You Cannot Improve What You Do not Measure: FPGA vs. ASIC Efficiency Gaps for Convolutional Neural Network Inference

ANDREW BOUTROS, SADEGH YAZDANSHENAS, and VAUGHN BETZ,
Department of Electrical and Computer Engineering, University of Toronto

Recently, deep learning (DL) has become best-in-class for numerous applications but at a high computational cost that necessitates high-performance energy-efficient acceleration. The reconfigurability of FPGAs is appealing due to the rapid change in DL models but also causes lower performance and area-efficiency compared to ASICs. In this article, we implement three state-of-the-art computing architectures (CAs) for convolutional neural network (CNN) inference on FPGAs and ASICs. By comparing the FPGA and ASIC implementations, we highlight the area and performance costs of programmability to pinpoint the inefficiencies in current FPGA architectures. We perform our experiments using three variations of these CAs for AlexNet, VGG-16 and ResNet-50 to allow extensive comparisons. We find that the performance gap varies significantly from 2.8× to 6.3×, while the area gap is consistent across CAs with an 8.7 average FPGA-to-ASIC area ratio. Among different blocks of the CAs, the convolution engine, constituting up to 60% of the total area, has a high area ratio ranging from 13 to 31. Motivated by our FPGA vs. ASIC comparisons, we suggest FPGA architectural changes such as increasing DSP block count, enhancing low-precision support in DSP blocks and rethinking the on-chip memories to reduce the programmability gap for DL applications.

CCS Concepts: • **Hardware → Reconfigurable logic and FPGAs**; **Hardware accelerators**; Reconfigurable logic applications;

Additional Key Words and Phrases: Deep learning, convolutional neural networks, FPGA, ASIC

ACM Reference format:
Andrew Boutros, Sadegh Yazdanshenas, and Vaughn Betz. 2018. You Cannot Improve What You Do not Measure: FPGA vs. ASIC Efficiency Gaps for Convolutional Neural Network Inference. *ACM Trans. Reconfigurable Technol. Syst.* 11, 3, Article 20 (December 2018), 23 pages.
https://doi.org/10.1145/3242898

1 INTRODUCTION

Recent advances in deep learning (DL) have led to breakthroughs in a myriad of fields, achieving unprecedented accuracy in tasks that were thought to be inherently unsuitable for our computing machines to perform. It has become, in a very short time span, the *de-facto* standard for numerous applications ranging from simple image classification [36], machine translation [44],

Authors' addresses: A. Boutros and V. Betz, Department of Electrical and Computer Engineering, University of Toronto, 10 King's College Road, Toronto, Ontario M5S 3G4, Canada and Vector Institute, Toronto, ON, Canada; emails: andrew.boutros@mail.utoronto.ca, vaughn@eecg.utoronto.ca; S. Yazdanshenas, Department of Electrical and Computer Engineering, University of Toronto, 10 King's College Road, Toronto, Ontario M5S 3G4, Canada; email: sadegh.yazdanshenas@mail.utoronto.ca.

1936-7406/2018/12-ART20 $15.00
https://doi.org/10.1145/3242898

and speech recognition [10] to generating artistic paintings [9], composing music [7], and beating world champions in complex board games [41]. Interestingly, the basic foundations of DL and the algorithm currently used to train deep neural networks (DNNs), known as back-propagation, were established in the 1980s [35]. But it was not until recent years that it experienced a resurgence of interest [20], powered by both the abundance of data required for training and the availability of the tremendous compute-power necessary to train and deploy those models.

However, the main drawback of DNNs remains to be their high computational complexity when compared to conventional detection and classification computer vision algorithms based on hand-crafted features. For example, a relatively simple eight-layer convolutional neural network (CNN), AlexNet [20], has a computational complexity of 25.8GOP/Mpixel for its convolutional layers, which is 36.9× higher than that of a conventional histogram of oriented gradients feature extractor [43]. This gap grows even wider as we seek to improve the accuracy of CNNs by building deeper, bigger and more complex models that can surpass human-level performance on visual recognition tasks [14]. The ImageNet large-scale visual recognition challenge witnessed a 15× increase in operations required per image inference in return for an 11.7% reduction in classification error between 2012 and 2015 [15, 36]. This substantial increase in compute requirements motivates high-performance and energy-efficient hardware accelerators to replace or co-exist with conventional CPUs in executing both CNN training and inference tasks.

The training of CNN models is commonly performed in floating-point representation on graphics processing units (GPUs) having thousands of cores and large external memory bandwidth. It does not require much effort to deploy existing models or train new ones on GPUs using various frameworks (e.g., Caffe [18] and TensorFlow [1]) that exploit highly optimized GPU libraries such as Nvidia CuDNN [5] for dense and sparse matrix operations. Although GPUs can deliver high performance by performing batch computations, they are extremely power-hungry. This is affordable for training, which has no constraints on output latency and is carried out a limited number of times during the development phase. However, when it comes to inference, this is not ideal for a wide class of applications that have limited power budget and tight latency constraints such as mobile embedded platforms, self-driving cars or large-scale datacenter services.

To achieve the best performance and energy-efficiency, many researchers have focused on building custom application-specific integrated circuits (ASICs) for accelerating CNNs inference workloads. Some examples are DaDianNao [3] that accelerates different types of DNNs using a multi-chip architecture and Eyeriss [4] that focuses on energy-efficient acceleration of convolutional layers by maximizing data re-use, performing data compression and using a zero-skipping technique. Despite being an attractive solution, ASICs do not offer enough flexibility to accommodate the rapid evolution of CNN models and the emergence of new types of layers used in them including the branching, elementwise addition and batch normalization layers as in more recent models (e.g., GoogLeNet [45] and ResNet [15]). As well, the high non-recurring engineering (NRE) cost and time for design, verification and fabrication of a large ASIC chip makes it difficult to keep pace with the rapid model improvements in this space.

As a trade-off between performance, power-efficiency, and flexibility, FPGAs offer an interesting design point between GPUs and ASICs and recently have had much success in accelerating datacenter workloads in general [32] and more specifically CNN inference tasks [30]. In contrast to GPUs, FPGAs are generally more energy-efficient. A high-end Titan X Nvidia GPU can consume up to 5× more power compared to a high-end Intel Arria 10 FPGA running AlexNet inference tasks [2]. Several studies have also shown that CNN inference does not require high-precision floating-point computations and can be carried out using fixed-point arithmetic for less than 1% accuracy degradation [13]. This wide variety of precisions used in CNN inference matches well with FPGAs as they can execute non-standard custom bit-width datapaths with much higher efficiency

and flexibility than GPUs. However, they have a shorter turn-around time, less NRE cost, and can be re-configured to support new models and layer types when compared to ASIC accelerators. Another interesting advantage for FPGAs is that they offer a variety of I/Os that support different communication protocols. This is useful when the CNN accelerator is a part of a larger system and receives inputs from different types of digital and analog sensors as the case in automotive applications. However, FPGAs run at significantly lower frequencies due to their reconfigurability overhead and thus have lower raw performance compared to both GPUs and ASICs.

For this reason and despite their drawbacks, several companies have developed ASIC solutions to meet the processing needs of high-performance DL applications. A recent example for that is Google's Tensor Processing Unit [19] that was deployed in datacenters to accelerate inference tasks for various types of DNNs. It has almost 17× more multiply accumulate (MAC) units, 5.6× more on-chip memory and runs at 3.5× higher frequency when compared to Microsoft's Catapult V1 [32] that uses Intel Stratix V FPGAs. In this work, we study the area and performance gap between FPGAs and ASICs in accelerating inference tasks using multiple CNN computing architectures (CAs) to highlight the limitations of current FPGA architectures and how they affect the overall performance of DL accelerators. The motive behind this study is twofold; First, it shows which design practices are more suitable for FPGA platforms and make the best use of current FPGA architectures. Second, it provides FPGA architects with data on where FPGAs have the largest efficiency gap compared to ASICs, which can lead to insights on how current FPGA architectures could be modified to shrink this gap and deliver higher performance in a domain with extremely high demand such as DL.

In this article, we make the following contributions:

- We implement highly optimized RTL designs for three state-of-the-art CAs that use different parallelization schemes to accelerate CNNs. We then extend each of these previously published architectures to support all layer types required to implement three different CNN models: AlexNet, VGG-16, and ResNet-50 to ensure our comparisons consider a broadly representative set of CNN models and implementations.
- We present a quantitative comparison of area and performance results to measure the gap between the same CAs implemented on a high-end Intel Arria 10 FPGA and a 28nm ASIC.
- We trace back the bottlenecks resulting in this gap and pinpoint the limitations of current FPGA architectures in accelerating CNNs.

2 BACKGROUND

Deep Neural Networks are a class of machine-learning algorithms that were developed to mimic the information-processing paradigm in biological nervous systems. The human brain as an example has an average of around 86 billion neurons [16] connected in a complex network in which each neuron receives inputs from its surrounding neurons and fires an activation if those inputs are greater than a specific threshold. Inspired by this system, DNNs typically consist of several layers each of which has $d^{(l)}$ neurons where l is the layer number ranging from 1 to L. Each artificial neuron performs a biased weighted sum of all its inputs followed by a non-linear activation function to produce its output as shown in Equation (1), where $x_i^{(l)}$ is the output of neuron i of layer l, $w_{ij}^{(l)}$ is the weight parameter between the neuron j in layer l and neuron i in layer $l-1$, $w_{0j}^{(l)}$ is the bias term and θ is the non-linear activation function that can be a sigmoid, tanh, or rectified linear unit (ReLU) function. This equation can be viewed as a series of MAC operations, which form the majority of computations in DNNs:

$$x_j^{(l)} = \theta\left(w_{0j}^{(l)} + \sum_{i=1}^{d^{(l-1)}} x_i^{(l-1)} w_{ij}^{l}\right). \tag{1}$$

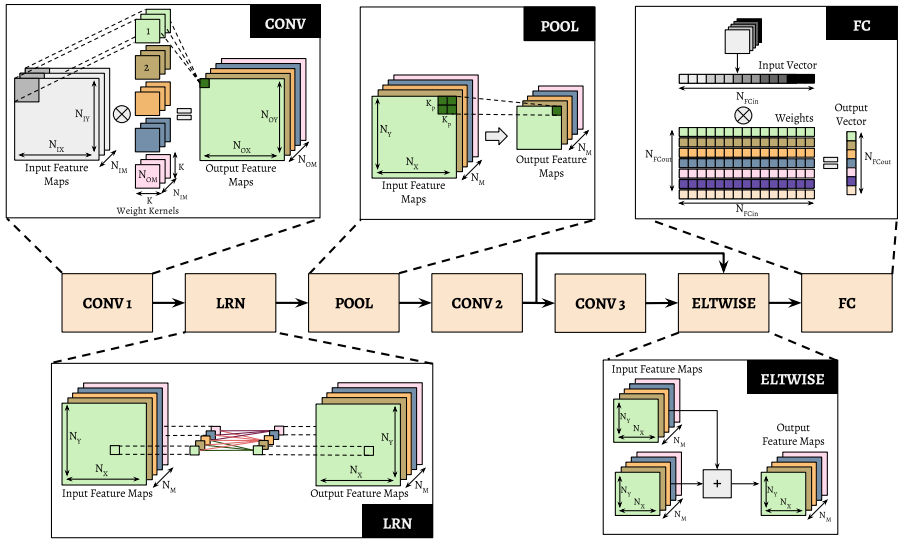

Fig. 1. Different layer types in an example CNN.

CNNs are a subset of DNNs in which the connections between neurons of successive layers are sparse. Each neuron receives inputs only from neighboring neurons of the previous layer or so-called its *receptive field*. This significantly reduces the number of weights and MAC operations required and achieves high accuracy in applications with spacial or temporal correlation between input samples such as image classification, gesture and speech recognition. Sections 2.1 and 2.2 describe the main layers of a CNN and present a summary of the previous related work on accelerating CNNs on FPGAs.

2.1 Overview of CNN Layers

CNN models typically consist of different layer types cascaded together such that the output of a specific layer is consumed by the subsequent one in a feed-forward scheme during inference. In Figure 1, we show an example CNN, and we illustrate the functionality of each of the layer types subsequently explained in this section.

2.1.1 Convolutional (CONV) Layers. A CONV layer takes a set of N_{IM} two-dimensional input feature maps. It accumulates the results of 2D convolutions with stride S between each input feature map and its corresponding $K \times K$ kernel of learnable weights to produce a two-dimensional output feature map. This is performed using N_{OM} different sets of kernels to generate N_{OM} output feature maps that are consumed by the subsequent layer. CONV layers are very compute-intensive and represent the majority of computation in a CNN, which motivated many designers to focus on accelerating only the CONV and not all CNN layers [55]. We also notice that as CNN models get deeper, the portion of CONV layers operations compared to the total number of operations increases as they constitute 91.6%, 99.1%, and 99.8% of the total operations count for AlexNet, VGG-16, and ResNet-50, respectively.

The computation of CONV layers can be summarized using the six nested loops in Algorithm 1; they are highly parallelizable and can achieve high gains through hardware acceleration. However, it is a non-trivial optimization problem to choose the tiling and unrolling factors of those loops to achieve the best performance within the limited available hardware resources [27]. Typically, a

ALGORITHM 1: Nested loops for CONV layers computation

Loop 1: for $(j = 0; j < N_{OM}; j++)$ **do**
  Loop 2: for $(x = 0; x < N_{OX}; x \mathrel{+}= S)$ **do**
    Loop 3: for $(y = 0; y < N_{OY}; y \mathrel{+}= S)$ **do**
      Loop 4: for $(i = 0; i < N_{IM}; i++)$ **do**
        Loop 5: for $(k_x = 0; k_x < K; k_x++)$ **do**
          Loop 6: for $(k_y = 0; k_y < K; k_y++)$ **do**
            $out(j, x, y) \mathrel{+}= in(i, x + k_x, y + k_y) \times weight(j, i, k_x, k_y)$
      $out(j, x, y) \mathrel{+}= bias(j)$

non-linear activation function such as the ReLU function $\theta(x) = max(0, x)$ is applied to the outputs of a CONV layer before passing them to the next layers.

2.1.2 Local Response Normalization (LRN) and Batch Normalization (BNORM) Layers. LRN is a heavily arithmetic layer that was used in the early CNN models such as AlexNet to normalize each element in its input feature maps with respect to the elements at the same location in the adjacent K_N maps using the formula in Equation (2). The function of the LRN layer is to create lateral inhibition for the output values especially when using ReLU as an unbounded activation function [20]. However, this layer is removed in newer models and is sometimes replaced by BNORM layer followed by scaling, as in ResNets, which cuts down the required training steps and achieves the same accuracy. The computation for the BNORM layer is shown in Equation (3) where μ and σ^2 are statistically computed over the training data set and γ and β are learned during the training phase of the CNN [17] but are all constants for inference:

$$out(j, x, y) = in(j, x, y) \times \left(1 + \frac{\alpha}{K_N} \sum_{i=\max(0, j-\frac{K_N}{2})}^{\min(j+\frac{K_N}{2}, N_{OM})} in^2(i, x, y)\right)^{-\beta}, \tag{2}$$

$$out(j, x, y) = \gamma \left(\frac{in(j, x, y) - \mu}{\sqrt{\sigma^2}}\right) + \beta. \tag{3}$$

2.1.3 Pooling (POOL) Layers. Another key layer in CNN is the POOL layer, which acts as a down-sampling function such that its input feature maps of size $N_X \times N_Y$ are reduced in size but the number of input and output feature maps stays the same. There are different variations for POOL layers such as Max-POOL and Average-POOL, where each element in the output feature map represents the maximum or average value of a window of size $K_P \times K_P$ in the original input feature map, respectively.

2.1.4 Element-Wise (ELTWISE) Layers. Recent CNNs have more complex models with branching layers and skipping connections forming a directed acyclic graph as shown in Figure 1 after CONV2 layer. An ELTWISE layer combines two branches by performing an element-wise addition of the elements of a skipping branch and the results of a CONV layer. Reference [38] proposed the use of weighted addition in ELTWISE layers for deeper networks with more than 100 layers; however, we focus on the unweighted variation of ELTWISE layers in this work. For this layer, the dimensions of the output feature maps match those of the input feature maps.

2.1.5 Fully Connected (FC) Layers. The last layers of CNNs are typically FC layers, which are similar to those of conventional DNNs. The output of an FC layer is a one-dimensional vector of size $N_{FC_{out}}$. Each element in this vector is a weighted sum of all the outputs of the previous layer, which were re-shaped into a one-dimensional vector of size $N_{FC_{in}}$. As shown in Figure 1,

it is characterized by the large number of weights involved in computation ($N_{FC_{in}} \times N_{FC_{out}}$) that, unlike the convolution kernels, cannot be re-used. Therefore, FC layers are usually memory-bound, but more recent CNN models have a smaller number of FC layers with fewer weights making them less problematic. For instance, ResNet-50 has only 1 FC layer that has about 8% of the total number of weights in the network compared to three FC layers with 96% of the network weights in AlexNet, which further prioritizes the acceleration of CONV layers over other types of layers.

2.2 Related Work

Research efforts to accelerate CNNs on FPGAs can be classified into two major categories. The first category of work focuses on optimizing the mapping of CNN models to current FPGA architectures. For example, Reference [55] presents an analytical design methodology for design space exploration using the roof-line model to find the optimal loop unrolling and tiling parameters for the CONV loops shown in Algorithm 1. This work is extended in Reference [56] to a multi-FPGA cluster using dynamic programming with the target of maximizing throughput or minimizing latency. To overcome under-utilization of resources resulting from different sizes of CONV layers, References [39] and [40] partition the available resources using a dynamic programming technique into multiple convolutional layer processors, each of which is optimized for a subset of CONV layers. Another aspect of optimizing CNNs for FPGA acceleration is model compression by using techniques such as Singular Value Decomposition for FC layers [33]. Another compression technique reduces precision down to ternary [31, 49] or binary [29, 47] networks that are inherently more FPGA-friendly, and exhibit little or no accuracy degradation by increasing the size of the network as in Reference [28]. The use of non-standard floating-point number representations has also been proposed by Microsoft's BrainWave project [6] that uses its custom 8-bit/9-bit floating-point precision without suffering any accuracy loss. Recent work has also proposed the use of mathematical optimizations such as Winograd and Fast Fourier Transformations to decrease the number of MAC operations required in CONV layers as in References [2, 24, 57].

The second category seeks to ease development of DL accelerators on FPGAs such that it requires minimal hardware design expertise. Some works have investigated the use of High-Level Synthesis FPGA tools to implement CNNs in high-level programming languages that are synthesized into hardware [42]. Another widely investigated approach is to build automatic compilers to produce an end-to-end optimized accelerator for a specific CNN model and a specific FPGA platform [23, 25, 26]. In Reference [48], the authors present a framework that takes a CNN model described in a domain-specific language, converts it to a synchronous dataflow graph, optimizes performance and resource utilization via algebraic transformations, and finally generates a Vivado HLS hardware design. An open-source RTL template-based compiler that transforms a high-level description of the CNN model in the same *prototxt* format used by Caffe into an FPGA accelerator is also presented in Reference [37]. Similar frameworks were presented in References [51] and [11] that use Caffe-described and TensorFlow-described models along with RTL and RTL-HLS hybrid templates, respectively, to implement FPGA accelerators for not only CNN models but also Multi-Layer Perceptrons and Recurrent Neural Networks. The authors of Reference [52] implement an automated design flow that generates high-performance systolic array CNN architectures and a two-phase design space exploration scheme using analytical models as well as on-board implementations.

Our work is complementary to these studies and serves as the first step toward improving the current FPGA architecture, which was considered a constant factor by all previous works, for more efficient acceleration of emerging and highly motivated applications as DL. To the best of our knowledge, this work is the first attempt to quantify the area and performance gap between FPGA and ASIC implementations of state-of-the-art CNN CAs, highlight the architectural features

Table 1. Main Differences between the Three CAs

Comparison Aspect	ASU-like	Intel-DLA-like	Chain-NN-like
MAC Units Array	Three-dimensional	Two-dimensional	One-dimensional
Convolution Method	Conventional sliding-window	Winograd Transform for 3×3 convolutions	Conventional sliding-window
Weight Buffers	A centralized buffer for a group of PEs	A centralized buffer for a group of PEs	A small distributed buffer for each PE
Double Buffering	Double buffers for weights in FC	Interchangeable double buffers for features in CONV	No double buffering

of current FPGA architectures causing it, and present suggested architectural solutions that can reduce this gap.

3 COMPUTING ARCHITECTURES

We implement three different highly optimized state-of-the-art CAs for accelerating CNN inference tasks in RTL using parameterizable SystemVerilog HDL. We refer to the three CAs as ASU-like [26, 27], Intel-DLA-like [2], and Chain-NN-like [50]. We implement all the hardware computational blocks required to execute all the layers described in Section 2.1 for three different CNN models: AlexNet, VGG-16, and ResNet-50. We also implement the control logic required to run the CAs starting from reading the input features and weights from on-chip buffers, transferring them to the computational blocks, and writing the final results in the output feature buffers. The on-chip buffer sizes and the parallelization factors for each of the nested CONV loops are fixed in both the FPGA and ASIC implementations for each of these CAs according to the optimal design point originally reported in References [2, 27, 50]. For consistency and to enable fair comparisons, we also use a fixed-point data representation for all three CAs with 16-bit features and 8-bit weights as in Reference [27], which causes less than 2% accuracy degradation. We consider the external memory interface and direct memory access engines to be out of the scope of this work, as they do not affect the conclusions we seek to draw about the performance and area gaps or the bottlenecks of current FPGA architectures in accelerating CNNs. However, our performance models put off-chip data transfer into consideration according to any external memory interface that we specify. In our experiments, we report two sets of results: one assuming infinite bandwidth and the other assuming one bank of DDR4 memory at 1200MHz with a total bandwidth of 17GB/s similar to that used in Reference [2].

We carefully chose those three CAs out of numerous architectures proposed in the literature to be diverse; the wide variations between them help ensure our analysis of FPGA vs. ASIC efficiency has broad applicability. The main differences between the three CAs, summarized in Table 1, are:

- All three CAs have different parallelization schemes. In other words, the array of MAC units in each CA has a different number of dimensions leading to different execution orders, tiling and unrolling factors for the CONV loops in Algorithm 1. Output tiles of size ($P_{OM} \times P_{OX} \times P_{OY}$), ($P_{OM} \times P_{OX} \times 1$), and ($P_{OM} \times 1 \times 1$) are produced by the ASU-like, Intel-DLA-like, and Chain-NN-like PE arrays, respectively.
- The Intel-DLA-like CA uses a mathematical optimization for CONV layers with kernels of size 3×3 known as the Winograd Transform [22], which reduces the number of MAC operations needed to compute convolutions. However, the ASU-like and Chain-NN-like CAs

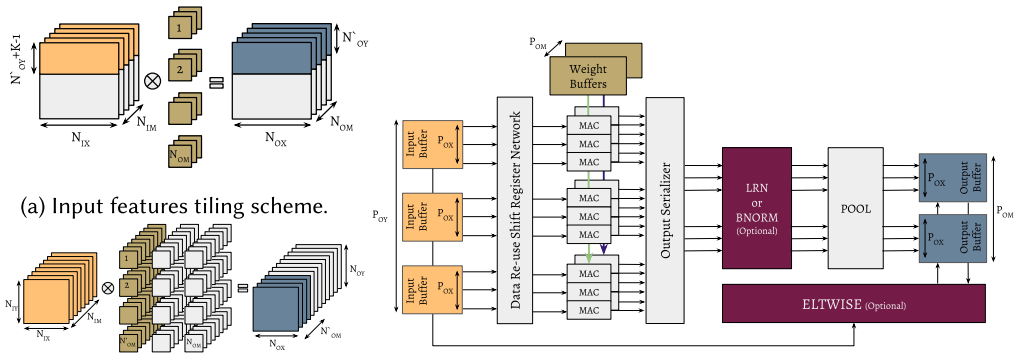

Fig. 2. ASU-like CA tiling schemes and hardware architecture.

perform conventional sliding-window convolution operations. This enables us to explore different convolution schemes with different degrees of control logic complexity and observe their effect on the area and performance gaps.

- The three CAs implement their weight buffers differently. The Chain-NN-like CA stores the kernel weights in small distributed buffers such that every MAC unit has its local scratchpad for weights implemented in the FPGA's soft logic (MLABs). In contrast, both the ASU-like and Intel-DLA-like CAs have larger weight buffers implemented using on-chip memory blocks (BRAMs) to feed a group of MAC units. In FC layers, the Intel-DLA-like CA also interchanges the roles of weight and feature buffers.
- The CAs differ in whether and how they use double-buffering to hide memory transfer time. The ASU-like CA uses double-buffering for weights to hide the computation time of FC layers by filling one buffer from off-chip memory while using the weights in the other buffer for computations. The Intel-DLA-like CA uses double-buffering by interchanging input and output buffers after each layer to eliminate any external memory transfers if all the output feature maps of a layer can fit in on-chip buffers. The Chain-NN-like CA does not use any double-buffering techniques.

None of the three CAs is available as an open-source implementation, and hence we implemented them from scratch to carry out the study presented in this article under controlled conditions (e.g., RTL implementation, same FPGA platform, same weight and activation precisions, etc.) to enable fair comparisons and focus only on the architectural aspects of these CAs. In Sections 3.1, 3.2, and 3.3, we describe the details of the three CAs we implemented and any extensions added to them for the sake of our study.

3.1 ASU-like CA

This CA was proposed in Reference [27] by Ma et al. from Arizona State University (ASU) and then expanded in Reference [26] to support the ELTWISE and BNORM layers used in recent CNN models. The core of this CA, shown in Figure 2(c), is a three-dimensional MAC unit array of size $P_{OM} \times P_{OX} \times P_{OY}$ that can compute both CONV and FC layers.

Feature maps and weights are tiled to minimize external memory transfers by either buffering all weights or all input feature maps in on-chip memory at any layer of the CNN model. In the shallower layers of the network, all the weights but only $N'_{OY} + K - 1$ rows of the input feature maps are buffered on-chip such that $0 < N'_{OY} \leq N_{OY}$ as shown in Figure 2(a). In the deeper layers

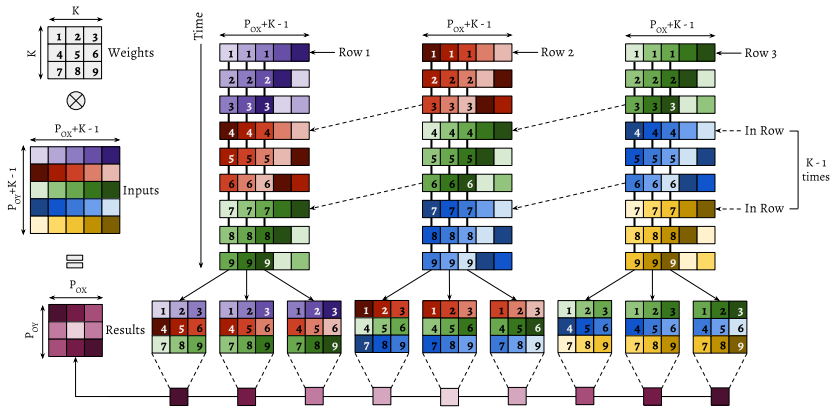

Fig. 3. Data re-use shift register network operation for ASU-like CA with $P_{OX} = P_{OY} = K = 3$ and $P_{OM} = 1$.

with smaller input and output feature maps and more weights, all features but only N'_{OM} sets of weight kernels are buffered on-chip such that $0 < N'_{OM} \leq N_{OM}$ as shown in Figure 2(b). The on-chip input and weight buffers, all implemented in BRAMs, are organized to supply the MAC units in the convolution engine with enough inputs to keep them busy at every clock cycle. There are P_{OY} input buffers, each of which supplies the MAC units with P_{OX} input features that get multiplied by weights from P_{OM} different weight buffers as shown in Figure 2(c).

The convolution engine performs the computation of Loops 1, 2, and 3 in Algorithm 1 in parallel using the three-dimensional array of MAC units. Each MAC unit sequentially accumulates the results of one kernel (Loops 5 and 6) across all input feature maps (Loop 4) and stores the partial sum locally in the accumulator. This means that after $K \times K \times N_{IM}$ cycles, each MAC unit outputs its final result producing $P_{OM} \times P_{OX} \times P_{OY}$ outputs at the same time. This parallelization scheme has several advantages; it does not require any movement of partial sums as every MAC unit locally accumulates the results across Loops 4, 5, and 6 without the need for communication between MAC units or any intermediate on-chip storage. It also allows flexible implementation of convolutions of any input feature map count and any kernel size as a result of sequentially executing Loops 4, 5, and 6. For example, for any input feature map count, convolutions of size 3×3 and 5×5 are executed in 9 and 25 cycles, respectively. The convolution engine is preceded by a complex network of P_{OY} circular shift registers of size $(P_{OX} + K - 1)$ each. Figure 3 shows how $P_{OX} \times P_{OY}$ convolution results are computed using this shift register network over $K \times K$ time steps, where colored boxes are input/output features, white numbered boxes are kernel weights and colored numbered boxes indicate a multiplication operation between an input feature and a kernel weight. At every time step, the multiplication result is accumulated inside the MAC unit and a shift left of the input data is performed. Every K time steps a new row is loaded from the input buffers and data is re-arranged and transfered between the circular shift registers as indicated by the dashed arrows in the figure. After $K \times K$ time steps, this is repeated for N_{IM} input maps before each MAC unit produces its final result. The convolution engine is followed by an output serializer that takes $P_{OM} \times P_{OX} \times P_{OY}$ results and serializes them over P_{OY} cycles. After the output serializer, there can be a normalization block that is either LRN or BNORM according to the implemented CNN model, then max pooling block and finally the output buffers. An optional ELTWISE block is used in the ResNet-50 model.

Extensions: Both References [27] and [26] originally implement this CA for several CNN models including ResNet-50 and VGG-16. Therefore, they implement all the hardware blocks shown

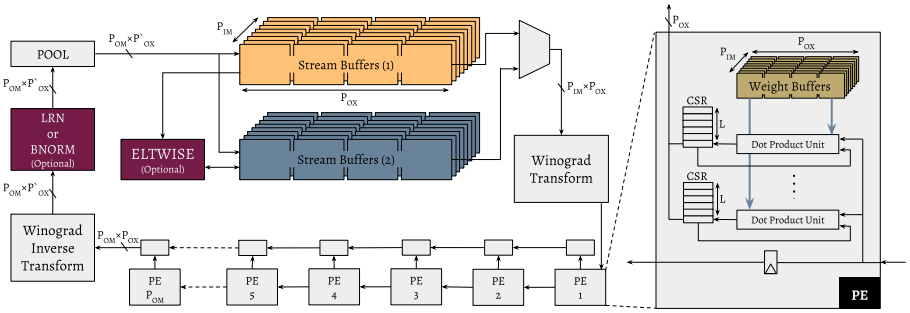

Fig. 4. Intel-DLA-like CA and the internal architecture of each processing element.

in Figure 2(c) except for the LRN block used in the AlexNet model. The LRN block is a heavily arithmetic block as it contains squaring, addition, multiplication and exponentiation operations. Since all the DSP blocks are consumed by the convolution engine, we implement all multiplication operations in the LRN block using soft multipliers that are found to be not limiting the maximum operating frequency. We implement the exponentiation operation of Equation (2) using a piecewise-linear function consisting of 20 points that we computed using the α and β values from the AlexNet model similar to Reference [42].

3.2 Intel-DLA-like CA

In Reference [2], Intel presented the Deep Learning Accelerator (DLA), which is considered to be the state-of-the-art FPGA accelerator for the AlexNet CNN model. The core of this CA is an array of P_{OM} processing elements (PEs) connected in a daisy chain scheme where each PE receives input features and passes them to the subsequent PE in the next clock cycle as shown in Figure 4.

This CA uses double-buffered *stream buffers* such that input features of a CONV layer are read from one buffer and its outputs are stored in the other one, which then serves as the input buffer for the next layer. The two buffers continue to interchange roles as input and output buffers after every layer without the need to store any intermediate results in external memory. After the last CONV layer, outputs are stored in off-chip memory before starting the computations of FC layers. Each PE contains local weight buffers that feed its dot product units with inputs at every clock cycle. For the FC layers, batch processing is used to allow weight re-use among multiple input features. In contrast to the CONV layers, features of a batch of size B inputs are stored in "weight" buffers inside the PEs while the weights are stored in the stream buffers and are passed between the PEs using the daisy chain connection. For our study, we report the results for both $B = 1$ that minimizes latency and can be compared to other CAs that do not support batch processing and $B = 96$ that maximizes throughput and aligns with the reported results in Reference [2].

A major feature of this CA is its use of a mathematical optimization known as the Winograd Transform to reduce the number of MAC operations required to compute a convolution [22]. In Reference [2], an $F(4 \times 4, 3 \times 3)$ transform is performed using a weight matrix of size 3×3 and an input feature matrix of size 6×6 resulting in an output matrix of size 4×4. Equation (4) shows the Winograd transform and inverse transform for these sizes where G and B^T are used to transform the weight matrix W and the input feature matrix X, respectively, $\odot$ is the element-wise multiplication operator and then A^T is used to perform the inverse transform and obtain the output matrix Y. For this CA, the transform of the learned weights is done beforehand for the CONV layers of kernel size 3×3, since they are fixed after training the model while the transform of input features and inverse transform of the final result cannot be performed in advance and hence are performed

on-chip:

$$Y = A^T\Big[[GWG^T] \odot [B^T XB]\Big]A,$$

$$A^T = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & 2 & -2 & 0 \\ 0 & 1 & 1 & 4 & 4 & 0 \\ 0 & 1 & -1 & 8 & -8 & 1 \end{bmatrix} \quad G = \begin{bmatrix} \frac{1}{4} & 0 & 0 \\ -\frac{1}{6} & -\frac{1}{6} & -\frac{1}{6} \\ -\frac{1}{6} & \frac{1}{6} & -\frac{1}{6} \\ \frac{1}{24} & \frac{1}{12} & \frac{1}{6} \\ \frac{1}{24} & -\frac{1}{12} & \frac{1}{6} \\ 0 & 0 & 1 \end{bmatrix} \quad B^T = \begin{bmatrix} 4 & 0 & -5 & 0 & 1 & 0 \\ 0 & -4 & -4 & 1 & 1 & 0 \\ 0 & 4 & -4 & -1 & 1 & 0 \\ 0 & -2 & -1 & 2 & 1 & 0 \\ 0 & 2 & -1 & -2 & 1 & 0 \\ 0 & 4 & 0 & -5 & 0 & 1 \end{bmatrix}. \quad (4)$$

Each PE in the convolution engine of this CA consists of a buffer for the Winograd-transformed weights, P_{OX} dot-product units and their corresponding circular shift registers for storing partial sums. Each dot product unit is pipelined into L stages and uses the dedicated chain between DSP blocks on the FPGA to multiply and accumulate P_{IM} Winograd features and weights and then store the partial result in a circular shift register (CSR) of size L as shown in Figure 4. Therefore, each dot product unit can interleave the computation of L different MACs such that after L cycles, it takes as an input the partial sum previously produced and adds to it the MAC result of the next P_{IM} features and weights. After all N_{IM} features are processed, the final result is produced and the circular shift register is reset to zeros before starting the processing of the next set of input features. The convolution engine consists of P_{OM} PEs connected in a daisy chain scheme allowing a better floorplan of the design on the FPGA with less fan-out from the input stream buffer to the convolution engine, and thus enabling a higher operating frequency. The convolution engine is followed by an inverse Winograd block that transforms $P_{OX} \times P_{OM}$ inputs into $P'_{OX} \times P_{OM}$ outputs. This is followed by LRN and POOL blocks that process $P'_{OX} \times P_{OM}$ results in parallel before storing them back into the output stream buffer. Both the P_{OX} and P'_{OX} parameters are specified to be 6 and 4, respectively, according to the Winograd transform size used. Design space exploration was carried out in Reference [2] to find the optimal values for P_{IM} and P_{OM} and they were chosen to be 8 and 48, respectively.

Extensions: This CA was originally implemented for the relatively small AlexNet CNN model in which input and output feature maps can fit in on-chip buffers. This enables the use of interchangeable input and output buffers that eliminates the need to store any intermediate results in external memory. However, this feature is inapplicable to at least the first layers of the other CNN models used in our study as their feature maps exceed the capacity of on-chip buffers. For this case, we use a scheme similar to that of the ASU-like CA to tile input and output feature maps and store intermediate results in off-chip memory. For layers that have small enough feature maps, we maintain the double buffering technique to eliminate data transfers from and to the external memory. We also carried out an experiment in which we increased the size of stream buffers such that more layers can make use of the double buffering technique. However, this resulted in degrading the maximum operating frequency of the design, leading to a net loss in performance, and therefore we decided to keep the sizes of the stream buffers the same as that used for the AlexNet model. In addition, we implemented BNORM and ELTWISE blocks for this CA that were not part of the original implementation in Reference [2].

3.3 Chain-NN-like CA

This CA was proposed in Reference [50] by Wang et al. from Waseda University. It was implemented as an ASIC (using TSMC 28nm process technology), specifically for accelerating the CONV layers of AlexNet. It uses a dual-channel 1D systolic chain of N_{chain} PEs to flexibly compute 2D convolutions of any kernel size. Each PE has a multiplier and a set of input multiplexers controlled

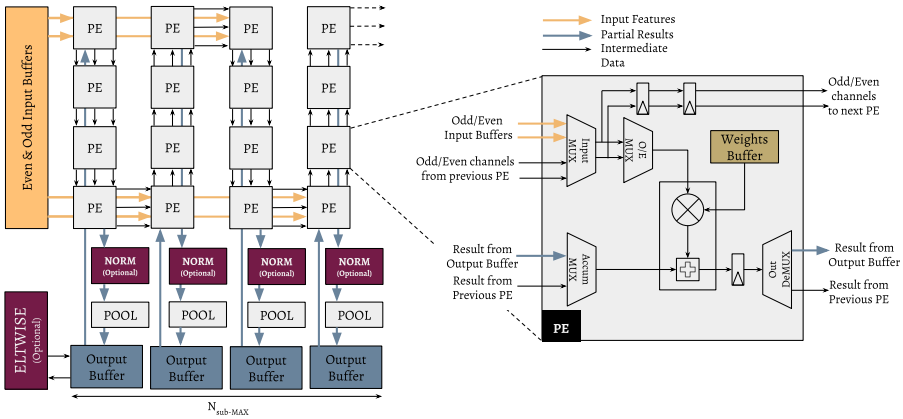

Fig. 5. Chain-NN CA with ($N_{chain} = 16, K = 2, N_{sub} = 4$) and the internal architecture of each PE.

by complex central control logic that splits the PE chain into N_{sub} smaller sub-chains according to the size of the convolution kernel, where $N_{sub} = N_{chain}/(K \times K)$, as shown in Figure 5. We implemented this CA for our study, because, despite being originally proposed as an ASIC implementation, it has compelling resemblance to FPGA architectures that can efficiently implement 1D systolic chains of multipliers using the on-chip hard DSP blocks.

This CA separates the input feature maps into odd and even columns and uses two separate input buffers to store them. The two input buffers supply inputs to the first PE of every sub-chain (i.e., the first of every 9, 25, and 121 PEs to implement convolutions of kernel size 3×3, 5×5, and 11×11, respectively). There are $N_{sub-MAX}$ output buffers, each of which stores the outputs produced by a sub-chain where $N_{sub-MAX} = N_{chain}/(3 \times 3)$, since that 3×3 is the smallest kernel size used in AlexNet CONV layers. Each PE in the chain contains both a multiplier and a small local buffer of 512 words for storing the weights needed for the computations performed in this specific PE. The largest Arria 10 FPGA contains 3,136 multipliers but only 2,713 BRAMs. We therefore implement the local weight buffers in the soft logic (MLABs) and use the BRAMs to implement input and output feature buffers.

Figure 5 shows the details of the dual-channel PE used in the 1D systolic chain of this CA. The two input channels receive odd-column and even-column input features either from the odd and even input buffers, respectively, if it is the first PE of a sub-chain, or from the channels of the previous PE, otherwise through an input multiplexer. The odd-column and even-column inputs propagate to the next PE after two cycles due to the systolic registers added to the chain. Another odd/even multiplexer chooses the MAC unit input to be either the odd-column or even-column input feature. The MAC unit multiplies the chosen input with the corresponding weight from the local weight buffer and adds the output to the previous partial result from the output buffers if it is the first PE of a sub-chain or to the output of the previous PE otherwise. For a CONV layer with kernel size K, the convolution engine produces the partial results of a tile of size $N_{OX} \times K$ across N_{sub} output feature maps. Then this is repeated N_{IM} times (Loop 4 in Algorithm 1) with the partial results used as inputs to the MAC units of the first PE in each sub-chain to produce the final results of this tile. The next tile of the same N_{sub} output feature maps is processed in the same manner (Loop 3) until the whole $N_{OX} \times N_{OY} \times N_{sub}$ are computed after which the computations of the next N_{sub} output feature maps (Loop 1) starts.

The selection lines for the input multiplexer and output de-multiplexer of each PE are generated by a central control unit and are dynamically changed after each CONV layer according to the

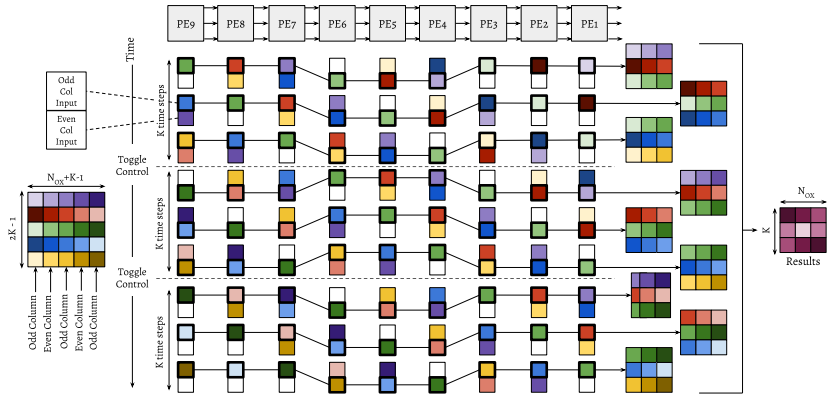

Fig. 6. Odd-column and even-column input selection schemes for $N_{OX} = 3$ and $K = 3$.

layer's kernel size. The control logic to choose between odd-column and even-column inputs is explained in Figure 6, which shows, as an example, a sub-chain of 9 PEs in the case of a CONV layer with $K = 3$ and $N_{OX} = 3$. To compute a tile of size $N_{OX} \times K$ outputs, it requires an input tile of size $(N_{OX} + K - 1) \times (2K - 1)$. The figure shows the inputs streamed from the input buffers to the sub-chain at every time step starting from time step 9 when the pipeline is filled. Input features from the even-column buffer lag behind those from the odd-column buffer by K cycles as shown in the first time step in Figure 6. After streaming a complete column of the input tile ($2K - 1$ input features), no new inputs are fed into the pipeline for the next time step after which features from the next column of same type (odd or even) are fed into the sub-chain. The thick boxes in Figure 6 show the odd/even selection for each PE in every time step. At any time step, the input selections alternate between odd and even for every K PEs in the sub-chain. After every K time steps the selections are toggled to form all the convolution windows required.

Extensions: Since it was originally proposed as an ASIC architecture only for CONV layers, we migrated and optimized this CA for FPGAs and added POOL, LRN, BNORM and ELTWISE blocks that were not part of the original implementation in Reference [50]. The POOL block buffers the final results of the N_{sub} output feature maps until a pooling window is ready to be computed. The LRN block operates on results of K_N adjacent maps and the BNORM and ELTWISE blocks operate on single results separately so their integration to this CA was straightforward. Since the other two CAs compute both CONV and FC layers using the same hardware, to provide a fair comparison, we extended this CA by mapping both the 1×1 CONV layers used in ResNet-50 and the FC layers to its convolution engine instead of implementing a dedicated engine for those layers. Unlike the conventional CONV layers, each output feature in this layers is the result of a dot-product of two vectors. Therefore, we use sub-chains of size 9 PEs as dot-product units that multiply and accumulate an input feature vector with N_{sub} weight vectors to produce N_{sub} partial results in parallel. The main drawbacks of this approach is that it does not exploit the dual-channel architecture and the complex control logic, since there is no need to arrange data in convolutional windows as previously explained. Also, the effective efficiency of the PEs is significantly degraded when executing these layers due to wasting the majority of cycles filling and flushing the pipeline of the systolic sub-chain to produce the result of one dot-product.

4 METHODOLOGY

We implement the three CAs described in Section 3 using parameterizable SystemVerilog, in which we specify the CA variation to be BSC, LRN, or ELT, which is the notion we will use for the rest of

Table 2. CA Parameters and Experimental Setup

Feature/Weight Precision	16-bit/8-bit fixed point
ASU-like Parameters	$P_{OX} = P_{OY} = 14, P_{OM} = 16$
Intel-DLA-like Parameters	$P_{IM} = 8, P_{OX} = 6, P'_{OX} = 4, P_{OM} = 48$
Chain-NN-like Parameters	$N_{chain} = 2904$ (LRN), $N_{chain} = 2304$ (BSC and ELT)
ASIC Process Technology	28 nm STMicroelectronics standard-cell libraries
ASIC Design Corner	worst-case, 1.0V, 125°C
ASIC CAD	Synopsys Design Compiler 2013.03 and Cadence Innovus 16
FPGA Device	20nm Intel Arria 10 GX 1150 (10AX115N2F45I1SG)
FPGA Design Corner	slowest, 0.95V, 100°C
FPGA CAD	Intel Quartus Prime 17.00

the article to refer to CAs that implement VGG-16, AlexNet, and ResNet-50 CNN models, respectively. The variations of each CA contain only the blocks required for each of their corresponding CNN models. For instance, the BSC variation will not contain LRN, BNORM, or ELTWISE blocks as there are no normalization or elementwise layers in the VGG-16 model. For all the CAs, we use 16-bit and 8-bit fixed-point features and weights, respectively. For the ASU-like and Intel-DLA-like architectures, we use the same parameters reported in References [27] and [2]. For the Chain-NN-like CA, since it was originally implemented as an ASIC, the parameters used in Reference [50] will leave most of the FPGA's DSP blocks unutilized. Therefore, we assigned the number of PEs (N_{chain}) to be the minimum value that achieves the highest performance given the available DSP block count constraint. As an example, for an Arria 10 device with 3,036 hard multipliers, in case of VGG-16 that has 3×3 CONV layers with 512 output channels, we can fit a maximum of $\lfloor 3{,}036 \div (3 \times 3) \rfloor = 337$ sub-chains that occupy 3,033 multipliers and compute this CONV layer in $\lceil 512 \div 337 \rceil = 2$ rounds. However, we can use only 2,304 hard multipliers (i.e. 256 sub-chains) instead, which computes the same layer also in 2 rounds but uses fewer DSP blocks and does not affect the performance of other layers as well. Table 2 summarizes the experimental setup and the parameters used in each CA.

We optimize the performance of the three CAs implemented on the FPGA to achieve the highest possible operating frequency for each one. We then migrate the exact same RTL implementations to ASICs using the same architecture parameters indicated in Table 2. One might argue that an optimized ASIC design can achieve higher performance by, for example, building custom highly efficient inter-PE network-on-chip such as in Reference [4] or fitting significantly more MACs on-chip [19]. However, the purpose of this study is not to benchmark FPGAs vs. ASICs in accelerating CNN inference, but rather highlight the bottlenecks of current FPGA architectures when implementing those CAs. Therefore, the ASIC implementations in this study serve as an upper-bound on the performance and area-efficiency of FPGA-optimized CNN accelerators where all the FPGA programmability has been removed. Comparing the same CAs on FPGAs and ASICs enables us to quantify the effect of FPGA programmability on the performance and area of those CAs and pinpoint the causes of this gap in current FPGA architectures; this would not be possible if we instead compared existing ASIC implementations to totally different state-of-the-art FPGA ones.

4.1 Performance Modeling

To obtain the performance results of the three CAs, we build analytical performance models based on our RTL simulations that calculate the number of cycles required for the computation of each layer as well as the time required for any necessary memory transfers of weights and features. We assume that the layout of the features and weights in the external memory is optimized for

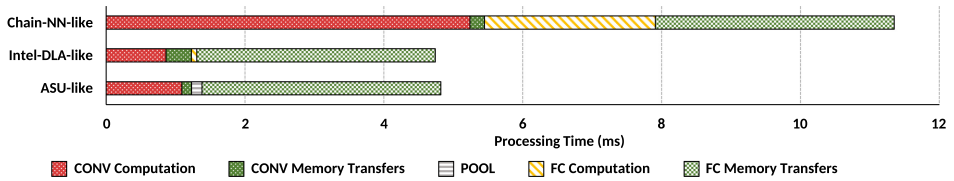

Fig. 7. Processing time breakdown of one image for the LRN variation of the three CAs.

the parallelization schemes of each CA, which allows us to utilize the burst capabilities and all the external memory bandwidth available. Given a high-level description of the CNN model, the operating frequency of the accelerator, the bit-widths of weights/features, and the available external memory bandwidth, our performance models produce the computation and memory transfer time required for each layer of the CNN. Our performance models assume either a single bank of DDR4x64 memory at 1,200MHz (for a total bandwidth of 17GB/s) or unlimited bandwidth to obtain *effective performance* and *computational performance* results, respectively. As an example, Figure 7 shows the performance model output for AlexNet on the three CAs. We then use this output to calculate the throughput in GOPS counting each MAC as two operations (i.e., a multiplication and an addition). We verified our performance models against the results reported in References [27] and [2], and we found that our models align well with the published results.

4.2 ASIC Flow

For the ASIC implementations, we use Synopsys Design Compiler 2013.03 to synthesize the CAs using 28nm STMicroelectronics standard-cell libraries; we target an unachievable clock period of 0ns to achieve the highest possible frequency and then perform area recovery by setting the maximum area to 0 and carrying out an incremental compilation. The standard-cell library comes with a wide variety of variations for different processes, voltages and operating temperatures, from which we choose the 1.0V, 125°C, and worst-case process corner for our experiments.

Memory Compiler: We use COFFE's memory compiler [46] to generate on-chip memories for our ASIC implementations. Although this memory compiler was previously used to design FPGA BRAM blocks, it is capable of designing custom memory blocks for ASICs with any required word size and depth, without any FPGA-specific circuitry. The memory cell layout as well as the verification of its area and timing results against state-of-the-art industrial and academic designs are detailed in Reference [46]. Our experiments also show that the area of memory blocks generated by COFFE's memory compiler align well with that generated by the OpenRAM [12] memory compiler for memories having different word sizes and depths. The ASIC CAs have the flexibility to implement on-chip memories of the required size and type (i.e., simple or dual port) unlike the FPGA implementations, which are constrained by the fixed size of BRAM blocks.

Place and Route Correction Factors: Using synthesis-only results for ASIC designs can overestimate frequency and underestimate area as it only predicts routing effects. However, pushing all nine designs that we implemented through multiple iterations of the place-and-route flow proved computationally infeasible due to the very high runtime of such large designs and the limited tool licenses available. However, we exploit the modular nature of the three architectures and place and route smaller instances of the CAs with fewer PEs (1/8 to 1/4 of the full size designs) to obtain correction factors for our synthesis-only results of the full-size CAs. We use Cadence Innovus 16 to place and route our designs. Our experiments show that the frequency achieved in synthesis is degraded after placement and routing by factors of 0.65, 0.74, and 0.73 for the ASU-like, Intel-DLA-like, and Chain-NN-like CAs, respectively. We observed that the area of the CAs scale linearly and

Table 3. Frequency, Effective Performance, and Image Processing for the FPGA Implementations of Different CAs

CA	ASU-like			Intel-DLA-like			Chain-NN-like		
Variation	**BSC**	**LRN**	**ELT**	**BSC**	**LRN**	**ELT**	**BSC**	**LRN**	**ELT**
Frequency (MHz)	266	216	258	339	336	345	186	197	188
Eff. Perf. (GOPS)	1,077	303	580	1,660	307	620	423	128	54
Processing Time (ms)	27.2	4.8	13.3	16.1	4.5	10.5	73.1	11.4	143.5

BSC, LRN, and ELT CA variations implement the VGG-16, AlexNet, and ResNet-50, respectively. The Intel-DLA-like results are with batch size 1.

that the correction factors are consistent across different sizes of the CAs, as we expected given the modular nature of these architectures, and this increases our confidence in the correction factors. We also needed to bloat the area of the ASIC implementations by 5% for ASU-like and 11% for both Intel-DLA-like and Chain-NN-like architectures to achieve a successful routing that met timing. We apply those correction factors to our synthesis-only results to obtain more accurate and realistic area and performance numbers for the placed and routed ASIC implementations.

4.3 FPGA Flow

For the FPGA implementations, we use Intel Quartus Prime 17.0 to synthesize, place and route the three variations of each CA for the largest and fastest speed-grade Arria 10 device. The functionality of all the designs is verified using ModelSim Intel FPGA Starter Edition 10.5b. To estimate the area occupied by the CAs on the FPGA, we first convert all the utilized resources to equivalent ALMs (eALMs). It is reported in Reference [34] that the costs of an M20K block and a DSP block in Stratix V architecture are 40 and 30 eALMs, respectively. For the Arria 10 architecture, which uses the same M20K blocks as Stratix V, we use the same cost for BRAMs; however, we account for the 10% increase in DSP block area compared to Stratix V due to adding support for floating-point arithmetic [21] leading to a DSP block cost of 33 eALMs. After that, we use the publicly available area of the 65 nm Stratix III ALM [53] and scale it down to 28nm to get an area estimate in squared millimeters that is comparable to the area of the ASIC implementations. Although the ALM architecture has only minor changes from Stratix III to Arria 10, we believe that the area results of the FPGA implementations in squared millimeters can still be optimistic, since we assume ideal scaling from 65 to 28nm. However, we are most interested in relative trends in our area gap analysis, which can help us identify the blocks that have relatively higher gap than others, rather than finding the absolute area results in squared millimeters with high accuracy.

5 RESULTS

In this section, we first compare the FPGA implementations of the different variations of the three CAs in terms of performance, resource utilization, and area breakdown. Then, we study the performance and area gap compared to the ASIC implementations. Finally, we analyze these results and suggest FPGA architectural changes to achieve more efficient CNN inference acceleration.

5.1 FPGA Results

Table 3 summarizes the maximum frequency and the processing time of one image and Figure 8(a) shows the performance results in TOPS for all variations of the three CAs. We show the performance results of the Intel-DLA-like CA in case of both processing a batch of size $B = 96$ images, similar to what was reported in Reference [2], and $B = 1$ similar to the other CAs. Besides using the Winograd transform that significantly reduces the amount of required operations and reducing external memory transfers by using double-buffered stream buffers, the Intel-DLA-like CA also

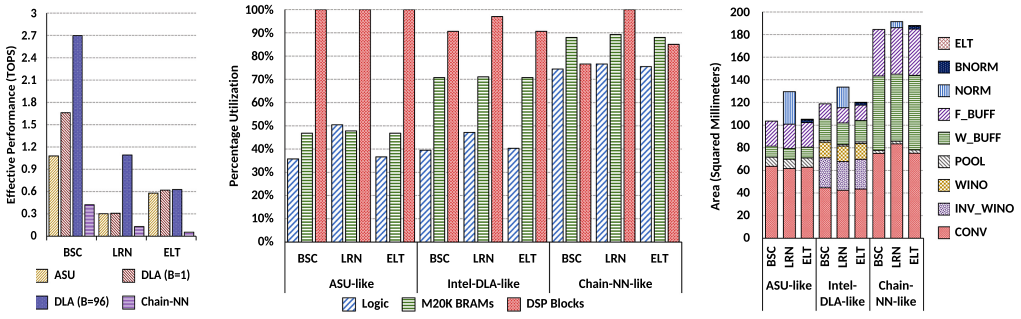

Fig. 8. FPGA Results: (a) Performance in TOPS. (b) Resource utilization. (c) Area breakdown.

achieves the highest frequency because of its pipelined daisy-chain architecture that allows an optimized placement of the PEs with less fan-out from the feature/weight buffers to the PEs when compared to the other CAs. Therefore, the Intel-DLA-like CA achieves the highest performance with 1.54× and 1.07× more TOPS than that achieved by the ASU-like CA (which uses more PEs) for the BSC and ELT and LRN variations, respectively, in case of a single image inference.

The Intel-DLA-like CA has the highest advantage over the ASU-like-CA in the BSC variation, since all the CONV layers of VGG-16 are of size 3×3 that benefit the most from the Winograd transform. This advantage decreases in the ELT variation as the ratio of 3×3 CONV layers to all layers decreases in ResNet-50, and we cannot fully make use of the double-buffering technique due to the ELTWISE layers that require storing intermediate results to the external memory. However, despite the significantly higher performance reported in Reference [2] in case of batch processing of FC layers, it achieves slightly more TOPS when compared to the ASU-like CA in case of single image inference using AlexNet. Figure 8(a) also shows that the gains from batch processing (4.2× and 1.8× more TOPS in the LRN and BSC variations, respectively) almost vanishes in ELT, since the ResNet-50 model has only one small FC layer compared to three larger FC layers in AlexNet and VGG-16.

The Chain-NN-like CA has the lowest performance results in all variations, since it runs at a significantly lower frequency than the other CAs. We believe that this is due to the high utilization of the FPGA's soft fabric (between 74%–77% as shown in Figure 8(b)), leading to physically stretched critical paths. The large fan-out from the odd/even input buffers to the first PE of all sub-chains and the large multiplexers used for selecting the outputs of sub-chains for different convolution sizes (i.e., selecting between every 9th, 25th, 49th, or 121st PE for CONV layers of size $K = 3, 5, 7$, or 11, respectively) also negatively affect the frequency. Finally, the performance of this CA is significantly degraded in FC layers and 1×1 CONV layers, since it was originally implemented for accelerating only the CONV layers as explained in Section 3.3.

Figure 8(b) shows the percentage utilization of ALMs, M20K BRAM blocks, and DSP blocks for each CA variation. The highest utilization percentage in most cases is for the DSP blocks, which are the core of the convolution engine in all CAs. The ASU-like CA uses all the 1,518 DSP blocks (3,036 18-bit multipliers) to implement the three-dimensional array of MAC units in its convolution engine and off-loads 100 MAC units to the FPGA's soft fabric. The BSC and ELT variations of the Intel-DLA-like CA use 91% of the DSP blocks, 224 of which are used for the Winograd transform and inverse transform, while 1,152 blocks are used to implement the dot product units in its PEs. In addition, its LRN variation uses the remaining DSP blocks to implement some of the multiplication operations of the LRN layers. The Chain-NN CA uses significantly more soft logic, because it

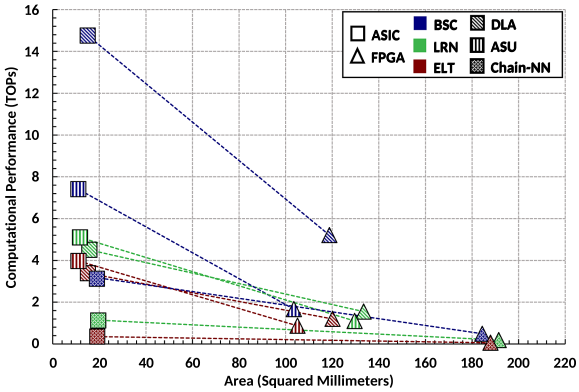

Fig. 9. Area and performance gaps.

Table 4. Summary of Area and Performance Ratios

Var	CA	AR[1]	CPR[2]	EPR[3]
BSC	ASU	9.38	4.44	2.09
	Intel-DLA	7.87	2.83	1.26
	Chain-NN	8.16	6.33	3.63
LRN	ASU	11.02	4.63	1.25
	Intel-DLA	8.48	2.91	1.15
	Chain-NN	8.38	5.98	2.29
ELT	ASU	9.48	4.58	2.08
	Intel-DLA	7.93	2.82	1.5
	Chain-NN	8.27	6.26	5.35
Geomean		8.73	4.31	2.01

[1]Area Ratio (FPGA/ASIC).
[2]Computational Performance Ratio (ASIC/FPGA).
[3]Effective Performance Ratio (ASIC/FPGA).

implements the weight buffers as distributed memories in MLABs. In Figure 8(c), we show the area in squared millimeters estimated according the methodology of Section 4.3 and its breakdown for all the CAs. With the exception of the Chain-NN-like CA that uses a significant amount of the soft fabric to implement weight buffers, the area of the two other CAs is dominated by the computational blocks such as the convolution, pooling and normalization blocks. In the Intel-DLA-like CA, the Winograd transform and inverse transform blocks contribute to the total area by 29–33%, which is almost as expensive as the convolution engine, which consumes 32–37% of the total area.

5.2 Performance Gap

Figure 9 illustrates the area and computational performance gap between the FPGA and ASIC implementations of the three variations of each CA. The FPGA implementations are represented as triangles while the ASIC implementations are represented as squares. The colors and patterns of the data points represent the variation and the CA, respectively, and the dotted lines connect each FPGA implementation to its ASIC counterpart. The closer the data point is to the upper left corner of the graph, the better it is as it will have smaller area and higher performance. Table 4 summarizes the FPGA-to-ASIC area ratios as well as the computational performance and effective performance ASIC-to-FPGA ratio for each CA variation. The computational performance ratio (CPR) represents the performance gap between the FPGA and ASIC implementations assuming infinite external memory bandwidth. However, the effective performance ratio (EPR) represents the performance gap assuming a single-bank external memory interface as specified previously. We believe that the computational performance ratio better captures the cost of FPGA programmability and its effect on the computational core performance of the three CAs as it is not limited by a relatively low-performance external memory interface. The values of EPR are less than those of the CPR as shown in Table 4 due to the external memory bandwidth constraints. As the performance of the computational engine increases, the CAs can use multiple DDR memory banks or high-bandwidth memory to enhance the overall performance. Therefore, EPR and CPR represent lower and upper bounds for design points using different external memory systems. Since the main focus of this work is studying the computational gap caused by the FPGA programmability, we believe that the CPR is the more important metric.

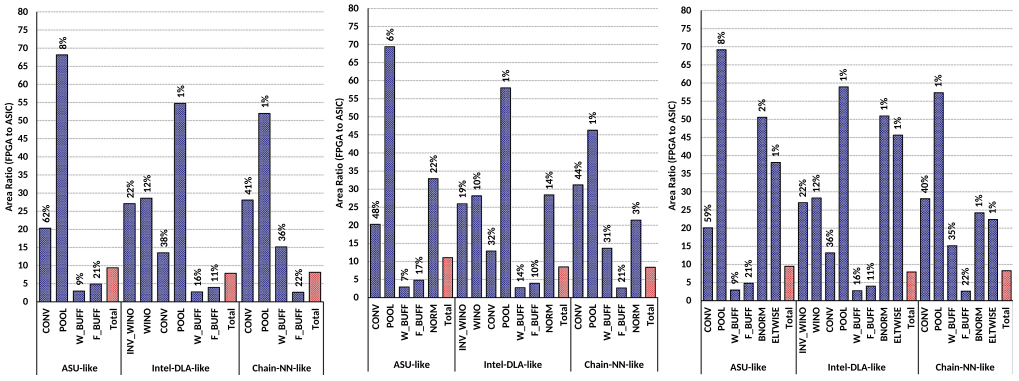

Fig. 10. Area gap between FPGA and ASIC implementations for different blocks of: (a) BSC, (b) LRN, and (c) ELT. The percentages represent the contribution of each component to the total area of the FPGA implementation.

Interestingly, the computational performance gap is not consistent among different CAs; however different variations of the same CA have similar gap results. The Intel-DLA-like CA has the smallest ASIC-to-FPGA computational performance ratio (≈2.9) compared to the ASU-like and Chain-NN-like CAs (≈4.6 and 6.2, respectively). We believe that the reason is that the Intel-DLA-like CA has a modular daisy-chain architecture, which is more routing-friendly and benefits the FPGA implementation more than the ASIC one due to the relatively slow speed of FPGA routing.

5.3 Area Gap

On average, the FPGA implementations have 8.7× larger area than their ASIC counterparts and the gap is, in contrast to the performance gap, fairly similar across different variations of the three CAs. To understand the reasons for this gap, Figures 10(a), 10(b), and 10(c) illustrate the area ratio of different components in the FPGA implementations to those in the ASIC implementations for the BSC, LRN, and ELT variations, respectively. The percentages written above the bars represent the area breakdown of each FPGA implementation into different components and hence indicate the contribution of each component to the overall area gap. We notice that the convolution engine, which has the largest contribution to total area (up to 60% in some cases) and thus the strongest impact on the total area gap, has an FPGA-to-ASIC area area ratio ranging from 13 to 31 for different variations of the three CAs. The Intel-DLA-like uses Winograd transform to significantly reduce MAC operations in convolution, which costs almost the same area as the convolution engine in the FPGA implementation. However, the Winograd transform and inverse transform blocks in this CA have FPGA-to-ASIC area ratios of 28 and 26, respectively, which are almost twice the area gap for the convolution engine, since they contain a large number of multi-input adders implemented in the FPGA's soft fabric compared to the convolution engine, which is mostly implemented in hard DSP blocks. The smallest area gap is in the feature and weight buffers, since the RAMs in the FPGA and the ASIC implementations are both custom SRAM blocks. However, the buffers area ratios are still significant (≈3–5) because of the area overhead of the programmable routing in BRAM tiles as well as the underutilization of some of the M20K blocks on the FPGA, whereas in the ASIC implementations, we use memories with the exact required sizes. The NORM block has an area ratio of 32 and 28 and consumes 22% and 14% of the total area in ASU-like and Intel-DLA-like CAs,

respectively, since it is a heavily arithmetic block and is mostly implemented in the soft fabric. However, it only consumes 3% of the total area in the Chain-NN-like CA, which produces outputs in one dimension only and therefore does not normalize output features at different locations in parallel. The POOL, ELTWISE and BNORM blocks have large area ratios, however they have small overall areas and hence limited impact on the total gap.

An interesting observation is that the area gap in the convolution engine of the Intel-DLA-like CA is significantly less than that of the other two CAs: an area ratio of 13 compared to 20 and 29 in ASU-like and Chain-NN-like CAs, respectively. This is because the Intel-DLA-like CA uses the hard adders in the DSP blocks to implement its dot-product unit, while the other two CAs pay for the area of the complete DSP block on the FPGA but only make use of the multipliers inside it and thus have a higher area gap compared to their ASIC counterparts. This observation motivates the investigation of new DSP block designs that could bring more of the convolution engine functionality inside the hard DSP block. For instance, the ASU-like CA needs two separate accumulators for the two independent 18-bit multipliers, which is not supported in current DSP blocks. Hence, the DSP block accumulators are wasted and soft logic is used to implement the accumulators. The convolution engine of the Chain-NN-like CA has the highest area gap as it implements input multiplexing, accumulation, and output de-multiplexing in the soft fabric.

5.4 Architectural Insights

Based on the results of Sections 5.1 and 5.2, we can draw several architectural insights:

- According to the resource utilization results in Figure 8(b), the limiting factor is the DSP block count available on-chip, with close to 100% resource utilization in most cases. One direct approach to gain higher performance is adding more DSP blocks to current FPGAs, especially given that a DSP-focused device spends only 5% of its core area on DSP blocks [21]. This requires a careful architectural study to determine the optimal ratio and area distribution between DSPs, BRAMs, and ALMs for DL-tuned FPGAs that are still flexible enough and suitable for other applications as well. These architectural explorations require a suite of DL benchmark circuits such as the one we developed in this work, and which we plan to expand and open-source in future work.
- As shown in Figure 10, the area gap of the convolution engine of the Intel-like-DLA CA is significantly less than that of the other two CAs, since it makes better use of the DSP block available functionalities such as the internal adders and hard cascade chains. By looking at the ASIC area breakdown of the convolution engine, we can see that about 72% of the logic in the convolution engine of the Intel-DLA-like CA was implemented inside hard DSP blocks on the FPGA compared to only 32% and 35% in the ASU-like and Chain-NN-like CAs, respectively, and the rest is implemented in the soft fabric. We believe that small changes to the DSP block architecture could capture more of the convolution engine hardware inside the hard circuitry of the DSP block. For example, adding an operation mode that configures the two internal adders as independent accumulators for two independent 18-bit MACs (such as in the ASU-like CA) or having a small circular shift register accumulator for interleaving dot-product operations (as in the Intel-DLA-like CA) would save soft logic. Neither of the DSP block enhancements would add much logic to the block, nor would they require more block routing ports (inputs and outputs) and, therefore, the DSP block area increase would be minimal. To increase the DSP block count on-chip, as mentioned in our first suggestion, we not only wish to avoid significant block area increase, but also remove DSP block functionalities that are unnecessary for DL applications and would not cause severe performance degradation when implemented in the soft fabric. For example, removing the

built-in constant coefficient banks in the Arria 10 DSP blocks should be evaluated as they are not usable by any of our CAs.

- In this study, we used 16- and 8-bit fixed-point precision for features and weights, respectively, in all CAs to ensure fair comparisons. However, the most suitable precision for CNN inference is debatable and varies widely in the literature from single-precision floating-point down to ternary and binary [28]. Currently, DSP blocks from Intel and Xilinx support a limited number of precisions. For instance, a DSP block in Intel Arria 10, and similarly Stratix 10, FPGAs supports two 18-bit, one 27-bit, or one single-precision floating-point multiplication. However, a DSP slice in Xilinx Virtex Ultrascale FPGAs supports one 27×18 multiplication. Designers can sometimes fit more low-precision multiplies that match certain patterns using clever tricks such as performing two 8-bit multiplies that share one operand using a single Xilinx DSP slice [8]. Even with these operand packing tricks, using lower precision leaves a portion of the DSP block logic idle. We can avoid this by designing DSP blocks that natively support low-precision multiplications and reuse routing ports and multiplier sub-arrays to keep the area overhead minimal.
- When implementing the three CAs, we noticed that the required on-chip buffers are either deep central buffers for input and output features or smaller and more distributed buffers for the weights. When we tried to extend the double-buffering technique used in the Intel-DLA-like CA to more layers of models larger than AlexNet by implementing deeper stream buffers, it resulted in a net performance degradation as the operating frequency dropped significantly due to depth stitching of M20K BRAMs to implement those deep buffers. However, when implementing the small weight buffers of the Chain-NN-like CA in MLABs, the high utilization of the soft fabric also resulted in lower operating frequency. This observation indicates that having only M20K BRAMs and MLABs to implement on-chip memories might not be a good fit for DL acceleration on FPGAs. This also requires a more detailed architectural study to determine the best size and ratio of on-chip BRAMs and their effect on the overall performance using DL-representative benchmarks, and we believe our parameterized CAs can form the start of this benchmark set. In addition, the memory-richness of FPGAs can be enhanced by employing emerging technologies such as Magnetic Tunneling Junction memories, which can provide bigger yet more dense BRAMs for memory-intensive applications as shown in Reference [54].

6 CONCLUSION

In this article, we implemented three highly optimized state-of-the-art CAs for accelerating CNN inference, which are: ASU-like, Intel-DLA-like, and Chain-NN-like CAs. We implemented three variations of each CA (BSC, LRN, and ELT) for three different CNN models (VGG-16, AlexNet, and ResNet-50, respectively) on an Intel Arria 10 FPGA device and compared them to 28nm ASIC implementations of the same CAs to quantify the programmability cost that comes with using FPGAs on the performance and area of DL accelerators. Across different variations of the three CAs, we observed a consistent area gap with an average FPGA-to-ASIC area ratio of 8.7×, to which the convolution engine contributes the most with area ratios ranging from 13 to 31 for different CAs. The performance gap, unlike the area gap, varies significantly across different CAs. The computational performance of the ASIC implementations is 2.8× to 6.3× faster than that of the FPGA implementations when assuming infinite external memory bandwidth. We find that the Intel-DLA-like CA has the smallest performance gap compared to its ASIC counterpart indicating that focusing on modular and routing-friendly designs is of great importance for building efficient FPGA-based DL accelerators. Finally, we suggest several FPGA DSP and RAM architecture changes for future

work that could reduce the area and performance gaps and enable more efficient DL acceleration on FPGAs.

ACKNOWLEDGMENTS

The authors thank Martin Langhammer, Debbie Marr, and Eriko Nurvitadhi for helpful discussions, as well as Huawei, Intel, and NSERC for funding support.

REFERENCES

[1] M. Abadi et al. 2016. TensorFlow: A system for large-scale machine learning. In *Proceedings of the OSDI*. 265–283.
[2] U. Aydonat et al. 2017. An OpenCL (TM) deep learning accelerator on Arria 10. In *Proceedings of the FPGA*. 55–64.
[3] Y. Chen et al. 2014. DaDianNao: A machine-learning supercomputer. In *Proceedings of the MICRO*. 609–622.
[4] Y. Chen et al. 2017. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. In *Proceedings of the JSSC*, Vol. 52. 127–138.
[5] S. Chetlur et al. 2014. CuDNN: Efficient primitives for deep learning. *arXiv:1410.0759*.
[6] E. Chung and J. Fowers. 2017. Accelerating persistent neural networks at datacenter scale. In *Proceedings of the HOT CHIPS*, Vol. 29.
[7] F. Colombo et al. 2017. Deep artificial composer: A creative neural network model for automated melody generation. In *Proceedings of the EvoMUSART*. 81–96.
[8] Y. Fu et al. 2016. Deep learning with INT8 optimization on Xilinx devices. In *white paper of Xilinx*.
[9] L. Gatys et al. 2015. A neural algorithm of artistic style. *arXiv:1508.06576*.
[10] A. Graves et al. 2013. Speech recognition with deep recurrent neural networks. In *Proceedings of the ICASSP*. 6645–6649.
[11] Y. Guan et al. 2017. FP-DNN: An automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates. In *Proceedings of the FCCM*. 152–159.
[12] Matthew R. Guthaus et al. 2016. OpenRAM: An open-source memory compiler. In *Proceedings of the ICCAD*.
[13] P. Gysel et al. 2016. Hardware-oriented approximation of convolutional neural networks. *arXiv:1604.03168*.
[14] K. He et al. 2015. Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification. In *Proceedings of the ICCV*. 1026–1034.
[15] K. He et al. 2016. Deep residual learning for image recognition. In *Proceedings of the CVPR*. 770–778.
[16] S. Herculano-Houzel. 2009. The human brain in numbers: A linearly scaled-up primate brain. In *Frontiers in Human Neuroscience*, Vol. 3.
[17] S. Ioffe and C. Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the ICML*. 448–456.
[18] Y. Jia et al. 2014. Caffe: Convolutional architecture for fast feature embedding. *arXiv:1408.5093*.
[19] N. Jouppi et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the ISCA*. 1–12.
[20] A. Krizhevsky et al. 2012. ImageNet classification with deep convolutional neural networks. In *Proceedings of the NIPS*. 1097–1105.
[21] M. Langhammer and B. Pasca. 2015. Floating-point DSP block architecture for FPGAs. In *Proceedings of the FPGA*. 117–125.
[22] A. Lavin and S. Gray. 2016. Fast algorithms for convolutional neural networks. In *Proceedings of the CVPR*. 4013–4021.
[23] Z. Liu et al. 2016. Automatic code generation of convolutional neural networks in FPGA implementation. In *Proceedings of the FPT*. 61–68.
[24] L. Lu et al. 2017. Evaluating fast algorithms for convolutional neural networks on FPGAs. In *Proceedings of the FCCM*. 101–108.
[25] Y. Ma et al. 2016. Scalable and modularized RTL compilation of convolutional neural networks onto FPGA. In *Proceedings of the FPL*. 1–8.
[26] Y. Ma et al. 2017. An automatic RTL compiler for high-throughput FPGA implementation of diverse deep convolutional neural networks. In *Proceedings of the FPL*. 1–8.
[27] Y. Ma et al. 2017. Optimizing loop operation and dataflow in FPGA acceleration of deep convolutional neural networks. In *Proceedings of the FPGA*. 45–54.
[28] A. Mishra et al. 2017. WRPN: Wide reduced-precision networks. *arXiv:1709.01134*.
[29] E. Nurvitadhi et al. 2016. Accelerating binarized neural networks: Comparison of FPGA, CPU, GPU, and ASIC. In *Proceedings of the FPT*. 77–84.
[30] K. Ovtcharov et al. 2015. Accelerating deep convolutional neural networks using specialized hardware. In *Microsoft Research Whitepaper*, Vol. 2.

[31] A. Prost-Boucle et al. 2017. Scalable high-performance architecture for convolutional ternary neural networks on FPGA. In *Proceedings of the FPL*. 1–7.
[32] A. Putnam et al. 2014. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceedings of the ISCA*. 13–24.
[33] J. Qiu et al. 2016. Going deeper with embedded FPGA platform for convolutional neural network. In *Proceedings of the FPGA*. 26–35.
[34] R. Rashid et al. 2014. Comparing performance, productivity and scalability of the TILT overlay processor to OpenCL HLS. In *Proceedings of the FPT*. 20–27.
[35] D. E. Rumelhart et al. 1985. *Learning Internal Representations by Error Propagation*. Technical Report.
[36] O. Russakovsky et al. 2015. Imagenet large scale visual recognition challenge. In *Proceedings of the IJCV*, Vol. 115. 211–252.
[37] H. Sharma et al. 2016. From high-level deep neural models to FPGAs. In *Proceedings of the MICRO*. 1–12.
[38] F. Shen et al. 2016. Weighted residuals for very deep networks. In *Proceedings of the ICSAI*. 936–941.
[39] Y. Shen et al. 2016. Overcoming resource underutilization in spatial CNN accelerators. In *Proceedings of the FPL*. 1–4.
[40] Y. Shen et al. 2017. Maximizing CNN accelerator efficiency through resource partitioning. In *Proceedings of the ISCA*. 535–547.
[41] D. Silver et al. 2017. Mastering the game of go without human knowledge. In *Nature*, Vol. 550. 354–359.
[42] N. Suda et al. 2016. Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks. In *Proceedings of the FPGA*. 16–25.
[43] A. Suleiman et al. 2017. Towards closing the energy Gap between HOG and CNN features for embedded vision. *arXiv:1703.05853*.
[44] I. Sutskever et al. 2014. Sequence to sequence learning with neural networks. In *Proceedings of the NIPS*. 3104–3112.
[45] C. Szegedy et al. 2015. Going deeper with convolutions. In *Proceedings of the CVPR*.
[46] Kosuke Tatsumura et al. 2016. High density, low energy, magnetic tunnel junction based block RAMs for memory-rich FPGAs. In *Proceedings of the FPT*. 4–11.
[47] Y. Umuroglu et al. 2017. FINN: A framework for fast, scalable binarized neural network inference. In *Proceedings of the FPGA*. 65–74.
[48] S. Venieris and C. Bouganis. 2016. fpgaConvNet: A framework for mapping convolutional neural networks on FPGAs. In *Proceedings of the FCCM*. 40–47.
[49] G. Venkatesh et al. 2017. Accelerating deep convolutional networks using low-precision and sparsity. In *Proceedings of the ICASSP*. 2861–2865.
[50] S. Wang et al. 2017. Chain-NN: An energy-efficient 1D chain architecture for accelerating deep convolutional neural networks. In *Proceedings of the DATE*. 1032–1037.
[51] Y. Wang et al. 2016. DeepBurning: Automatic generation of FPGA-based learning accelerators for the neural network family. In *Proceedings of the DAC*. 1–6.
[52] X. Wei et al. 2017. Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs. In *Proceedings of the DAC*. 1–6.
[53] H. Wong et al. 2011. Comparing FPGA vs. custom CMOS and the impact on processor microarchitecture. In *Proceedings of the FPGA*. 5–14.
[54] S. Yazdanshenas et al. 2017. Don't forget the memory: Automatic block RAM modelling, optimization, and architecture exploration. In *Proceedings of the FPGA*. 115–124.
[55] C. Zhang et al. 2015. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *Proceedings of the FPGA*. 161–170.
[56] C. Zhang et al. 2016. Energy-efficient CNN implementation on a deeply pipelined FPGA cluster. In *Proceedings of the ISLPED*. 326–331.
[57] C. Zhang and V. Prasanna. 2017. Frequency domain acceleration of convolutional neural networks on CPU-FPGA shared memory system. In *Proceedings of the FPGA*. 35–44.

Received December 2017; revised April 2018; accepted July 2018