32 leaves and we map
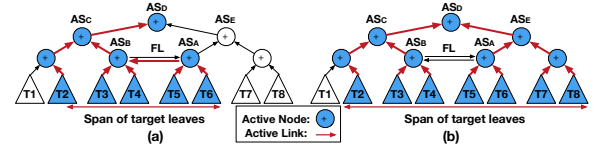

Figure 7. Two examples of forwarding link reconfiguration.

multiple adder trees that accumulates five values, ART can accommodate four adder trees at the same time providing non-blocking reduction.

We have formally proved these two properties. However, in the interest of space, we provide an intuition on why the ART supports multiple parallel reductions, and will provide the formal proof as supplementary material once MAERI is released. Intuitively, additional connectivity between adjacent adder switches on the same level, but with different parents, provides opportunity to accumulate more partial sums closer to the lower levels which reduces the number of required links in upper levels. Without the additional connectivity, partial sums from the adjacent nodes have to traverse upper levels to be accumulated, increasing the possibility of link congestion. We do not add forwarding links (FLs) between nodes sharing the same parent node because the parent node anyway needs to be traversed to reach the top.

3.2.3 Microarchitecture, Routing, and Flow-Control

The ART is built using *adder switches (AS)*, whose microarchitecture is shown in Figure 3. Each AS is *statically configured* to act as either 2:1 ADD, 3:1 ADD, 1:1 ADD plus 1:1 forward, or 2:2 forward. Section 4.1 describes the configuration algorithm. Each AS also houses a comparator for POOL layers.

4 Mapping Dataflows over MAERI

Our dataflow mapping is mapping neurons one by one over the MSes. We call this Virtual Neuron (VN) Construction. It essentially means configuring the ART, since that is the one that decides which multiplier outputs need to be reduced.

Once the ART is configured, each VN can operate in parallel. The dataflow flexibility in our design comes from allowing each VN, which is a MAC operation, to take arbitrary number of MSes (rather than relying on fixed clusters). We also support folding of a VN over itself and multiplex multiple multiply operations over fewer MSes (Section 4.8).

4.1 Virtual Neuron (VN) Construction over ART

We describe the algorithm for VN construction over the ART for the example in Figure 7(a). Each triangle represents a sub-tree, and each circle is an AS. We focus on the FL marked in the figure. Here, the VN spans from T2 to T6 (Figure 7(a)).

Step 1: Compute the span of the neuron on the left and right side of the FL. Set the direction of the FL from the smaller to the larger span. We define span to be the number of sub-trees that the VN (generating the psums) crosses. In Figure 7 (a), the VN spans from T2 to T4 (i.e., three) on the left, and from T5 to T6 (i.e., two) on the right.

The direction of the FL is set from right to left. If the spans are same, then the direction can be set arbitrarily.

Step 2: Check if the sub-trees in direction of the smaller span need to use the parent to this FL on that direction. If not, activate this FL. The parent of FL on the right side (i.e., P_R) does not need to be activated for this neuron, since T7 and T8 are not part of the span. Thus the FL is activated by configuring AS_A to forward the output from T5 to AS_B, and AS_B to act as a 3:1 adder.

Figure 7 (b) shows an alternate scenario. Here, Step 1 determines the direction to be left to right. However, on the left, the AS_C has to be activated regardless of the FL, because T2's output goes to it, hence FL does not need to be activated. AS_B is configured to forward its output to AS_C, not AS_A.

How is the span computed by the algorithm? We represent the leaves (MSes) spanning each neuron using a bit-vector. The ART controller starts from FLs in the lowest level to activate FLs, before going up the levels. The number of bits that are 1 on the left and right of this FL in the bit-vector are used to compute the span. Whenever a FL is activated, the bits corresponding to smaller span (i.e., the leaves that will create the psums that will cross this FL) are cleared. This prevents activating multiple FLs in different levels of the ART for the same partial sums.

Next, we demonstrate how example DNN dataflows can be mapped over MAERI.

4.2 Mapping a CONV Layer

We demonstrate how a CONV layer can be mapped over MAERI with a walk-through example in Figure 8. The weight filter is 2x2, and the input/output activations with one channel are 4x4. This example assumes that each MS stores one input activation and one filter weight value locally and the chubby ART, together with the PB, provides sufficient bandwidth to cover all simultaneous reduction flows.

Stage 1: VN Construction. MAERI first constructs a VN by configuring the ART based on the dimension of the target CNN layer, as Figure 8-(1) shows. The controller then maps the filter weights to a set of consecutive multiplication switches in each VN, and configures the corresponding sub adder tree of the ART using the reconfiguration algorithm described earlier in Section 4.1. Each VN is responsible for generating one row of output activations, and two VNs can share one AS. In the example, VN 0 and 1 share the AS marked in purple. Although an AS (or a node of the ART) is shared, the overall structure still maintains the non-blocking feature since one of the inputs is sent up the tree, and the other is sent laterally using the forwarding link simultaneously, as Section 4.1 described.

This configuration step happens before running each CNN layer and remains constant throughout the run. In case the layer does not fit, it can be folded and mapped multiple times as we explain later in this section.
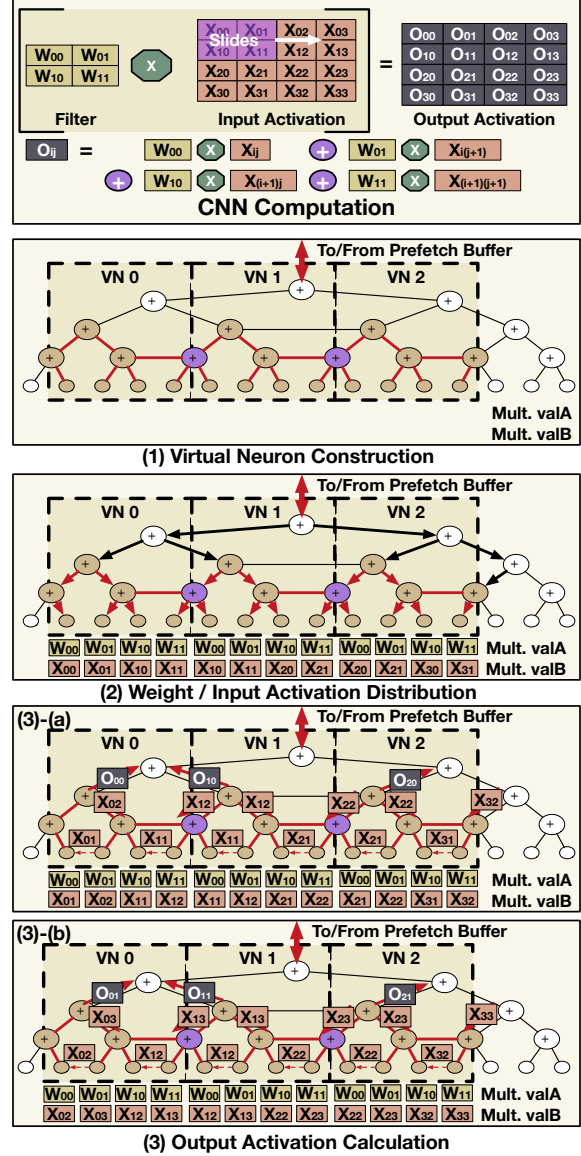
Figure 8. CONV computation in MAERI. W, X, and O represent weights, input activations, and output activation, respectively. The indices of W, X, and O represent the position of each value within each matrix.

Stage 2.1: Weight Distribution. Next, MAERI starts to distribute filter weights, as Figure 8-(2) shows. Recall that in CNNs, the weight matrix slides over input images (Section 2.1); as a result the same weight value is required by multiple VNs, each of which is computing an output activation. We exploit the multicast functionality of the distribution tree, by sending one value from the PB and replicating it at the intermediate simple-switches. For example, weight W_{00}, W_{01}, W_{10}, and W_{11} are sent to the first, the second, the third, and the fourth multiplier switch in each VN, respectively. We can exploit the bandwidth of the chubby tree structure to deliver multiple unique weights simultaneously to different multiplier switches. Because each VN requires the same set

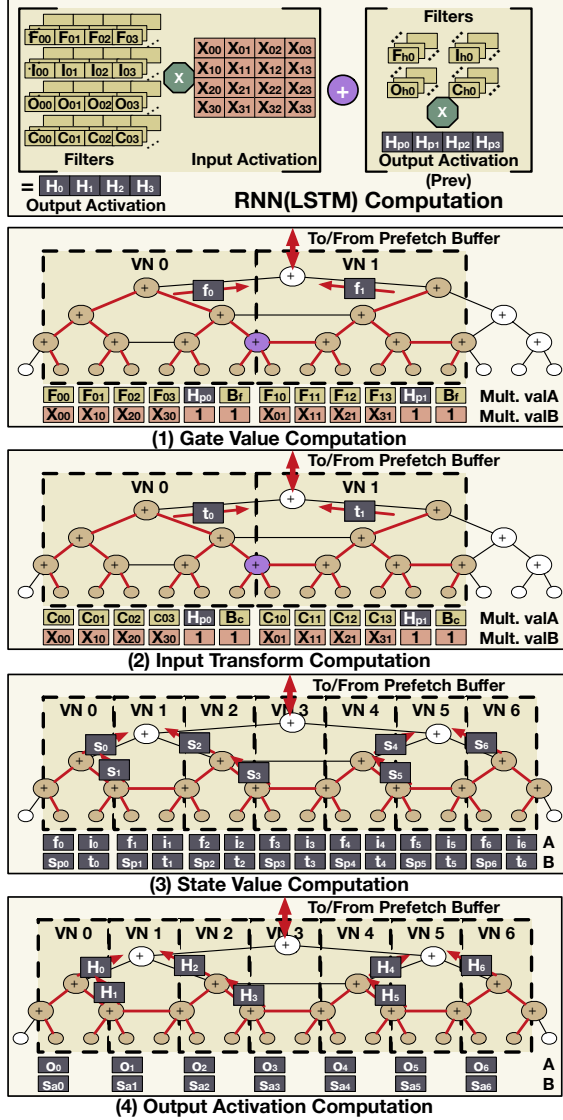**Figure 9.** LSTM RNN computation in Maeri. **F**, **I**, **O**, and **C** indicate weights for forget/intput/output gated and input transform multiplied with input activations. $\mathbf{F}_h$, $\mathbf{I}_h$, $\mathbf{O}_h$, and $\mathbf{C}_h$ represent weights for forget/intput/output gated and input transform multiplied with previous output activations. **X** and **H** represent input and output activations. The indices of **F**, **I**, **O**, and **C** indicate the ID of corresponding neuron and position within the weight vector (e.g., $\mathbf{F}_{30}$ indicates the first forget gate filter weight value for neuron 4.) The index of $\mathbf{F}_h$, $\mathbf{I}_h$, $\mathbf{O}_h$, and $\mathbf{C}_h$ means its corresponding neuron ID (e.g., $\mathbf{C}_{h3}$ represents the filter weight value to be multiplied with the previous output activation in neuron 4). The four steps presented generate an output activation for each VN. $\mathbf{f}_k$, $\mathbf{i}_k$, $\mathbf{o}_k$, and $\mathbf{t}_k$ represent forget/input/output gate values and input transform at the current time epoch. $\mathbf{B}_f$, $\mathbf{B}_i$, $\mathbf{B}_o$, and $\mathbf{B}_c$ are bias values for each gate value and input transform. $\mathbf{s}_k$ and $\mathbf{s}_{pk}$ are the state values for the current and previous epoch, respectively.

of weight values, the PB distributes weights only once for every CONV layer, which remain stationary throughout the run of the layer.

Stage 2.2: Input Activation Distribution. The same input activations are used by multiple neurons, just like the weights. For example, both VN 0 and 1 require input activations X_{10} and X_{11}, and both VN 1 and 2 require X_{20} and X_{21}. These are multicasted from the PB. Unlike weight distribution, the distribution of new input activations from the PB needs to be performed whenever each VN has finished computing all psums for one output activation. This part is overlapped (pipelined) with the generation of output activations (Stage 3). For instance, in Figure 8-(3a), we can see that new input activations X_{02} and X_{12} arrive at VN 0, X_{12} and X_{22} arrive at VN 1, and so on, while it is computing O_{00}.

We model the sliding window behavior of the CNN filter weight matrix by using the local forwarding links between the multiplier switches. Each input activation is forwarded left up to the width of the filter row (which is two in this example) and then discarded. For instance, in Figure 8-(3a), we can see that new input activations X_{01} and X_{11} are forwarded from the second and fourth MSes to the first and third respectively. Because of the data forwarding, each VN requires only two new input activation values (same as row size of the filter) from the PB every cycle, for generating a new output activation. This reduces the bandwidth requirement of the distribution tree, and the overall energy consumption by reducing the interconnect traversal length.

Stage 3: Output Generation. After the series of initialization steps (VN construction and weight/input activation distribution) finishes, Maeri starts to produce output activation values. Each VN generates output activation values for one row of the output matrix.For the CNN computation example in Figure 8, Figure 8-(3) shows VN 0 producing O_{00}, followed by O_{01}, and so on. Similarly, VN 1 generates O_{10}, O_{11}, O_{12}, and O_{13}. After finishing one row, input activations corresponding to another row are mapped on the VN, and so on till the end of the current convolution layer.

Optimizing for Spatial Reuse in CNNs. Maeri tries to optimize and get the best of the three kinds of dataflows described in the Eyeriss [25] taxonomy. Each multiplier switch acts as a *weight stationary* node without requiring weights to be forwarded back and forth. Each row of the weight filter is mapped sequentially across the multipliers of a VN making it *row stationary*. And finally, the configurable ART within each VN acts like an *output stationary* node as it accumulates psums locally. Together, we get a design optimized for high-throughput and low-energy.

4.3 Mapping a RNN/LSTM Layer

Figure 9 shows how Maeri runs a LSTM layer. A LSTM computation consists of four steps: calculating (1) gate values, (2) input transform, (3) next state value using the results from step 1 and 2, and (4) output activation using the results from step 1 to 3.
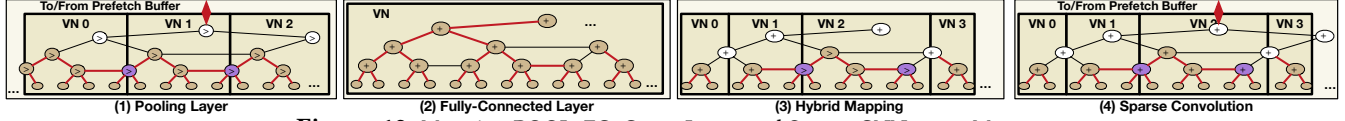
Figure 10. Mapping POOL, FC, Cross-Layer and Sparse CNNs over Maeri.

Gate values and input transform calculation. For step
1 and 2, Maeri first constructs VNs. Maeri distributes input
activations and weights, and performs the same multiply-
accumulate computation as the CONV example. However,
unlike the CONV case, it iterates four weight filters (for-
get/input/output gate and input transform) and reuses input
activation values for each gate value and input transform
computation. Step 1 and 2 require the same number of MSes
within a neuron and share the same input activation data set;
thus we merge them in Maeri. In other words, when each
VN receives input activations, it reuses them to calculate
all the gate values (step 1) and input transforms (step 2) for
the received input activation, before the PB distributes the
next round of input activation. The computed gate values
are collected by the PB (over the ART) and stored.

State value and output activation calculation. After the
completion of step 1 and 2 over target input activations (**X**),
Maeri reconstructs VNs to calculate state and output acti-
vation values, as Figure 9-(3) and (4) show. The reason for
reconstructing VNs is because the calculations in step 3 and
4 requires fewer number of multiplier switches; Retaining
the same VN configurations as step 1 and 2 would lead to
underutilization of the available multipliers. However, some-
times reconstructing VNs between step 3 and 4 may not help
if the VNs are too fine-grained (say each VN just has two
multiplier switches), since the ART might not have enough
bandwidth to compute and transmit all output activations
to the root every cycle. Our optimization tool considers this
aspect and generates appropriate parameters for the ART
controller so Maeri can prevents such a contention scenario.

Step 3 and 4 also calculates partial sums and accumulates
them like all the other CNN/RNN computation steps. For
state value computation (step 3), each VN receives the previ-
ous state value calculated in the previous time epoch from
the PB using the distribution tree, and the forget/input gate
values and input transform calculated in step 1 and 2 in the
current time epoch. It then generates the current state values
based on the received values, as Figure 9-(3) shows. For the
output activation computation (step 4), each VN receives the
output gate value and current state value, multiplies the two,
and sends the result over to the activation units to produce
the final output activation.

4.4 Mapping a POOL Layer

Mapping a POOL layer over Maeri requires creating a VN
with with the values to be pooled, and configuring the AS to
act as a comparator, rather than an adder. The output of the
ART is then the pooled value.

4.5 Mapping a FC Layer

A FC neuron gets inputs from all neurons in the previous
layer. Correspondingly, the VN for this can be mapped as
before, except that it would span many more MSes. In the
extreme case, the entire ART can be configured to compute
the output for one neuron, as Figure 10 shows. In case one
neuron does not fit over the free MSes (either because the
number of inputs is greater than the total MSes, or some of
the MSes are already configured into other VNs), the neuron
can be mapped via folding, as Section 4.8 discusses.

4.6 Mapping Cross-Layers

A cross-layer mapping [35–37] can easily be supported over
Maeri since each VN can be independently configured. Thus
each VN could correspond to neurons of the same layer (as
Figure 8) illustrated, or different layers (Figure 10), without
requiring any change in the algorithm. In the latter case, the
intermediate output from the PB need not be sent to DRAM,
but can be streamed to the next VN on-chip directly.

4.7 Mapping Sparse Networks

Mapping sparse CNNs is also quite trivial in Maeri, as Fig-
ure 10 shows. The size of each VN would be different size
since the filter sizes vary depending on weight sparsity. Note
that Maeri can support mapping of sparse CNNs but might
still need additional support to identify sparsity and stored
compressed data [7, 8].

4.8 Optimization: Folding over Rows

There can be multiple reasons to fold a physical neuron over
fewer MSes than its inputs, such as: (i) there are insufficient
MSes, or (ii) the bandwidth of the PB or the chubby distribu-
tion tree is insufficient to cover all the input activation values
during output activation calculation, (iii) the VN is memory
bound and waiting for data from DRAM. To support N-way
folding, MSes need to have at least N local buffers. This
increases their area and power, but lowers the bandwidth
requirement from the network.

For example, in Figure 8, we can allocate two multiplier
switches rather than four within a virtual neuron. In such
a mapping, the output of the VN through the ART is an
intermediate psum (one row of the filter in this example).
This is sent to the PB for temporary storage and then sent
back to the corresponding VN to be accumulated into the
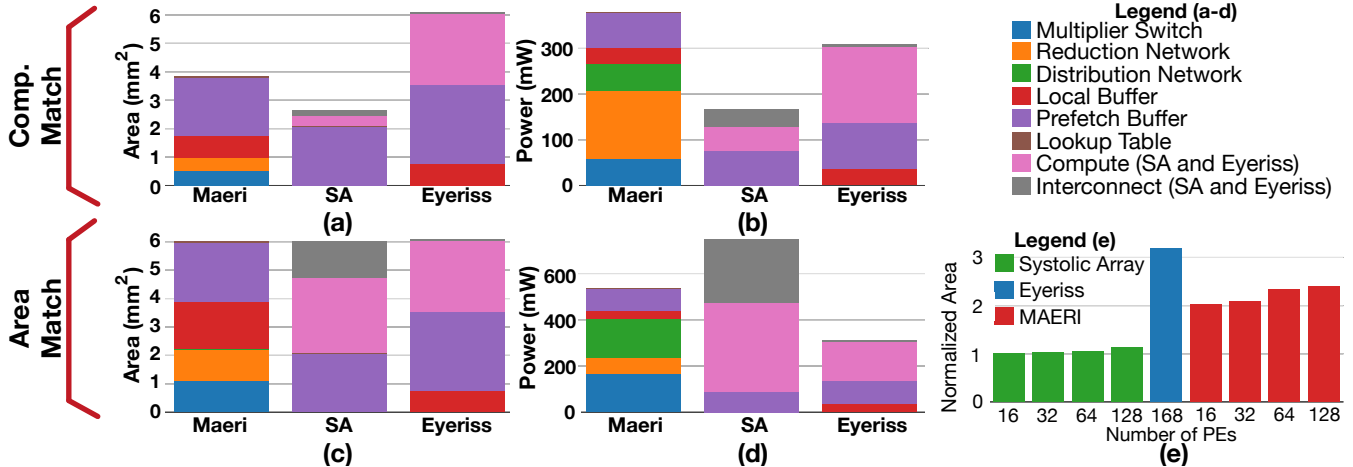final output activation that is sent to the activation units.

Figure 11. Area and power breakdown of MAERI, systolic array and Eyeriss. Comp match (a,b) and area match (c,d) indicate design points with the same number of compute units and area as Eyeriss (Table 3). The left and right column plots area (a, c) and power (b, d), respectively. (e) plots the post place-and-routed area of MAERI, systolic array and Eyeriss, normalized to the 16 PE systolic array.

Design	Eyeriss	SysArray (Comp)	SysArray (Area)	MAERI (Comp)	MAERI (Area)
Technology	28nm	28nm	28nm	28nm	28nm
Number of PEs (MultSwitches)	168	168	1192	168	374
Local SRAM/PE	512B	0	0	512B	512B
Prefetch Buffer	108 KB	80KB	80KB	80KB	80KB
Area	$6mm^2$	$2.62mm^2$	$6mm^2$	$3.84mm^2$	$6mm^2$

Table 3. MAERI implementation details and comparison with Eyeriss and Systolic Array. SysArray/MAERI (Comp) and SysArray/-MAERI (Area) are Systolic array/MAERI implementations that have the same number of compute units and area as Eyeriss respectively.

5 Implementation

We implemented MAERI in BSV (Bluespec System Verilog) [39] and synthesized it with TSMC 28nm standard cell and SRAM library at 200MHz. For comparison, we also synthesized and placed-and-routed Eyeriss [6][3] and a systolic array [23] in our 28nm environment. We created two design points - one where all three accelerators have the same number of compute (i.e., multiply-accumulate or MAC) units and SRAM size, and one where all three have the same chip area, as Table 3 shows. We find MAERI to be more area-efficient than Eyeriss for two reasons: (i) its fine-grained MS and AS together are much more area-efficient than a full PE, and (ii) MAERI does not require fully-addressable local register file like Eyeriss; it uses FIFOs instead and relies on delivery of the correct data in the correct order via the distribution tree or local forwarding links. For the same area, MAERI and systolic array can house 206 (2.23 ×) and 1024 (7.09 ×) more compute units than Eyeriss.

Because of the larger number of compute units and MAERI's near 100% utilization based on its fully non-blocking trees, synthesis tools report higher power in MAERI than Eyeriss. Thinning the adder tree links can bring us to the same power point, while still offering full configurability but at a loss of

---

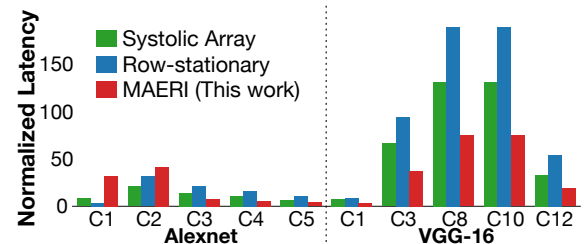[3]We thank the authors of Eyeriss for sharing their RTL with us.

Figure 12. Total latency and compute unit utilization of systolic array(SA), Eyeriss [25] style row-stationary accelerator, and MAERI with 64 PEs (multiplier switches) for selected conv layers in Alexnet and VGG16. The latency is normalized to the first Alexnet convolutional layer delay in an ideal accelerator with 64 PEs, infinite bandwidth between all the PEs and the PB, and 1-cycle fixed point computational units.
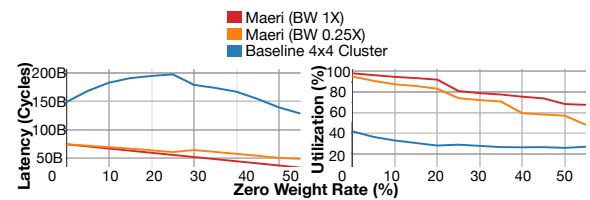

Figure 13. Total latency for VGG16 convolutional layer 8 (C8 in Figure 12) for sparse workload. MAERI includes 64 MSes. The baseline uses four 4x4 PE clusters connected by buses.

full utilization every cycle, which might be an acceptable trade-off as a lot of DNN layers are memory-bound [23]. The prefetch buffer (SRAM) dominates in both area and power in the two designs. For the same number of PEs (compute units), the systolic array required the smallest area and power because of its simple structure (MACs connected in a grid). This is also illustrated in Figure 11(e). However, systolic array suffers from low utilization [23] and requires a large number of SRAM reads because of the lack of data reuse inside a PE array as we demonstrate later in Section 6.3.
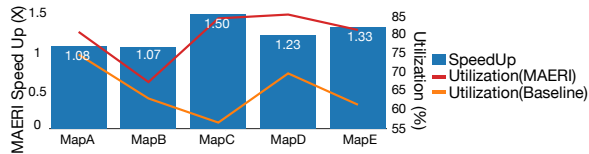
Figure 14. Speed up of MAERI over a 64 PE baseline (four 4x4 clusters) with hybrid cross-layer dataflow. MapA-E are Alexnet conv1+2+3, 2+3+4, 3+4+5, 1+2+3+4, and 2+3+4+5 respectively.

6 Evaluations

6.1 Performance with regular (dense) dataflow.

Given the same number of compute units, the performance of a spatial accelerator depends on both the utilization of the compute units, and the internal dataflow (which determines reuse and activity). We plot the total latency for running selected convolutional layers in AlexNet and VGGnet in Figure 12 across the following designs: MAERI, Systolic Array and Row-Stationary (i.e., based on Eyeriss [25]) with 64 multipliers/MACs/PEs respectively. MAERI provides a speedup of 72.4% on average across all layers. We observed 95% utilization in average across the multipliers in MAERI. We can see that large filter sizes, such as AlexNet's C1 (11×11; requires temporal folding) and C2 (5×5) layers are adversarial for MAERI as they lead to large VNs, leading to underutilization of the remainder MSes. But recent CNNs like VGG-16 with 3×3 filters provide the best utilization by allowing seven filters to be mapped simultaneously over the 64 MSes with only one MS idle.

6.2 Performance with irregular dataflow.

Sparse Dataflow. Figure 13 represents the total latency of VGG16 convolutional layer 8 with varying percentage of zero weights executed on MAERI with 64 multiplier switches and different chubby tree bandwidths (1X and 0.25X represent the bandwidth at the root of the tree, i.e., the non-blocking factor). The baseline is modeled similar to SCNN [8] and uses fixed 4x4 PE sized clusters. Even when the workload is dense (i.e., percentage of zero weights = 0), MAERI provides better utilization. This is because VGGNet uses 3x3 filters (and 3 channels). So each OFMAP requires 3x3x3=27 MACs, which use exactly 27 Multipliers/Adders in MAERI, and (4x4)x2=32 MAC units in the baseline, lowering utilization.

When the workload is sparser, the bandwidth requirement for partial sum collection increases because the PE array (multiplier switches and ART in MAERI) can cover more number of neurons at the same time. This becomes a bottleneck for the baseline where the clusters are connected by a bus (which limits bandwidth) even though the total number of computations goes down in a sparse workload. In contrast, MAERI provides 73.8% utilization even at 50% sparsity, and correspondingly 6.9 × speedup.

Cross-Layer Dataflow. We model five cross-layer dataflows by fusing a combination of AlexNet convolution layers. Figure 14 plots the utilization, and speedup of MAERI over a
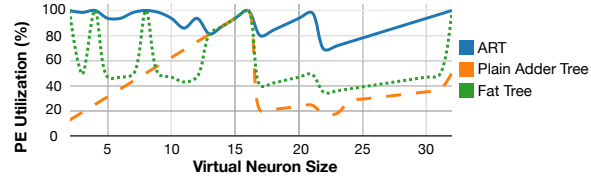

Figure 15. PE utilization with ART, fat-tree and four 16-wide plain adder trees when 64 PEs are used.

baseline accelerator with four 4x4 clusters (i.e., a 16:1 reduction trees in each). We observe 1.08 - 1.5 × speedup. Accelerators based on fixed clusters or fixed sized reduction trees are inefficient in utilizing all the PEs for certain mappings. For example, in Map C, we map three convolutional layers with three 3×3 filters. In the baseline, we can only use 9 PEs within each cluster; trying to utilize the remaining 7 PEs introduces non-uniform traffic that fixed interconnection cannot support. However, MAERI with its flexible interconnects can support such irregularity and thus maintains high utilization and provides maximum speedup.

6.3 MAERI Deep Dive

ART vs Fat Tree vs Plain Tree. Figure 15 plots the utilization of the MSes across three kinds of reduction trees: ART, fat-tree, and four plain adder trees as the size of each virtual neuron (VN) mapped over the trees increases. Our aim is to quantify the benefits discussed earlier in Figure 6. We can observe that the ART provides relatively uniform and high utilization while fat tree and plain adder trees involves significant fluctuation in utilization. Such fluctuations occur because the efficiency of plain adder trees and fat trees are highly sensitive to VN size (number of non-zero weights within a filter of a channel). The plain adder trees have low utilization because VNs less than size 16 end up instantiating an entire tree with idle multipliers, and thus provide 100% utilization only at a VN size of 16. If the VN size is a power of 2, the Fat Tree works identical to the ART since all operations fit within the binary tree structure and the ART's forwarding links are not required. When the VN size is not a power of two (which is the case in VGGnet and in sparse designs), the utilization of a fat-tree drops but ART continues to provide high utilization. ART also has fluctuations of utilization in cases where the total multipliers (64 in this example) is not a multiplier of the VN size as that leads to idle multiplier switches due to temporal folding (see Figure 17 (b)). Support from an advanced compiler may enable utilizing such multipliers by temporally folding large VNs over few multipliers.

MAERI trees vs. traditional NoCs. Compared to other traditional NoC designs, the NoC in MAERI is highly area- and power-efficient, as post-synthesis area and power overhead over NoC bandwidth plot in Figure 16 presents. The overhead of the NoC design in MAERI is minimal compared to mesh or crossbar while the NoC design provides sufficient bandwidth for the traffic in MAERI. This is because the
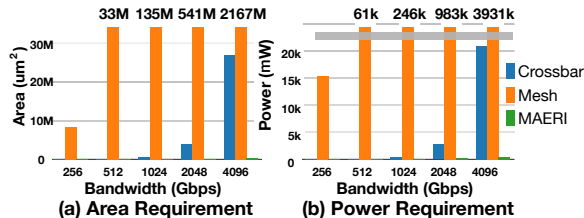
Figure 16. Area and power comparison between the NoC in MAERI and traditional NoCs.

NoC architecture in MAERI is optimized just for communication patterns within DNN accelerators, and it consists of extremely light-weight building blocks.

MAERI vs. systolic arrays. Systolic array is also considered as a major approach [23] to implement an accelerator because of the low PPA cost based on its relatively simple design and high parallelism it provides. We compare a systolic array-based design and MAERI using the example presented in Figure 17, which is a convolutional layer with eight 3x3x3 weight values, 5x5x3 input activation values, and a stride of one. In the example in Figure 17, both the systolic array (a) and MAERI (b) contain 64 PEs (i.e., MACs in Systolic array, multiplier and adder switches in MAERI). The systolic array reads input activation and weights from the left and the top of the array, respectively. Because systolic array is based on store-and-forward dataflow style, a controller needs to carefully adjust the injection time of each data, which lets the PEs remove complicated hardware for control data I/O. For example, if weight value A1 in the first 3D filter (red) and input activation a1 are injected to PE 0 at cycle *t*, weight value A1 in the second 3D filter (blue) needs to be injected at cycle *t+1* because input activation a1 arrives at the PE 1 at cycle *t+1*. To process one sliding window in the example convolution layer, the systolic array needs to read 216 weights and input activation (3×3×3×8). Because each sliding window is mapped on each row of the systolic array, the systolic array can process eight sliding windows within an iteration. Each iteration requires not 27 (the number of partial sums to be generated in a sliding window) cycles but 43 cycles (27 + 8 (injection delay)+ 8 (time to process the last weight/input activation set, which are weight I3 in the last filter (green) and input activation y3)) and generates eight output activations. Because the example requires sliding the window 25 times, the total number of iteration is four with an incomplete iteration that computes only one output activation at the last iteration. Therefore, the total cycles to process the example layer is 156 cycles (43 × 3 + 27). Although systolic array provides high throughput, it cannot reuse data within the PE array so it requires to read different data every cycle to each row and column, which results in high energy consumption [25]. Therefore, the systolic array need to read 1,323 times from the SRAM in the example.

In contrast, MAERI minimizes the number of SRAM reads by weight reuse in multiplier switches and multicasting input activations, which requires 516 reads (35% compared to the systolic array) for the same example. To the process the same convolutional layer in the example, MAERI first maps each channel in 3D filters (nine weights) as a VN across the multipliers, creating seven VNs in this example (with the last multiplier idle). We utilize temporary register in each adder switch to accumulate output activations from folded filters mapped in different iterations. In this manner, the number of iterations is four, and each iteration consumes 37 cycles (1 for configuration + 9 for weight distribution + 27 for multicasting input activations) with 8x chubby distribution tree. Therefore, MAERI requires 143 cycles to process the example convolutional layer, which reduces 9% of total latency compared to the example systolic array presented in Figure 17 (a).

In summary, MAERI provides 9% better throughput and 65% less SRAM reads in the above example. Extending the same analysis to 256x256 systolic array (TPU [23] specifications) vs MAERI with 256x256 multipliers on VGG16, we observe MAERI issues 6.3x less memory reads. Furthermore, the improvements can be more significant in irregular dataflows such as sparse and inter-layer fusion. However, the better performance and energy efficiency from less SRAM reads of MAERI comes at an area cost, as Figure 11 shows.

7 Related Works

In past five years, hardware support to accelerate DNN applications has been one of the most active research topics in computer architecture. The most related works to ours are related to supporting flexible dataflows and NoCs inside accelerators. In addition, we also discuss other related work across DNN accelerator design.

DNN accelerators for flexible dataflows: FlexFlow [34] and DNA [18] are two recent DNN ASIC proposals with similar motivation as this work. FlexFlow demonstrates a design that provides feature-map, neuron, and synapse-level parallelism, by different mapping strategies across PE rows and columns. DNA leverages Weight, Input, and Output Reuse within the same fabric. However, both FlexFlow and DNA's flexible dataflows are restricted *within* a layer, not across layers. Moreover, both FlexFlow and DNA are restricted to CNNs and cannot run LSTMs but MAERI can.

NoCs for DNN accelerators: Most of NoC studies for DNNs [40–43] have proposed meshes due to their flexibility to support all-to-all communication. Diannao [3] and Shidiannao [44] relied on mesh-based interconnects for data transfer. However, meshes add extremely high area and power overheads as our work and others [33] have shown. Dadiannao [4] employed fat tree for scatter and gather and 3D mesh via hyperTransport 2.0 to distribute data among nodes. Eyeriss [6] uses separate buses for its scatters and gathers.
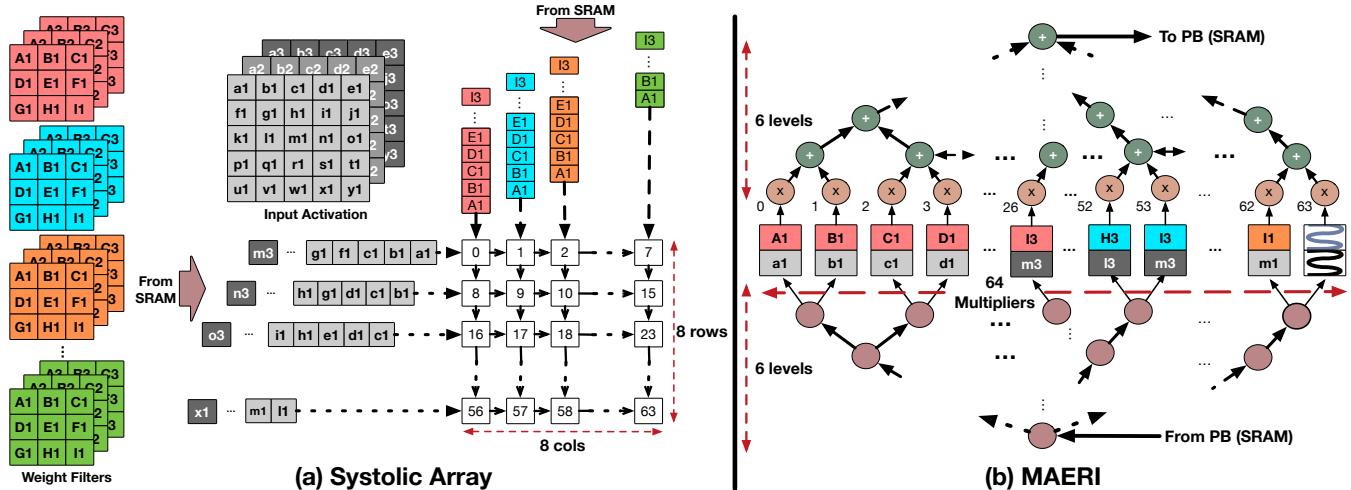
Figure 17. A mapping example of a convolution layer with eight 3x3x3 filters and 5x5x3 input activation over (a) a systolic array with 64 PEs and (b) MAERI with 64 multiplier switches.

CNN accelerators: Convolution Engine [45] explored the trade-off between flexibility and efficiency in accelerator domain with an example of convolution accelerator for image processing domain. Diannao [3], DaDiannao [15], ShiDiannao [44] are early spatial DNN accelerators. A recent work in FPGA [21] proposed constraint aware optimization tool for FPGA based accelerators. The design used simple adder trees within their computation engine, which can benefit from our ART. Eyeriss [25] analyzed data flow patterns in existing CNN accelerators and proposed a new data flow pattern for CNN acceleration called row stationary, which performs better than other data flow patterns in terms of throughput and energy efficiency. MAERI's VN mapping, without folding, essentially implemented row stationary across the multipliers, and output stationary over the ART to get the best of both dataflows. These works are all complementary to MAERI, since our configurable distribution and reduction trees can enable them to support more dataflows.

Cross-layer CNN Accelerators: A lot of recent works have performed design-space exploration of DNN accelerators, such as finding a better way to map data on to hardware or the best configuration of DNN accelerators, using novel simulation infrastructures [17, 46–48]. Fused-layer CNN [16] and others [35, 37, 49] have explored CNN architectures optimized for cross-layer optimizations over FPGAs. FPGAs provide immense flexibility in tuning the RTL for the right dataflow pattern and filter size(s) for mapping. MAERI aims to provide an ASIC substrate with similar flexibility.

Sparse CNN Accelerators: SCNN [8] is a recent accelerator for sparse CNNs, that leverages sparsity in activations and weights. Cnvlutin [19] compresses activation values based on the ReLU operator. Cambricon-X uses weight sparsity to keep only non-zero weights in its buffers. EIE [7] uses a compressed representation of weights and activations, delivering only non-zero operands to multipliers, for FC layers.

MAERI's aim is not to identify sparsity or compress data, but rather to construct arbitrary sized VNs given sparse data.

RNN accelerators: Although the algorithm of RNNs has been discussed for more than 20 years [27, 50–52], hardware design of RNNs was not as active as that of CNNs. In the last two years, hardware acceleration of LSTMs on FPGAs [53–57] has been explored. Maeda et al. suggested new learning algorithm of Hopfield RNN [58] which has been implemented on FPGA [59, 60]. An ASIC implementation for accelerating the control of RNN networks was recently demonstrated [61]. DNPU [62] implements CNN and RNN-LSTM in a single chip like MAERI but requires independent compute units for CNN and RNN while MAERI computes both CNN and RNN with the same substrate. Google's TPU [23] can also run CNNs and LSTMs together though it uses a Systolic Array which cannot provide the same level of configurability as our distribution and reduction networks.

8 Conclusion

In this paper we describe and evaluate MAERI, a fabric for mapping arbitrary dataflows that arise in DNNs due to its topology or mapping. Our approach is to augment multipliers and adders with tiny switches, and interconnect them via a novel reconfigurable interconnect that supports arbitrary sized neurons. We demonstrate how MAERI is not only capable of running CONV, LSTM, POOL and FC layers, but also supports cross-layer mapping and sparsity. MAERI's NoCs add minimal overheads over NoCs in state-of-the-art CNN accelerators while providing tremendous flexibility. We believe that MAERI is quite robust to supporting new optimizations in DNNs as it can construct arbitrary sized MAC engines very efficiently. This work also opens up exciting opportunities in compiler designs that can take arbitrary DNNs and map them efficiently over a MAERI-like fabric.

References

[1] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[2] K. Ding, N. Du, E. Elsen, J. Engel, W. Fang, L. Fan, C. Fougner, L. Gao, C. Gong, A. Hannun, T. Han, Vaino, L. Johannes, B. Jiang, C. Ju, B. Jun, P. LeGresley, L. Lin, J. Liu, Y. Liu, W. Li, X. Li, D. Ma, S. Narang, A. Ng, S. Ozair, Y. Peng, R. Prenger, S. Qian, Z. Quan, J. Raiman, V. Rao, S. Satheesh, D. Seetapun, S. Sengupta, K. Srinet, A. Sriram, H. Tang, L. Tang, C. Wang, J. Wang, K. Wang, Y. Wang, Z. Wang, Z. Wang, S. Wu, L. Wei, B. Xiao, W. Xie, Y. Xie, D. Yogatama, J. Yuan, J. Zhan, and Z. Zhu, "Deep speech 2: End-to-end speech recognition in english and mandarin," *arXiv preprint arXiv:1512.02595*, 2015.

[3] 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*, pp. 269–284, 2014.

[4] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "Dadiannao: A machine-learning supercomputer," in *MICRO*, pp. 609–622, 2014.

[5] 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.

[6] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2017.

[7] 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.

[8] 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 *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 27–40, ACM, 2017.

[9] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *NIPS*, pp. 1097–1105, 2012.

[10] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015.

[11] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 770–778, 2016.

[12] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[13] S. O. Arik, M. Chrzanowski, A. Coates, G. Diamos, A. Gibiansky, Y. Kang, X. Li, J. Miller, J. Raiman, S. Sengupta, A. Ng, and M. Shoeybi, "Deep voice: Real-time neural text-to-speech," *arXiv preprint arXiv:1702.07825*, 2017.

[14] 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 *ACM Sigplan Notices*, vol. 49, pp. 269–284, ACM, 2014.

[15] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "Dadiannao: A machine-learning supercomputer," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 609–622, IEEE Computer Society, 2014.

[16] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer CNN accelerators," in *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[17] L. Song, Y. Wang, Y. Han, X. Zhao, B. Liu, and X. Li, "C-brain: A deep learning accelerator that tames the diversity of cnns through adaptive data-level parallelization," in *DAC*, pp. 1–6, 2016.

[18] F. Tu, S. Yin, P. Ouyang, S. Tang, L. Liu, and S. Wei, "Deep convolutional neural network architecture with reconfigurable computation patterns," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2017.

[19] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in *ISCA*, pp. 1–13, 2016.

[20] 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.

[21] 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*, pp. 161–170, 2015.

[22] 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.

[23] N. P. Jouppi, , C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.

[24] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "Imagenet large scale visual recognition challenge," *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, 2015.

[25] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *ISCA*, pp. 367–379, 2016.

[26] J. Cong and B. Xiao, "Minimizing computation in convolutional neural networks," in *International Conference on Artificial Neural Networks*, pp. 281–290, Springer, 2014.

[27] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[28] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[29] Theano Development Team, "Theano: A Python framework for fast computation of mathematical expressions," *arXiv e-prints*, vol. abs/1605.02688, May 2016.

[30] M. Abadi, A. Agarwal, and P. Barham, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.

[31] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A matlab-like environment for machine learning," in *BigLearn, NIPS Workshop*, 2011.

[32] Y. Ma, Y. Cao, S. Vrudhula, and J.-s. Seo, "Optimizing loop operation and dataflow in fpga acceleration of deep convolutional neural networks," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 45–54, ACM, 2017.

[33] H. Kwon, A. Samajdar, and T. Krishna, "Rethinking nocs for spatial neural network accelerators," in *NOCS*, 2017.

[34] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li, "Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks," in *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*, pp. 553–564, IEEE, 2017.

[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 *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 161–170, ACM, 2015.

[36] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer cnn accelerators," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pp. 1–12, IEEE, 2016.

[37] H. Sharma, J. Park, E. Amaro, B. Thwaites, P. Kotha, A. Gupta, J. K. Kim, A. Mishra, and H. Esmaeilzadeh, "Dnnweaver: From high-level deep network models to fpga acceleration," in *the Workshop on Cognitive Architectures*, 2016.

[38] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, J. Wanderer, U. Holzle, S. Stuart, and A. Vahdat, "Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, pp. 183–197, 2015.

[39] R. Nikhil, "Bluespec system verilog: efficient, correct rtl from high level specifications," in *MEMOCODE*, pp. 69–70, IEEE, 2004.

[40] D. Vainbrand *et al.*, "Network-on-chip architectures for neural networks," in *NOCS*, pp. 135–144, 2010.

[41] J. Harkin *et al.*, "Reconfigurable platforms and the challenges for large-scale implementations of spiking neural networks," in *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pp. 483–486, IEEE, 2008.

[42] T. Theocharides *et al.*, "A generic reconfigurable neural network architecture implemented as a network on chip," in *SOC*, 2004.

[43] R. Emery *et al.*, "Connection-centric network for spiking neural networks," in *NOCS*, pp. 144–152, 2009.

[44] 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 *ACM SIGARCH Computer Architecture News*, vol. 43, pp. 92–104, ACM, 2015.

[45] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz, "Convolution engine: balancing efficiency & flexibility in specialized computing," in *ISCA*, pp. 24–35, 2013.

[46] Y. Ji, Y. Zhang, S. Li, P. Chi, C. Jiang, P. Qu, Y. Xie, and W. Chen, "Neutrams: Neural network transformation and co-design under neuromorphic hardware constraints," in *MICRO*, pp. 1–13, 2016.

[47] Y. S. Shao, B. Reagen, G.-Y. Wei, and D. Brooks, "Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures," in *ISCA*, pp. 97–108, 2014.

[48] M. Zhu, L. Liu, C. Wang, and Y. Xie, "Cnnlab: a novel parallel framework for neural networks using gpu and fpga-a practical study with trade-off analysis," *arXiv preprint arXiv:1606.06234*, 2016.

[49] Y. Shen, M. Ferdman, and P. Milder, "Maximizing CNN accelerator efficiency through resource partitioning," in *44th International Symposium on Computer Architecture (ISCA)*, 2017.

[50] J. L. Elman, "Finding structure in time," *Cognitive science*, vol. 14, no. 2, pp. 179–211, 1990.

[51] M. I. Jordan, "Serial order: A parallel distributed processing approach," *Advances in psychology*, vol. 121, pp. 471–495, 1997.

[52] C. Goller and A. Kuchler, "Learning task-dependent distributed representations by backpropagation through structure," in *IEEE Neural Networks*, vol. 1, pp. 347–352, 1996.

[53] A. X. M. Chang, B. Martini, and E. Culurciello, "Recurrent neural networks hardware implementation on fpga," *arXiv preprint arXiv:1511.05552*, 2015.

[54] Y. Guan, Z. Yuan, G. Sun, and J. Cong, "Fpga-based accelerator for long short-term memory recurrent neural networks," in *ASP-DAC*, pp. 629–634, 2017.

[55] S. Li, C. Wu, H. Li, B. Li, Y. Wang, and Q. Qiu, "Fpga acceleration of recurrent neural network based language model," in *FCCM*, pp. 111–118, 2015.

[56] M. Lee, K. Hwang, J. Park, S. Choi, S. Shin, and W. Sung, "Fpga-based low-power speech recognition with recurrent neural networks," in *SiPS*, pp. 230–235, 2016.

[57] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, H. Yang, and W. J. Dally, "Ese: Efficient speech recognition engine with sparse lstm on fpga," in *FPGA*, pp. 75–84, 2017.

[58] J. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities," *Proceedings of the national academy of sciences*, vol. 79, no. 8, pp. 2554–2558, 1982.

[59] Y. Maeda and M. Wakamura, "Simultaneous perturbation learning rule for recurrent neural networks and its fpga implementation," *IEEE Transactions on Neural Networks*, vol. 16, no. 6, pp. 1664–1672, 2005.

[60] R. Tavcar, J. Dedic, D. Bokal, and A. Zemva, "Transforming the lstm training algorithm for efficient fpga-based adaptive control of nonlinear dynamic systems," *Informacije Midem-Journal of Microelectronics Electronic Components and Materials*, vol. 43, no. 2, pp. 131–138, 2013.

[61] J. Kung, D. Kim, and S. Mukhopadhyay, "Dynamic approximation with feedback control for energy-efficient recurrent neural network hardware," in *ISLPED*, pp. 168–173, 2016.

[62] D. Shin, J. Lee, J. Lee, and H.-J. Yoo, "14.2 dnpu: An 8.1 tops/w reconfigurable cnn-rnn processor for general-purpose deep neural networks," in *ISSCC*, pp. 240–241, 2017.