Automatic Compilation of Diverse CNNs Onto High-Performance FPGA Accelerators

Yufei Ma, *Student Member, IEEE*, Yu Cao, *Fellow, IEEE*, Sarma Vrudhula, *Fellow, IEEE*, and Jae-Sun Seo, *Senior Member, IEEE*

***Abstract*—A broad range of applications are increasingly benefiting from the rapid and flourishing development of convolutional neural networks (CNNs). The FPGA-based CNN inference accelerator is gaining popularity due to its high-performance and low-power as well as FPGA's conventional advantage of reconfigurability and flexibility. Without a general compiler to automate the implementation, however, significant efforts and expertise are still required to customize the design for each CNN model. In this paper, we present an register-transfer level (RTL)-level CNN compiler that automatically generates customized FPGA hardware for the inference tasks of various CNNs, in order to enable high-level fast prototyping of CNNs from software to FPGA and still keep the benefits of low-level hardware optimization. First, a general-purpose library of RTL modules is developed to model different operations at each layer. The integration and dataflow of physical modules are predefined in the top-level system template and reconfigured during compilation for a given CNN algorithm. The runtime control of layer-by-layer sequential computation is managed by the proposed execution schedule so that even highly irregular and complex network topology, e.g., GoogLeNet and ResNet, can be compiled. The proposed methodology is demonstrated with various CNN algorithms, e.g., NiN, VGG, GoogLeNet, and ResNet, on two standalone Intel FPGAs, Arria 10, and Stratix 10, achieving end-to-end inference throughputs of 969 GOPS and 1604 GOPS, respectively, with batch size of one.**

***Index Terms*—Convolutional neural networks (CNNs), FPGA, neural network hardware.**

I. Introduction

CONVOLUTIONAL neural networks (CNNs) have become the dominant approach in many computer vision applications, such as image classification [1]–[6] and object detection [7], [8]. The large number of operations and parameters as well as the highly varying layer dimensions have challenged the real-time implementation of CNNs on FPGA-based accelerators. Constrained by the limited computing resources and costly external memory (e.g., DRAM) access, the FPGA-based CNN accelerator must fully reuse the hardware resources for different convolution layers and increase data locality to reduce the data movement and off-chip communication. Therefore, the CNN acceleration strategy, such as loop unrolling and tiling techniques, must be optimized to properly manage the parallel computation, data storage patterns in buffers, and the external memory accesses [9]–[11].

To improve their accuracy for classification and expand their domain of applications, CNNs with greater depth, new types of layers and more complex networks have been proposed. For example, the deep residual networks (ResNets) [4], [6], [12] can achieve substantially greater accuracy at the cost of having more than 1000 convolution layers (Conv) with widely differing dimensions and kernel sizes, as well as many other various types of layers. Unlike earlier CNNs, such as AlexNet [2] and VGG [3], in which the layers are strung out in a sequence, the layers in the more recent CNN algorithms, such as ResNet, GoogLeNet [5], and Inception [6], form a directed acyclic graph (DAG), as shown in Fig. 1. They have multiple parallel branches and include feedforward connections between nonadjacent layers.

All these trends, which have increased the complexity of CNN algorithms, have made it more difficult to design a general-purpose CNN hardware accelerator to efficiently map

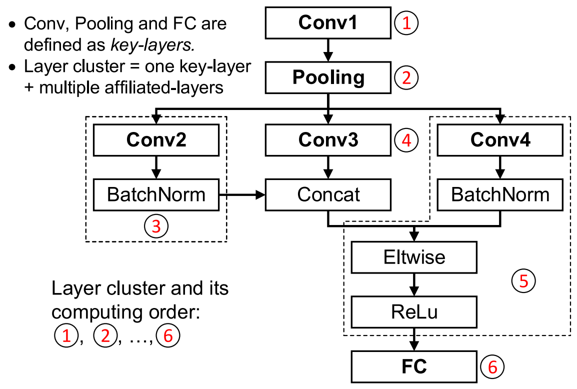

• Conv, Pooling and FC are defined as *key-layers*.
• Layer cluster = one key-layer + multiple affiliated-layers
Conv1
1
Pooling
2
Conv2
BatchNorm
3
Conv3
4
Concat
Conv4
BatchNorm
Eltwise
ReLu
5
FC
6
Layer cluster and its computing order:
1, 2, ..., 6

Fig. 1. Example of DAG form layer connections in recent CNN algorithms with multiple parallel branches involving different types of layers.

Manuscript received April 24, 2018; revised July 16, 2018 and September 27, 2018; accepted November 18, 2018. Date of publication December 4, 2018; date of current version January 18, 2020. This work was supported in part by the NSF I/UCRC Center for Embedded Systems through NSF under Grant 1230401, Grant 1237856, Grant 1701241, Grant 1361926, and Grant 1535669, in part by NSF under Grant 1652866 and Grant 1715443, in part by Intel Labs, and in part by C-BRIC, one of six centers in JUMP, an SRC program sponsored by DARPA. This paper was recommended by Associate Editor W. Zhang. *(Corresponding author: Yufei Ma.)*

Y. Ma, Y. Cao, and J.-S. Seo are with the School of Electrical, Computer and Energy Engineering, Arizona State University, Tempe, AZ 85287 USA (e-mail: yufeima@asu.edu; yu.cao@asu.edu; jaesun.seo@asu.edu).

S. Vrudhula is with the School of Computing, Informatics, Decision Systems Engineering, Arizona State University, Tempe, AZ 85287 USA (e-mail: vrudhula@asu.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2018.2884972

a diverse range of CNN algorithms. One approach is to undertake custom hardware design at the register-transfer level (RTL) for each specific CNN. Experience has shown that such an approach requires detailed knowledge of both the CNN algorithm and the FPGA architecture, and many months of design effort involving numerous design iterations. As the size and complexity of CNN algorithms continue to increase, the custom approach becomes increasingly impractical, and automated approach is essential.

In this paper, the design of a library-based CNN RTL compiler is described. Its overall structure and process flow is depicted in Fig. 2. The inputs that need to be supplied are the abstract description of the CNN model and the set of design variables that characterize the usage of hardware resources. The compiler enables fast and automatic mapping of various deep CNN algorithms from software deep learning frameworks, e.g., Caffe [13], onto FPGAs. It exploits the reconfigurability of FPGAs and the fine-grain optimization that is possible with an RTL description. As CNNs are assembled by composing iterative computing primitives or layers, scalable RTL building block modules are designed for different types of layers and reused by different CNNs. The RTL compiler configures these modules with CNN parameters, and scales the sizes of processing engines (PEs) and on-chip buffers based on the user-specified hardware design variables. The result is a resource efficient and high performance FPGA implementation.

This paper significantly extends the authors' earlier work [14] by including: 1) dual buffer structure to further improve the accelerator performance; 2) scalability demonstration of the compiler on the latest Stratix 10 FPGA with more hardware resources; 3) newly added Concat layer to support GoogLeNet and Inception; and 4) integration of batch normalization layer with its precedent convolution layer during inference. The main contributions of this paper and of [14] are as follows.

1) An execution schedule derived from the CNN structure that maps a wide range of CNN algorithms onto a system-level reconfigurable FPGA architecture.
2) A user-friendly and high-level compiler that automatically configures the FPGA-based accelerator for various large-scale CNN algorithms with user-specified hardware resource constraints, such as computing parallelism and buffer usage, targeting FPGA platforms with different amount of hardware resources.
3) An RTL module library that can accommodate different types of layers with manually coded Verilog templates. The modules are designed based on the optimized acceleration strategy in [9], which defines the parallel computation, data movement, and memory access. This library has been designed to allow incorporation of new layers or operations for future deep learning algorithms.
4) Performance analysis based on the *roofline model* approach [10] to identify the performance bottleneck limited by the computation resources and memory bandwidth, which leads to potential benefits of changing various parameters and optimization steps.

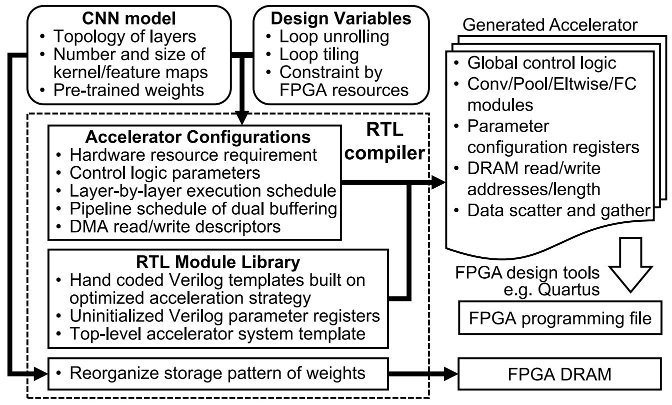

CNN model
• Topology of layers
• Number and size of kernel/feature maps
• Pre-trained weights
Design Variables
• Loop unrolling
• Loop tiling
• Constraint by FPGA resources
Generated Accelerator
• Global control logic
• Conv/Pool/Eltwise/FC modules
• Parameter configuration registers
• DRAM read/write addresses/length
• Data scatter and gather
Accelerator Configurations
• Hardware resource requirement
• Control logic parameters
• Layer-by-layer execution schedule
• Pipeline schedule of dual buffering
• DMA read/write descriptors
RTL compiler
RTL Module Library
• Hand coded Verilog templates built on optimized acceleration strategy
• Uninitialized Verilog parameter registers
• Top-level accelerator system template
FPGA design tools e.g. Quartus
FPGA programming file
• Reorganize storage pattern of weights
FPGA DRAM

Fig. 2. Overall compilation flow of the proposed CNN RTL compiler: the CNN algorithm information and the hardware design variables are the inputs; the hardware resource usage and the execution schedule of the FPGA accelerator are configured for the given CNN model; the RTL module library defines the computation pattern and dataflow of different types of layers with parameterized Verilog templates; and the pretrained CNN weights are reorganized and loaded into the external memory.

The flexibility of the proposed compilation methodology is validated by implementing the inference task of both conventional CNNs: NiN [15] and VGG-16 [3]; and the more complex DAG networks: GoogLeNet [5] and ResNets [4] with 50 and 152 convolution layers, respectively. The accelerator is demonstrated on two standalone Intel FPGAs, Arria-10, and Stratix-10, achieving end-to-end inference throughputs of 968 GOPS on Arria-10 and 1604 GOPS on Stratix-10 with batch size of one to minimize the inference latency for real-time applications.

II. Overview of Proposed CNN RTL Compiler

The dimensions and connections of CNN layers and pretrained kernel weights are obtained from Caffe [13], and provided as inputs to the CNN compiler. The various dimensional parameters of the CNN algorithm and the accelerator design variables, e.g., loop unrolling and tiling sizes as shown in Fig. 3 (described in detail in Section III), can be tuned by the user to balance the performance and required hardware resources. Then, a layer-by-layer execution schedule [see Fig. 4(a) and (b)] is generated from the CNN graph representation. The execution schedule is translated into the global control logic on the FPGA, and it also determines the order of the reads and writes of certain kernel weights or pixels from different layers that are stored in external memory. The associated read and write addresses are generated and sorted to control the transactions between external and on-chip memories.

The RTL module library consists of manually coded Verilog templates describing the computations and dataflow of various types of layers. The templates are built on the optimized CNN acceleration strategy described in [9]. That strategy is designed to minimize the memory access and data movements while maximizing the resource utilization. The Verilog parameters that determine the size of PEs and buffers are configured based on the design variables. The parameters for runtime control are initialized by compiler and stored in configuration registers.

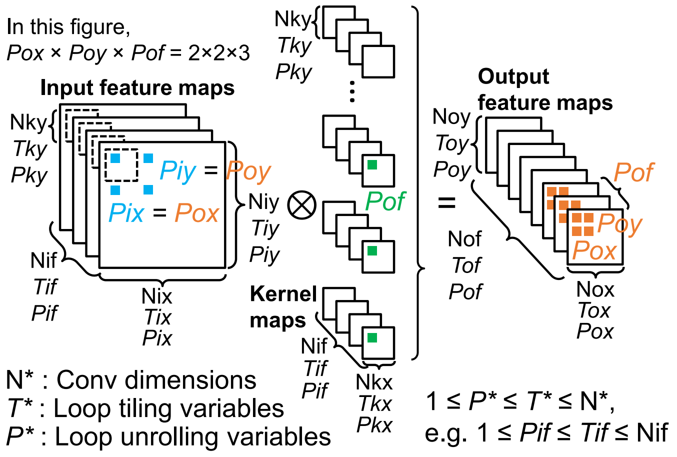


Fig. 3. Convolution loop dimensions (N^*) and accelerator design variables of loop unrolling (P^*) and loop tiliing (T^*). Type: i: input; o: output; k: kernel; f: feature.

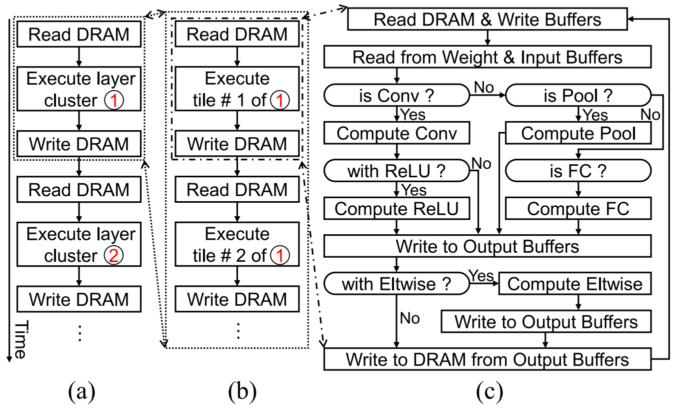


Fig. 4. Execution schedule is designed to handle different CNN topology. (a) Layer-by-layer execution. (b) Intertile execution inside one layer. (c) Intratile process inside one tile [14].

The intratile execution flow of layers, as shown in Fig. 4(c), is predefined in the templates and can be customized by the compiler to enable execution of certain layers during run time. The top-level accelerator system template, shown in Fig. 5, integrates these modules with the reconfigurable dataflow, where only the required computing modules are compiled for a given CNN model, bypassing the unused modules.

III. Acceleration of Convolution Loops

A. Convolution Loop Optimization and Design Variables

Convolution involves 3-D multiply and accumulate operations (MAC) of input feature maps and kernel weights as illustrated in Fig. 3, where the parameters (N^*) prefixed with capital N denote the algorithm-defined dimensions of feature and kernel maps of one Conv layer. Since convolution dominates the CNN operations, the acceleration strategy of convolution loops dramatically impacts the parallel computation efficiency and memory access requirements. Therefore, we employ the loop optimization techniques in [9] to customize the convolution computation and communication patterns. Loop unrolling design variables (P^*) determine the degree of parallelism of certain convolution loops, and thus the required size and architecture of PEs. Loop tiling increases the data locality by dividing the entire data of one layer into multiple tiles, which can be fit into the on-chip buffers. The loop tiling design variables (T^*) determine the required minimum sizes of the on-chip buffers, and affect the required external memory accesses.

B. Convolution Acceleration Strategy

The design of the module templates at the RTL is based on the CNN acceleration strategy described in [9]. It achieves a uniform mapping of PEs and reduces the accelerator architecture complexity. Fig. 3 shows the dimensions of the inputs (input feature maps and kernel maps) and the output feature maps. The loop unrolling or the parallel computations are only employed within one input feature map and across multiple kernel maps. The computations shown in Fig. 3 are as follows.

1) *$Pix = Pox > 1$ and $Piy = Poy > 1$:* In every cycle, $Pix \times Piy$ number of pixels from different (x, y) locations in the same input feature map are multiplied with one identical weight.
2) *$Pof > 1$:* In every cycle, one input pixel is multiplied by Pof weights from Pof different kernel maps, which contributes to Pof pixels in Pof output feature maps.

The total number of parallel operations is $Pox \times Poy \times Pof$ with $Pkx = Pky = Pif = 1$. By this means, each PE contributes to one independent output pixel and no adder tree is needed to total the partial sums of different PEs [9]. Therefore, a PE is an MAC unit consisting of one multiplier followed by an accumulator in this paper. Both pixels and weights are reused by multiple MAC units and high degree of parallelism can be supported with large $Nox \times Noy \times Nof$. The data required to compute one final output pixel are fully buffered to minimize the partial sum storage, i.e., $Tkx = Nkx$, $Tky = Nky$, and $Tif = Nif$. We also set $Tox = Nox$ so that an entire row is buffered to improve the DRAM transactions with data from continuous addresses. Furthermore, the required buffer sizes can be changed by tuning Toy and Tof. Following the above optimized settings, different P^* and T^* design variables can be adjusted by the user to explore the best tradeoff between performance and hardware resource usage, e.g., DSP blocks and block RAMs (BRAMs), for the target FPGA platform.

IV. End-to-End CNN Accelerator

A. Layer-by-Layer Execution Schedule

In conventional CNN algorithms, different layers are connected in sequence, which allows for a straightforward layer-by-layer serial computation. The recent CNN algorithms (e.g., ResNet [4]) are DAGs, with combinations of serial and parallel branches. A reconfigurable layer-by-layer execution schedule is designed to handle the different combinations of stacked layers and the DAG as shown in Fig. 4. Therefore, the present mapping of a DAG onto an FPGA still results in a serial computation of the layers.

There are many types of layers in a CNN algorithm, and the number and order of these stacked layers could be quite different. A CNN layer that reads the DRAM for its input is referred to as a *key-layer*. Therefore, Conv, Pool, and FC are assigned as key-layers so that the computation or design variable settings between these layers are relatively independent, while all other layers are *affiliated-layers* to the key-layers. The DRAM access of an affiliated-layer can be eliminated, however its computing pattern, e.g., unrolling and tiling variables, must depend on the key-layer configuration, which hampers its design flexibility. A layer *cluster* is a subgraph of the DAG that consists of a key-layer and zero or more affiliated-layers. The example DAG shown in Fig. 1 has six clusters, numbered ① through ⑥. The Conv1(①), Pooling(②), and FC(⑥) layers in Fig. 1 are individual key-layers (i.e., clusters with only a key-layer) whereas cluster ⑤ has one key-layer (Conv4) and three affiliated-layers (Batchnorm, Eltwise, and ReLu). The layer-by-layer serial computation is essentially the serial execution of clusters as illustrated in Figs. 1 and 4(a). The order of computation of the clusters is set before compilation, and the only rule is to ensure that all the predecessors of any key-layer is executed prior to that key-layer.

When tiling of loops is performed, each cluster is divided into multiple tiles to fit into the on-chip buffers. This is illustrated in Fig. 4(a) and (b). As clusters may contain different kinds of layers, (e.g., layer cluster ④ in Fig. 1 does not have BatchNorm and Eltwise), a general intratile execution schedule is designed as shown in Fig. 4(c) to control whether or not a layer is executed for a specific cluster during runtime. The select signals, e.g., "is Conv?" in Fig. 4(c), are stored in the configuration registers and initialized based on the input CNN topology during compilation. If a layer does not exist in the given CNN, the select signal becomes constant to be "No". This schedule is also flexible as it allows introduction of new types of layers by the simple addition of new select signals.

Three levels of control logic, namely global, intertile, and local control logic, are required to govern the layer-by-layer, intertile, and intratile sequential execution (Fig. 4). The parameters of each layer, e.g., kernel sizes, feature map dimensions, unrolling and tiling variables, and iteration numbers, are stored in configuration registers. The global control logic keeps track of the number of executed clusters, and loads the current layer's parameters from the configuration registers into the local control logic registers. Each type of layer module has its own local control logic to perform the iterations within the layer. By this means, we can just use one set of control logic for layers with varying dimensions by initializing configuration registers for different layers during compilation.

B. Top-Level Acceleration System and Dataflow

The overall CNN acceleration system and dataflow is shown in Fig. 5, where different types of layers are modularized to establish the RTL module library. During compilation, if a certain type of layer does not exist in the given CNN model, its corresponding module will not be compiled or synthesized to save the hardware resources, and the dataflow just bypasses this module. During runtime, whether or not a layer is executed is controlled by the global control logic by asserting "start" signal to the module following the execution schedule. After receiving a "done" signal from the current layer, global control logic iterates to the next layer.

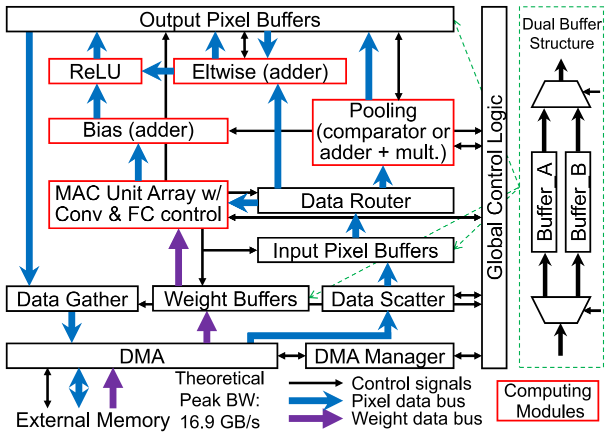


Fig. 5. Reconfigurable top-level CNN acceleration system, where the dataflow is from external memory to input buffers and then into computing modules, the results are stored in output buffers and finally sent back to external memory.

The reconfigurable computing modules, as shown by the red boxes in Fig. 5, are manually coded as maximally parameterized Verilog scripts. Each type of module template is designed to be reused by any layer of the same type, in any CNN. The varying layer sizes and loop design variables are handled by initializing the configuration registers based on the layer property. This RTL module library is designed to be easily extended with new layers for more CNN algorithms and the existing modules can also be further optimized for performance and efficiency. The detailed design of the computing modules is discussed in Section VI.

The direct memory access (DMA) engine is used to transfer data between external and on-chip memories. The data scatter module is designed to distribute a data stream from one DMA write port to multiple input buffers, and the data gather module is designed to collect data from multiple output buffers into one DMA read port. The detailed memory system design is presented in Section V.

V. External and On-Chip Memory System

A. Storage Pattern in DRAM

Due to the limited capacity of on-chip BRAMs, both kernel weights and intermediate pixel results are stored in external memory, i.e., the DRAM, and the on-chip BRAMs are used as buffers between DRAM and PEs. The proposed storage pattern of kernel weights and intermediate pixel results in the DRAM are illustrated in Fig. 6(a). The pretrained kernel weights and the input images are loaded into DRAM before the acceleration. All the intermediate output pixels are organized in the form from row-by-row, map-by-map to layer-by-layer in continuous DRAM addresses.

B. DMA Manager

The DMA engine is used to communicate data between DRAM and on-chip BRAMs. A custom DMA manager module is designed to control the DMA operation using preload

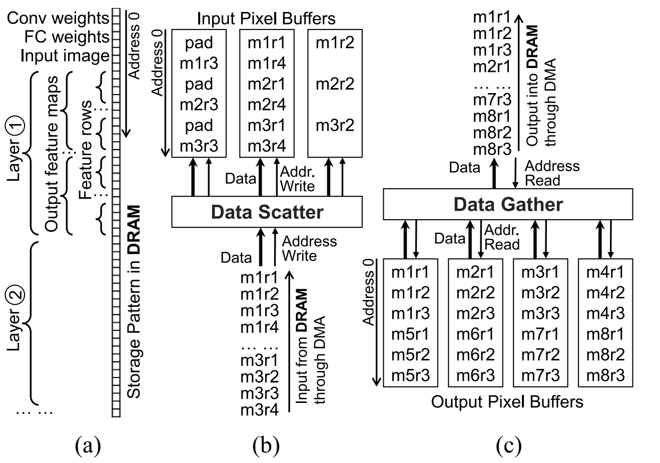


Fig. 6. (a) Storage pattern of kernel weights and intermediate pixel results in external DRAM. (b) Data scatter is used to distribute data stream from DRAM into multiple input buffers. (c) Data gather is used to collect data from multiple output pixel buffers to DRAM, where mXrY denotes the Yth row in the Xth feature map [14].

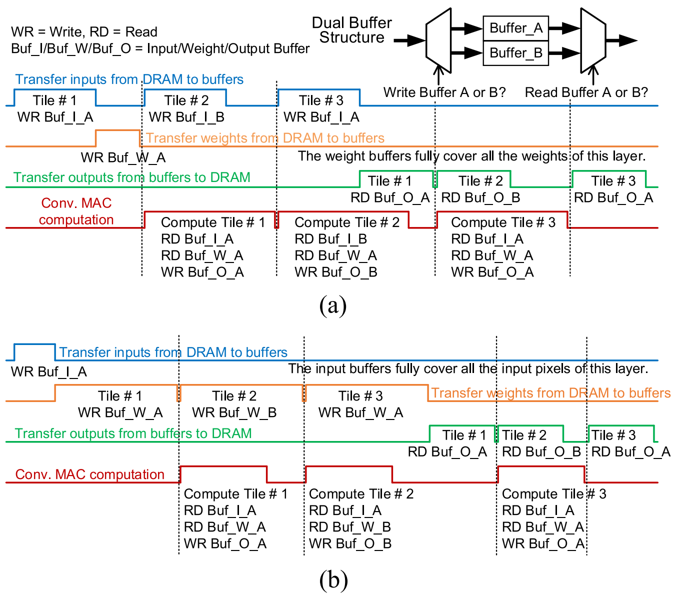

WR = Write, RD = Read
Buf_I/Buf_W/Buf_O = Input/Weight/Output Buffer
Dual Buffer Structure
Buffer_A
Buffer_B
Write Buffer A or B?
Read Buffer A or B?
Transfer inputs from DRAM to buffers
Transfer weights from DRAM to buffers
The weight buffers fully cover all the weights of this layer.
Transfer outputs from buffers to DRAM
Conv. MAC computation
Compute Tile # 1
Compute Tile # 2
Compute Tile # 3
The input buffers fully cover all the input pixels of this layer.
(a)
(b)

Fig. 7. Dual buffer structure and its pipeline schedule is used to overlap computation with memory communication to improve the throughput. (a) All the weights of this layer are fully buffered and the weights only need to be read once from DRAM. (b) All the pixels of this layer are fully buffered and the pixels only need to be read once from DRAM.

descriptors. The descriptor sets the source and destination addresses and the transaction bytes. Given the CNN parameters, loop design variables and the order of computation of the layers, the descriptors are generated by the compiler and stored in the on-chip BRAM. As weights are loaded into the DRAM before acceleration, we have the freedom the reorganize the weight storage pattern during compilation to enable the continuous DRAM read operations. Therefore, a tile needs only one descriptor to read the weights. As we compute multiple output feature maps in parallel, *Pof* weights from *Pof* kernel maps are grouped together and continuously stored in DRAM. The weight groups are stored in the order along *Nkx*, *Nky*, *Nif*, and *Nof* dimensions. To read/write the pixels from/to the DRAM, one descriptor is responsible to transfer a portion of one input/output feature map, e.g., $Tix \times Tiy$ continuous pixels. If one entire feature map is buffered, e.g., $Tix = Nix$ and $Tiy = Niy$, one descriptor can read/write multiple feature maps because these pixels across different maps are also continuously stored.

C. Data Scatter and Gather

The accelerator has two memory mapped slave ports to receive/send data from/to one DMA, respectively. The data stream from the DRAM is in continuous form and a data scatter is designed to distribute and rearrange data to multiple input pixel buffers as illustrated in Fig. 6(b), where mXrY denotes the Yth row in the Xth feature map. With different length of feature map rows, one mXrY may occupy different number of addresses. The data scatter module counts the number of received pixels based on the received DMA write signal and generates the write addresses and write enable signal for the buffers. Similarly, the data gather module in Fig. 6(c) is designed to collect data from multiple output pixel buffers into continuous form to benefit DMA transactions.

D. Dual Buffer Structure

The dual buffer structure (or ping-pong buffer structure) [10] is employed to overlap the PE computation with external memory communication to decrease the overall latency. By this means, while the DMA is writing/reading one buffer, the PE array can read/write the other buffer simultaneously, as illustrated in Fig. 7. With one DRAM bank, the DMA is designed not to read and write DRAM at the same time to avoid potential conflict, and the DMA only sequentially writes input/weight buffers and reads output buffers at different times.

Fig. 7(a) illustrates the pipeline schedule when the weight buffers fully cover all the weights of this Conv layer, where we only need to read the weights from DRAM once and different tiles can reuse the weights without reloading them from the DRAM again. Similarly, the input buffers fully cover all the input pixels of this Conv layer in Fig. 7(b), and different tiles can reuse the pixels. If the buffers cannot fully cover either all pixels or all weights of one layer, the same pixels or weights need to be read multiple times from the DRAM [9].

Before the computation of Tile #1, we need to load both input pixels and weights of Tile #1 into the buffers. While computing Tile #1, we can start to load the inputs [Fig. 7(a)] or weights [Fig. 7(b)] of Tile #2 into the other buffer and write outputs to the output buffer. In Fig. 7(a), the computation time of Tile #1 is longer than the delay of loading input buffer, so Tile #2 can only start after the completion of Tile #1 computation, which means its overall delay is bounded by the computation delay. On the other hand, in Fig. 7(b), the memory delay is longer than the computation time of Tile #1, so Tile #2 can only start after the memory transaction is finished, which makes its delay bounded by the memory communication delay. Since the DMA can only start reading the output buffer after the computation of this tile is fully completed, the outputs of Tile #1 are transferred to DRAM during/after the computation of Tile #2, while the outputs of Tile #2 are written into the other output buffer. To simplify the control logic, the pipeline of computation and memory transaction is currently only within each layer. By this means, the write of input/weight buffers of the first tile and the read of output buffer of the last tile are not overlapped with computation, which limits the efficiency of dual buffer structure to further improve the throughput.

E. Computation Bounded Versus Memory Bounded

The roofline model is introduced in [10] to analyze the performance bottleneck of the CNN accelerator, which is mainly affected by the available computation resources (DSP or MAC units) and the external memory (DRAM) bandwidth. When overlapping computations with external memory transactions, if the computation delay exceeds the memory delay, the design is said to be *computation bounded*, with the bound referred to as the *computation roof throughput*. Otherwise, it is said to be *memory bounded*, with the bound referred to as the *memory roof throughput*. The computation roof throughput (DSP_roof) is defined as

$$\text{DSP\_roof(GOPS)} = \frac{\text{\#operations(GOP)}}{\text{DSP\_delay(s)}}$$
$$\text{DSP\_delay(s)} = \frac{\text{\#operations}}{2 \times \text{\#MACs}} \times \text{clock\_period(s)} \quad (1)$$

where #operations is the number of operations and #MACs is the number of MAC units. One MAC unit computes two operations (one multiplication and one addition) at one clock cycle. Therefore, the DSP_roof is determined by the number of MAC units and the operating clock frequency. The memory roof throughput (DRAM_roof) is defined as

$$\text{DRAM\_roof(GOPS)} = \frac{\text{\#operations(GOP)}}{\text{DRAM\_delay(s)}}$$
$$\text{DRAM\_delay(s)} = \frac{\text{\#data(GB)}}{\text{DRAM\_BW(GB/s)}} \quad (2)$$

where DRAM_BW is the external memory bandwidth, and #data is the data size of memory accesses including both reading inputs/weights from DRAM and writing outputs to DRAM. The *roof throughputs* (DSP_roof and DRAM_roof) are shown in Fig. 8 for each Conv layer of different CNN algorithms. The DSP_roof of Arria 10/Stratix 10 are computed with different number of MAC units at 240/300 MHz, respectively. The DRAM_roof is directly proportional to computation to communication ratio (CTC) [10] by memory bandwidth, e.g., 12 GB/s in Fig. 8. If DSP_roof is lower than DRAM_roof, the design is computation bounded, otherwise it is memory bounded. Obviously, the attainable throughputs are lower than both roof throughputs. With relatively large intermediate feature map dimensions and kernel sizes, VGG-16 has a larger CTC ratio or memory roof throughput than NiN, GoogLeNet, and ResNet, which makes its implementation easier to be computation bounded as shown in Fig. 8. By this means, the increase of hardware resources, e.g., from Arria 10 to Stratix 10, is expected to benefit the throughput improvements of VGG-16 more than the other three algorithms, which will be demonstrated in Section VII. The DSP_roof with 6272 MAC units on Stratix 10 are already larger than DRAM_roof of most layers in NiN, GoogLeNet and ResNet as in Fig. 8 that makes the design memory bounded, which means the increase of the number of MAC units to be 8192 will only bring insignificant performance enhancement. Limited by the utilization of computation resources and the efficiency of external memory accesses, the real throughput of one layer may not be able to achieve the roof throughput of this layer.

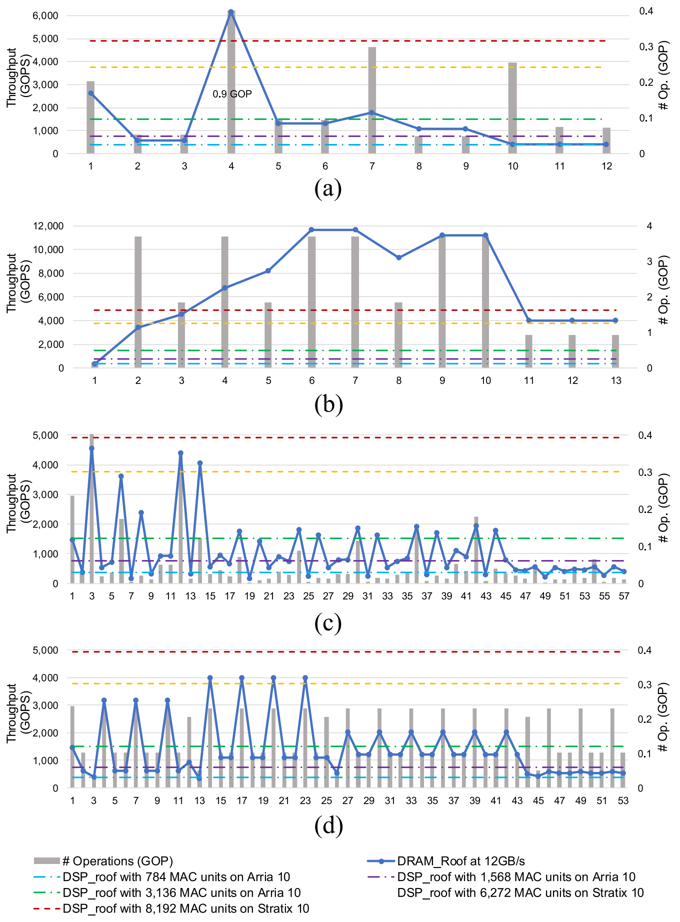


Fig. 8. *Roof throughputs* are limited by computation resources and memory bandwidth at different layers of diverse CNN algorithms. The computation roof throughputs [DSP_roof(GOPS)] of Arria 10/Stratix 10 are computed with different number of MAC units at 240/300 MHz, respectively. The memory roof throughputs [DRAM_roof(GOPS)] are computed with 12 GB/s bandwidth. The attainable peak throughput equals to min(DSP_roof, DRAM_roof). If DSP_roof is lower than DRAM_roof, the design is *computation bounded*, otherwise it is *memory bounded*.

VI. Reconfigurable CNN Computing Modules

A. Convolution Modules (Conv)

Based on our convolution acceleration strategy, the module template of Conv layer is designed as in Fig. 9, which follows the computing architecture in [9]. There are $Pox \times Poy \times Pof$ independent PEs in Conv module, and each PE is an MAC unit consisting of one multiplier followed by an accumulator. With judiciously chosen loop unrolling scheme, both pixels and weights are reused by multiple MAC units to reduce buffer read operations. The partial sums are consumed inside each MAC unit so that the movements of partial sums are minimized.

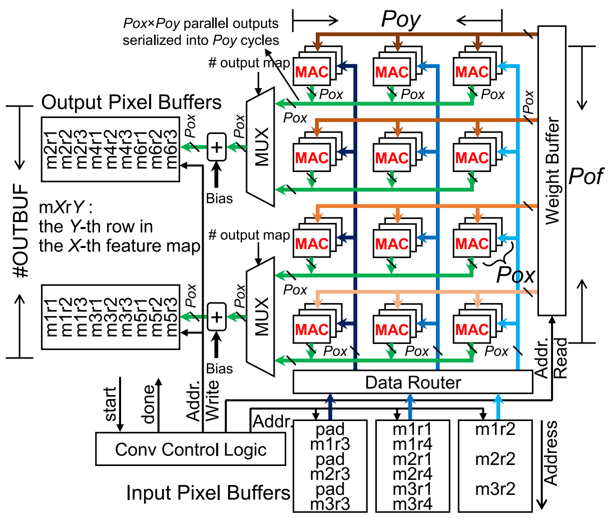


Fig. 9. Convolution computing module (Conv) including buffers, where one MAC is composed of one multiplier followed by an accumulator.

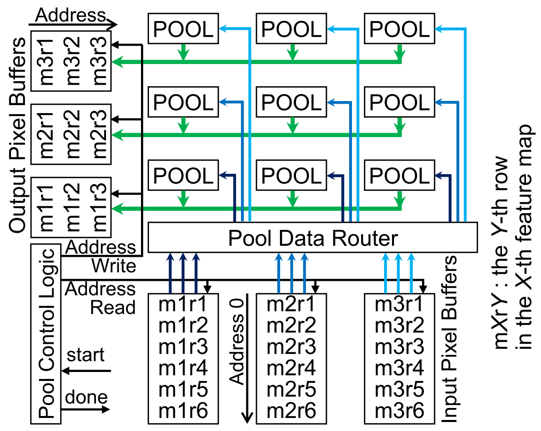


Fig. 10. Max-pooling computing module (Max-Pool) including buffers, where the POOL blocks are Max-Pool PEs of comparators [14].

The *local control logic* inside the Conv module receives the start flag signal from global control logic and controls the sequential computation of the four convolution loops. It is composed of multiple counters, which iterates from 0 to the dimensions of the feature and kernel maps, the number of input and output feature maps, respectively. These parameters are read from configuration registers by the global control logic during runtime. By this means, for different Conv layers, the compiler only needs to generate associated parameters for each layer and maintain the same logic implementation. The combination of the counter values in local control logic generates the buffer read and write addresses. Instead of assigning individual Conv module for each Conv layer as in [16], the computing module in this paper is reused by all the layers of the same type, thanks to the uniform mapping of PEs and shared local control logic.

The *data router* [9] inside the Conv module is used to reshape the data form and continuously feed input pixels from buffers into MAC units. It is comprised of multiple data buses to handle the dataflow of different configurations of sliding strides and zero paddings for different Conv layers. The control logic governs the switch among different data buses for the corresponding layer. The data router can easily handle different kernel sizes without penalty of idle clock cycles and additional logic resources, which is realized by sequentially sliding the kernel window ($Pkx = Pky = 1$). The compiler only needs to change the iteration boundary of the counters inside the control logic for the corresponding kernel size.

There are $Pox \times Poy \times Pof$ parallel outputs from the MAC units, and they are serialized into Poy consecutive clock cycles to reduce the required number of bias adders and the data width of output buffers. The $Pox \times Pof$ outputs are further serialized to be $Pox\times$ #OUTBUF using multiplexers with output feature maps stacked in the output buffer as shown in Fig. 9.

B. Pooling Modules (Pool)

Pooling layer (Pool) is commonly employed to reduce the dimensionality of feature maps by replacing pixels within a pooling sliding window by their maximum or average value. Pool only needs pixels from its previous layer, so it can be treated as an affiliated layer to eliminate DRAM accesses as in [9] and [16]. However, the loop design variables of Pool must depend on its key layer and this dependency can worsen the design flexibility. If the key layer has $Toy < Noy$, the pixels of one sliding window may be separated into two tiles, which demands the storage of pixels from the last tile and causes imbalance of pooling operations across tiles. Therefore, we treat Pool as a key layer to enable independent design configurations at the cost of DRAM access delay. Considering the small number of Pool layers in CNNs, the overhead in the total latency is insignificant. Since average-pooling (Ave-Pool) is normally at the end, where Noy is small with $Toy = Noy$, it is not affected by the tiling problem. To that end, we still implement Ave-Pool as an affiliated layer by reading input data directly from output buffers of its previous layer.

The Max-Pool module is shown in Fig. 10, which consists of local control logic, register arrays, and PEs. The difference from the Ave-Pool module is that the input data are from output buffers. The counters inside the local control logic control the sliding within one feature map and across different feature maps, and generate the buffer read and write addresses. The Pool PEs ("POOL" component in Fig. 10) are either comparators for Max-Pool or accumulators followed by constant coefficient multipliers for Ave-Pool. Pixels from one feature map are stored in one input buffer and processed by one row of PEs as illustrated in Fig. 10. The different data storage pattern in the input buffers from that of Conv layer is handled by the data scatter module. The column size of PE array is constrained by the input buffer output width and the row size equals to the number of used input buffers, which can be adjusted before compilation. The data router in Pool is employed to ensure continuous feeding of pixels into PEs without idle cycles.

C. Batch Normalization and Scale (Bnorm)

Batch normalization followed by scale has been commonly used in recent CNN models [4], [6], enabling fast training convergence. Their operations are depicted in

$$y = \frac{x - bn0}{\sqrt{bn1}} \tag{3}$$

$$z = sc0 \times y + sc1. \tag{4}$$

During the inference process, $bn0$, $bn1$, $sc0$, and $sc1$ are all constants for each output feature map along Nof. Therefore, we can combine batch normalization with scale (Bnorm) to be a single equation

$$z = A \times x + B \tag{5}$$

where $A = sc0/\sqrt{bn1}$ and $B = sc1 - sc0 \times bn0/\sqrt{bn1}$. However, (5) still requires multipliers and adders that are expensive. To further save the computation resources, we continue to merge Bnorm with its preceding Conv layer. The convolution operation can be briefly expressed as

$$x(no) = \sum_{ni=1}^{Nif \times Nky \times Nkx} p(ni) \times w(ni, no) + \text{bias}(no) \qquad no \in [1, Nof] \tag{6}$$

where $p(ni)$ is the input pixel and $w(ni, no)$ is the kernel weight, and the Conv output, e.g., $x(no)$, is the input to Bnorm in (5). After applying (6) to (5), we have

$$z(no) = \sum_{ni=1}^{Nif \times Nky \times Nkx} p(ni) \times A(no) \times w(ni, no) + A(no) \times \text{bias}(no) + B(no), no \in [1, Nof]. \tag{7}$$

By this mean, the Conv layer merged with Bnorm has new weights as $A(no) \times w(ni, no)$ and new biases as $A(no) \times \text{bias}(no) + B(no)$, with $no \in [1, Nof]$. Then, we can get rid of the Bnorm computations during inference, and the new weights and biases of Conv are precomputed off-line to replace the original data. Therefore, there is no Bnorm module in Fig. 5.

D. Element Wise (Eltwise)

The Eltwise layer performs element-wise addition to connect two branches of layers in ResNet CNNs as shown in Fig. 1. As discussed in Section IV, we serially compute the two branches. Eltwise is treated as an affiliated layer to the key Conv layer in one branch and the other branch is computed first.

Eltwise is performed after its previous layer in the same branch has stored all the results into the output buffers. Then, the pixels from the other branch are read from DRAM and written into the input pixel buffers. Subsequently, the pixels from the two branches are element-wise added by the adders and finally stored back into the output pixel buffers, as illustrated in Fig. 11. The output buffers are implemented as dual-port RAMs so that the adder results can be written back to the output buffers at their addends original locations without using additional buffers. A few pipeline stages are introduced in the adders to avoid the conflict of writing and reading at the same output buffer address.

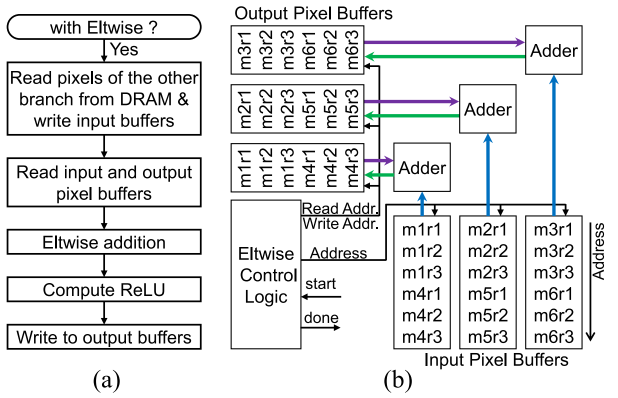

with Eltwise ?
Yes
Read pixels of the other branch from DRAM & write input buffers
Read input and output pixel buffers
Eltwise addition
Compute ReLU
Write to output buffers
(a)
Output Pixel Buffers
m3r1 m3r2 m3r3 m6r1 m6r2 m6r3
m2r1 m2r2 m2r3 m5r1 m5r2 m5r3
m1r1 m1r2 m1r3 m4r1 m4r2 m4r3
Adder
Adder
Adder
Eltwise Control Logic
Read Addr.
Write Addr.
Address
start
done
m1r1 m1r2 m1r3 m4r1 m4r2 m4r3
m2r1 m2r2 m2r3 m5r1 m5r2 m5r3
m3r1 m3r2 m3r3 m6r1 m6r2 m6r3
Address
Input Pixel Buffers
(b)

Fig. 11. (a) Eltwise execution schedule. (b) Eltwise module architecture [14].

E. Concat Layer

The Concat layer is used to concatenate the outputs of multiple layers together as shown in Fig. 1. In this paper, we assume the concatenation is only along multiple channels and all the input layers must have the same feature map sizes ($Nix \times Niy$), which is the case for GoogLeNet [5] and Inception [6]. If the inputs of one layer is from Concat, the compiler generates DMA descriptors that control DMA to read multiple layers of the Concat from different DRAM addresses as the inputs. Since there is no computation in Concat, it does not add overhead to the hardware.

F. Fully Connected

The FC layer can be treated as a special form of Conv with kernel size as $Nkx \times Nky = 1 \times 1$ and feature map size as $Nox \times Noy = 1 \times 1$. As the kernel weights are not shared by pixels of one feature map, FC layer normally has a large volume of weights but with only a few operations, which makes FC layers memory intensive. Therefore, FC layers reuse the weight buffers with Conv layers, and the dual buffer technique is still used to overlap the memory delay with computation, which improves the FC latency especially for VGG implementation with heavy FC layers. The parallel computation of FC matrix-vector multiplication is only employed across different output feature maps such that only one pixel is multiplied with multiple weights simultaneously, and the MAC units in Conv are reused for FC layers.

VII. Experimental Results

A. Experimental Setup

The proposed CNN compilation methodology is demonstrated by accelerating the inference process of both conventional CNNs, e.g., NiN and VGG, and complex DAG form CNNs, e.g., GoogLeNet and ResNet, on two Intel FPGAs. The two Intel FPGAs, e.g., Arria 10 GX 1150/Stratix 10 GX 2800 FPGA, consist of 427K/933K adaptive logic modules (ALMs), 3036/11 520 fixed-point 18-bit × 18-bit DSP blocks, and 2713/11 721 M20K BRAM blocks, where each M20K BRAM exhibits 20 Kbit storage. The underlying FPGA boards for Arria 10 and Stratix 10 are Nallatech 385A and Stratix 10 GX FPGA Development Kit, respectively, and both are equipped with DDR3 DRAM with theoretical peak memory bandwidth of 16.9 GB/s. The power consumption of Arria 10 and Stratix 10 boards are about 40 and 100 W, respectively, based on the datasheet specifications. The compiled Verilog scripts are synthesized by Quartus Prime. The fixed point data representation is employed by the compiler with dynamic quantization, which dynamically adjusts the decimal point according to the ranges of data values in different layers to fully utilize the existing data width [9], [17]. The data precision can be tuned to trade classification accuracy for hardware utilization and throughput.

B. Parallel Computation Efficiency

Considering that the DSP blocks in Arria 10 and Stratix 10 can implement 3036 and 11 520 fixed-point multipliers for MAC units, respectively, the maximum number of MAC units on the two FPGAs can be around 3000 and 11 000, respectively. To achieve better performance with higher parallelism, we attempt to maximize the usage of DSP blocks for the MAC operations. Based on the optimized acceleration strategy, the parallel or unrolled loop computations are within one feature map ($Pox \times Poy$) and across multiple output channels (Pof). Since the feature map sizes ($Nox \times Noy$) and the number of output channels (Nof) vary significantly across different layers in different CNN algorithms, the loop unrolling degree and shape may not perfectly match the feature map size and dimension, which causes inefficient utilization of DSP blocks or MAC units. Therefore, the DSP efficiency [18] is defined to measure how well the parallel computation scheme matches the convolution loop dimension

$$\text{DSP\_efficiency} = \frac{\#\text{effective} \quad \text{ops.}}{\#\text{actual} \quad \text{performed} \quad \text{ops.}}. \tag{8}$$

The DSP efficiency of different convolution layers is shown in Fig. 12 using GoogLeNet as an example. Although Fig. 12(a)–(d) have the same number of parallel MAC units ($Pox \times Poy \times Pof = 3136$) on Arria 10, their loop unrolling shape is different, which results in significant difference of the overall DSP efficiency from 0.63 to 0.93. The first several layers of GoogLeNet have large feature map sizes, e.g., 114×114 and 57×57, so that the loop unrolling sizes, e.g., 28×7 and 14×14, can be easily fit into the feature maps. The layers at the end have small feature map sizes, e.g., 14×14 and 7×7, which leads to DSP efficiency degradation except for Fig. 12(c) with small $Pox \times Poy = 7 \times 7$. However, GoogLeNet still has layers with small number of output channels, e.g., 16 and 32, in the middle, which hurts the DSP efficiency of Fig. 12(c) with large $Pof = 64$. Finally, Fig. 12(c) and (d) show similar overall DSP efficiency, and they are better than the other unrolling scenarios. Stratix 10 in Fig. 12(e) has larger parallel degrees ($=14 \times 7 \times 64$) than Arria 10, which makes it more difficult to exactly match the loop dimensions of all the layers and results in lower DSP efficiency.

C. Performance Analysis

The throughput of the CNN accelerator is collectively determined by the employed computation resources and memory

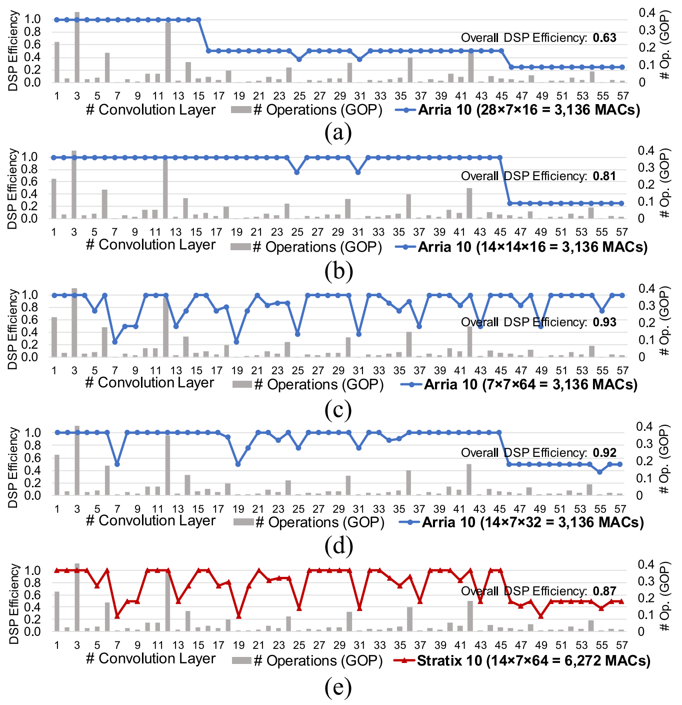


Fig. 12. DSP efficiency of different convolution layers in GoogLeNet is shown to measure the degree of matching between loop dimensions and loop unrolling ($Pox \times Poy \times Pof$), where (a)–(d) have the same size of loop unrolling (=3136) but with different shapes, and (e) has larger loop unrolling size with 6272 MAC units.

bandwidth as discussed in Section V-E, as well as the DSP efficiency, the number of external memory accesses, and the overlapping of computation and memory transactions. The throughput of each convolution layer in ResNet-50, GoogLeNet, and VGG-16 is shown in Fig. 13 with different number of MAC units on Arria 10 (running at 240 MHz) and Stratix 10 (running at 300 MHz). If the memory bandwidth is unlimited, the shape of the throughput curve should well match their corresponding DSP efficiency curve. However, with limited memory bandwidth, layers with small number of operations or small CTC ratios tend to be memory bounded, e.g., Conv #1 in VGG-16, Conv #7 in GoogLeNet, and Conv #3 in ResNet-50 in Fig. 13. With the increased number of MAC units, the design is more likely to be memory bounded with the same memory bandwidth, which limits further improvement of throughput by using more MAC units. As expected in Section V-E, layers in VGG-16 have large CTC on Arria 10 and Stratix 10, whose throughputs can be significantly improved with the increase of MAC units. On the contrary, a lot of layers in ResNet and GoogLeNet are memory bounded, especially for Stratix 10, which limits the additional improvement of throughput on Stratix 10.

As mentioned in Section VII-B, even if the number of MAC units is the same, the different loop unrolling shapes may considerably impact the DSP efficiency, which will further affect the performance. The effect of different loop unrolling shapes on the throughput of different CNNs is shown in Fig. 14 on Arria 10 with 3136 MAC units. Although the loop unrolling of $14 \times 14 \times 16$ has worse DSP efficiency than $7 \times 7 \times 64$ for

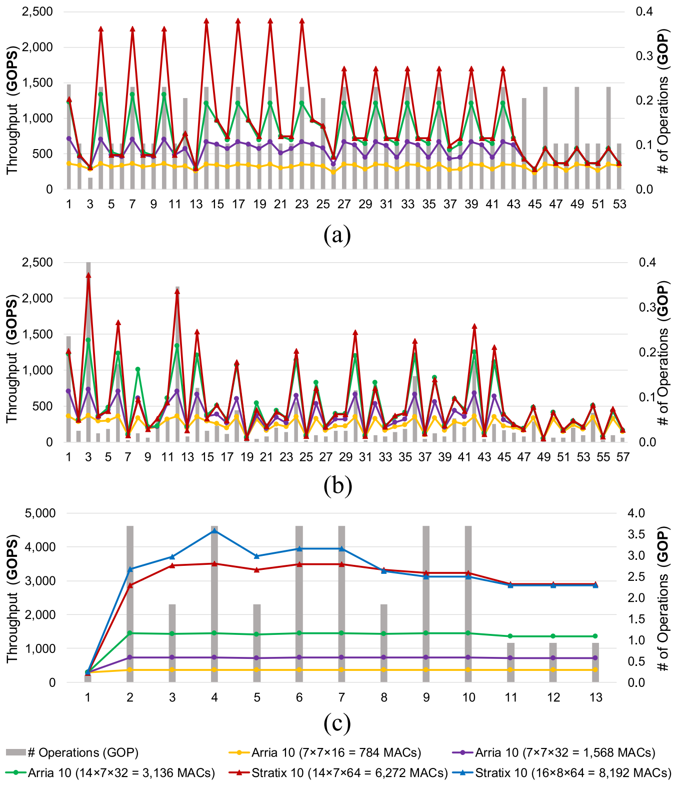


Fig. 13. Throughput of each convolution layer in ResNet-50, GoogLeNet, and VGG-16 with different number of MAC units on Arria 10 (240 MHz) and Stratix 10 (300 MHz).

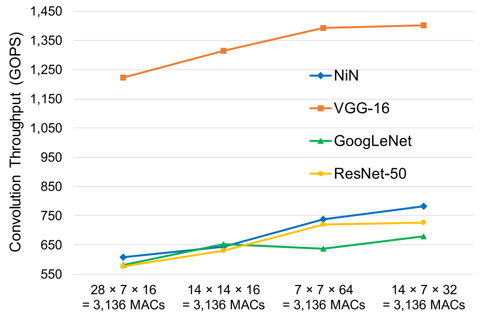


Fig. 14. Convolution throughput of different CNNs on Arria 10 with the same number of MAC units is affected by the shape of loop unrolling ($Pox \times Poy \times Pof$).

GoogLeNet in Fig. 12 resulting in longer computation latency, the throughput of $14 \times 14 \times 16$ is higher than that of $7 \times 7 \times 64$ in Fig. 14, which means $14 \times 14 \times 16$ of GoogLeNet allows better overlapping of computation and memory communication that overcompensates its longer computation time. The loop unrolling configuration of $14 \times 7 \times 32$ shows supreme throughput than other configurations for all the CNNs in Fig. 14, thus we take it as our optimal choice for the Arria 10 implementation.

The convolution throughputs of different CNNs on Arria 10 and Stratix 10 with different number of MAC units are shown in Fig. 15(a). As mentioned before, most layers in VGG-16 have large CTC ratios that makes them more likely to be computation bounded, thus the throughput improvement of VGG-16 can benefit more from the increase of MAC units

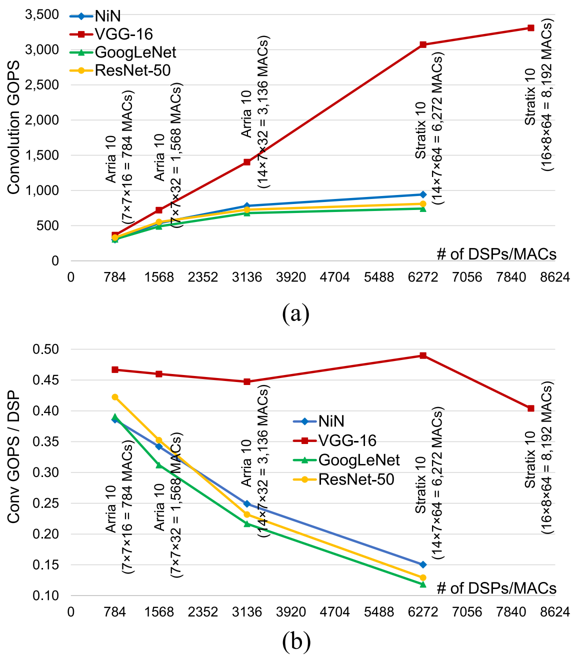


Fig. 15. (a) Compiler is scalable to change the number of MAC units ($Pox \times Poy \times Pof$) to trade throughput for resource usage, e.g., DSP blocks. The increasing of throughputs with more MAC units are saturating due to lower DSP efficiency and limited memory bandwidth. (b) With the increased number of DSPs, the convolution throughputs normalized to one DSP (Conv GOPS/DSP) tend to decrease due to the saturation of throughputs.

than the other three CNNs. The implementations of NiN, GoogLeNet, and ResNet with 6272 MAC units on Stratix 10 are already memory bounded so that more MAC units can only result in negligible throughput improvement, meanwhile more hardware resources are needed. If we target at smaller FPGA devices with less computation resources, e.g., DSP and logic, the compiler is scalable to decrease the number of MAC units by assigning smaller loop unrolling size to reduce the resource requirements at the cost of lower performance. The normalized convolution throughputs (Conv GOPS/DSP) are shown in Fig. 15(b) to measure the performance provided by a single DPS block or MAC unit, which tend to decrease with more MAC units as the throughputs are saturated due to the lower DSP efficiency and limited memory bandwidth. VGG-16 exhibits higher normalized throughputs than other algorithms due to the higher CTC ratio to benefit more from the increase of DSP blocks.

D. Results of the CNN Inference Accelerator

The specifications and performance of the proposed compiler configured CNN FPGA accelerators are compared in Table I. As discussed before, although Stratix 10 provides $>3.3\times$ higher computation capability than Arria 10, the overall throughput improvements of Stratix 10 over Arria 10 are from $1.06\times$ to $1.66\times$ due to the lower DSP efficiency and limited external memory bandwidth, which considerably reduce the normalized throughputs (GOPS/DSP) of Stratix 10. Suffered from heavy FC layers, which are memory bounded, the overall throughput of VGG-16 on Arria 10/Stratix 10 (968/1604 GOPS) is much lower than

TABLE I
COMPARISON OF THE COMPILED CNN ACCELERATORS ON ARRIA 10 AND STRATIX 10 FPGAS (BATCH SIZE = 1)

CNN	NiN	VGG-16	GoogLeNet	ResNet-50	ResNet-152
# of Operations (GOP)	2.2	30.95	3.18	7.74	22.62
# of Parameters	7.59 M	138.3 M	6.07 M	25.5 M	60.4 M
Weight/Pixel Precision (fixed)	16 bit	8/16 bit	16 bit	16 bit	16 bit
FPGA / Tech.	Intel Arria 10 GX 1150 / 20 nm				
Clock	240 MHz	240 MHz	240 MHz	240 MHz	240 MHz
$Pox \times Poy \times Pof$	$14 \times 7 \times 32$	$14 \times 7 \times 32$	$14 \times 7 \times 32$	$14 \times 7 \times 32$	$14 \times 7 \times 32$
DSP Blocks	3,036 (100%)	3,036 (100%)	3,036 (100%)	3,036 (100%)	3,036 (100%)
Logic (ALMs)	256K (60%)	208K (49%)	277K (65%)	286K (67%)	335K (78%)
On-chip RAM (M20K)	1,605 (59%)	2,319 (85%)	1,849 (68%)	2,356 (87%)	2,692 (99%)
Latency/Image (ms)	3.01	31.97	6.05	12.87	32.37
Overall Throughput (GOPS)	732.36	968.03	524.98	599.61	697.09
GOPS/DSP	0.24	0.32	0.17	0.20	0.23
FPGA / Tech.	Intel Stratix 10 GX 2800 / 14 nm				
Clock	300 MHz	300 MHz	300 MHz	300 MHz	300 MHz
$Pox \times Poy \times Pof$	$14 \times 7 \times 64$	$16 \times 8 \times 64$	$14 \times 7 \times 64$	$14 \times 7 \times 64$	$14 \times 7 \times 64$
DSP Blocks	6,304 (55%)	8,216 (71%)	6,304 (55%)	6,304 (55%)	6,304 (55%)
Logic (ALMs)	487K (52%)	469K (50%)	528K (57%)	559K (60%)	623K (67%)
On-chip RAM (M20K)	1,915 (16%)	2,421 (21%)	1,949 (17%)	3,014 (26%)	3,350 (29%)
Latency/Image (ms)	2.56	19.29	5.70	11.85	28.59
Overall Throughput (GOPS)	858.66	1604.57	557.08	651.49	789.44
GOPS/DSP	0.14	0.20	0.09	0.10	0.13

the convolution throughput (1402/3309 GOPS), respectively. The latency improvements brought by dual buffer structure is shown in Fig. 16. Since the computation and the memory transaction cannot be perfectly fully overlapped as mentioned in Section V-D, the actual total latency is larger than the theoretical minimum latency, which equals to the larger one of computation delay and DRAM delay. As the Stratix 10 FPGA board has only one DRAM bank, we also keep using one DRAM bank for the Arria 10 implementation for comparison purposes in this paper. Therefore, the throughput of ResNet on Arria 10 is lower than that in [14] using two DRAM banks, even though the dual buffer structure is used in this paper. If the Arria 10 implementations in [14] also use one DRAM bank, the throughputs of ResNet-50 and ResNet-152 could be decreased to 440 GOPS and 530 GOPS, which are 1.36× and 1.32× worse than this paper, respectively. Although the external memory bandwidth in this paper is only half of that in [14], the throughputs of NiN and VGG-16 are still 1.25× and 1.34× higher than [14], respectively, mainly due to the dual buffer structure, higher frequency and lower precision of weights in VGG-16. Despite of smaller loop tiling sizes used in this paper, the on-chip memory usage of M20K on Arria 10 is still higher than [14] due to the dual buffer structure, which directly doubles the M20K consumption of buffers.

Aimed at deep CNNs, our compiler stores all the weights and intermediate pixel results in DRAM by default. Considering current trends toward compressed CNNs with dramatically reduced data bit-width and small CNNs for simpler applications, it would be possible to fit the entire CNN model into FPGA on-chip BRAM. The potential modification of our compiler is to connect the DMA engine with a large enough BRAM instead of DRAM serving as the global memory, while retaining the computing architecture the same. With decreased data size and precision, more MAC units, higher frequency, and less memory access delay could be possible to obtain higher throughput.

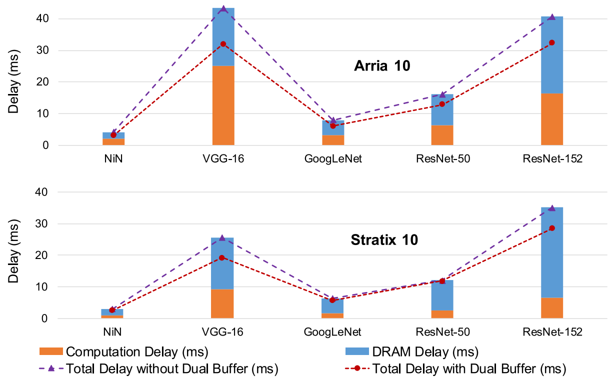


Fig. 16. Dual buffer structure is used to overlap computation delay with DRAM delay to reduce the overall total delay.

E. Comparison With Related Works

GPUs have been widely used to accelerate the training and inference tasks of CNN algorithms, which are realized by thousands of parallel processing cores, high operating clock frequencies at GHz level, and large memory bandwith of hundreds of GB/s. However, the power consumption of high performance GPUs is too high (>150 W) for power constrained applications [17], [20]. Furthermore, GPUs are best suited to process large batches of images together to fully utilize all the resources and realize high throughput (>1TFLOPS) at the cost of longer latency per image, which does not benefit the latency-critical applications, e.g., autonomous drive, that require real-time recognition results. On the other hand, numerous hardware accelerators based on application specific integrated circuits (ASICs) are recently developed achieving impressively high energy efficiency, e.g., up to 10 TOPS/W with 4-bit precision in [23] or even higher for fixed custom accelerators with binary precision [24]. However, CNNs with binary precision often incur accuracy loss, the ASIC-based accelerators are too specific to efficiently

TABLE II
RELATED WORKS ON AUTOMATED FPGA ACCELERATORS

	[19]	[17]	[20]	[20]	[21]	[21]	[14]	[18]	[22]
FPGA	Zynq XC7Z045	Zynq XC7Z045	Ultrascale KU060	Virtex7 690t	Stratix V GSMD5	Stratix V GSMD5	Arria 10 GX 1150	Arria 10 GT 1150	Stratix V GXA7
Tech.	28 nm	28nm	20 nm	28 nm	28 nm	28 nm	20 nm	20 nm	28 nm
Clock	100 MHz	150 MHz	200 MHz	150 MHz	150 MHz	150 MHz	200 MHz	232 MHz	200 MHz
CNN	NiN	VGG-16	VGG-16	VGG-16	VGG-16	ResNet-152	VGG-16	VGG-16	VGG-16
Precision	fixed	16 bit fixed	16 bit fixed	16 bit fixed	16 bit fixed	16 bit fixed	16 bit fixed	8-16 bit fixed	16 bit fixed
DSP[a]	-	780 (87%)	1,058 (38%)	2,833 (78%)	2,072 (65%)	2,072 (65%)	3,036 (100%)	3,000 (99%)	512 (100%)
Logic[b]	-	183K (84%)	100K (31%)	300K (81%)	42.3K (25%)	42.3K (25%)	132K (30%)	313K (73%)	107K (46%)
RAM[c]	-	486 (89%)	-	-	-	-	2,225 (82%)	1,668 (61%)	1,377 (73%)
Memory BW	12.8 GB/s	12.8 GB/s	12.8 GB/s	14.9 GB/s	-	-	2×16.9 GB/s	19 GB/s	5 GB/s
Latency[d]	52 ms	-	-	-	-	-	42.98 ms	26.85 ms	-
GOPS	43	137	266	354	364.36	226.47	720.15	1,171.3	669.1
GOPS/DSP	-	0.18	0.25	0.13	0.18	0.11	0.24	0.39	1.31

[a] DSP configured as fixed-point 18-bit × 18-bit for Intel FPGAs and 18-bit × 25-bit for Xilinx FPGAs.
[b] Logic elements: Xilinx FPGAs in LUTs and Intel FPGAs in ALMs.
[c] On-chip memory: Xilinx FPGAs in BRAMs (36Kb) and Intel FPGAs in M20Ks (20Kb).
[d] Latency per image with batch size of one. References without latency reported may use batch size more than one to enhance throughput.

handle various CNN algorithms, and the long development time of ASIC makes it difficult to catch up with the rapid evolution of CNN algorithms.

Benefited from the high reconfigurability and the freedom to customize the architecture, FPGAs have gained increasing popularity and there have been several works on automatically generating FPGA accelerators for CNN algorithms [16]–[22], [25], [26]. The performance and hardware utilization of these related state-of-the-art works are listed in Table II. Compared with previous works, our RTL compiler exhibits higher flexibility by handling not only conventional CNNs but also highly complex and irregular CNNs, e.g., GoogLeNet and ResNet, through reconfigurable execution schedule, on two different scale FPGAs, e.g., Arria 10 and Stratix 10. Our compiled CNN accelerators also significantly outperform prior works in terms of performance, which is achieved by hardware level optimization to accelerate convolution loops as in [9] with high hardware utilization and low data communication.

In [19], AlexNet and NiN are implemented to evaluate their FPGA accelerator generators. Our NiN implementation on Arria 10 (20 nm and 3036 DSP) obtains ~17.3× speedup compared to [19], which needs over 50 ms runtime on Xilinx Zynq-7045 (28 nm and 900 DSP). Guo *et al.* [17] presented a programmable and flexible CNN accelerator architecture, where the fixed 3 × 3 convolver used to parallel compute a kernel window could significantly degrade the DSP efficiency and throughput for irregular CNNs with varying kernel sizes, e.g., GoogLeNet and ResNet. Zhang *et al.* [20] proposed an HW/SW co-designed CNN FPGA accelerator based on high level synthesis (HLS). Our VGG-16 implementation on Arria 10 provides 2.7× and 3.6× overall throughput enhancement compared to [20] using Virtex7 690t (28 nm and 3600 DSP) and Ultrascale KU060 (20 nm and 2760 DSP) FPGAs, respectively. Guan *et al.* [21] proposed FP-DNN framework to automatically generate FPGA hardware to accelerate DNN with RTL-HLS hybrid templates. Although the Stratix V GSMD5 (28 nm and 3180 DSP) used in [21] has more DSP blocks than our Arria 10, our accelerator on Arria 10 can achieve 3.1× higher throughput for ResNet-152 by higher frequency and DSP utilization through the loop optimization technique [9]. Wei *et al.* [18] proposed an OpenCL-based automation flow to generate CNN design from high level C code to FPGA using systolic array architecture, which reduces the global PE interconnect fanout to achieve high frequency and resource utilization. The VGG-16 implementation on Arria 10 in [18] has 1.19× better latency than ours, probably because they have more efficient pipeline of dual buffering and can achieve higher memory bandwidth, e.g., 19 GB/s, which is especially important for memory bounded FC layers that comprise 28% of our Arria 10 VGG-16 total latency. However, [18] only evaluated two conventional CNNs, e.g., AlexNet and VGG-16, which have relatively regular data shape and network structure. The framework proposed in [22] automatically generates CNN accelerators on a CPU + FPGA heterogeneous computing platform, i.e., Intel HARP, where only the convolution layers are performed on FPGA except the first convolution layer in AlexNet. By reducing the convolution operation complexity by about 3× in frequency domain through algorithm optimization, [22] achieves high normalized throughputs, e.g., 1.31 GOPS/DSP, with a small number of DSPs, e.g., 512. In [26], the fpgaConvNet framework for mapping a CNN onto a Zynq-7000 FPGA platform is designed based on HLS and evaluated on several relatively small CNN models, e.g., convolutional face finder, LeNet-5, and MPCNN. The automatic CNN accelerator generation framework proposed in [25] is designed based on proposed instruction set architecture and accelerator template for both Intel and Xilinx FPGAs. The absolute performance numbers are not reported in [25] so that direct comparison cannot be made. Since our compiler generated accelerator is coded in Verilog, it is not difficult to implement on FPGAs from other vendors by changing FPGA board specified components, e.g., external memory controller.

VIII. CONCLUSION

In this paper, a library-based RTL compiler is proposed to automatically generate customized FPGA accelerator for the inference task of a given CNN algorithm, which enables high-level mapping of CNN from software to FPGA and keeps the benefit of low-level hardware optimization. An RTL library is developed to modularize the commonly used layers in CNNs with hand coded Verilog templates. These building block modules are built on the optimized acceleration strategy and configured by the hardware design variables to be scalable for different FPGAs. The topology of the given CNN is transformed into a DAG to configure the proposed execution schedule that controls runtime layer-by-layer serial processing. The flexibility of the proposed CNN compilation methodology is demonstrated on two Intel FPGAs, e.g., Arria 10 and Stratix 10, with different computing resources to implement both traditional CNNs, e.g., NiN and VGG-16, and complex CNNs, e.g., GoogLeNet and ResNets. Our compiled CNN accelerators on Stratix 10 exhibit superior performance compared to prior automation-based works by $>1.4\times$ for various well-known CNNs algorithms.

References

[1] O. Russakovsky *et al.*, "ImageNet large scale visual recognition challenge," *Int. J. Comput. Vis. (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. Conf. Neural Inf. Process. Syst. (NIPS)*, 2012, pp. 1–9.

[3] K. Simonyan and A. Zisserman. (2014). *Very Deep Convolutional Networks for Large-Scale Image Recognition*. [Online]. Available: http://arxiv.org/abs/1409.1556

[4] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 770–778.

[5] C. Szegedy *et al.*, "Going deeper with convolutions," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2015, pp. 1–9.

[6] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, Inception-ResNet and the impact of residual connections on learning," in *Proc. Conf. Artif. Intell. (AAAI)*, Feb. 2017, pp. 4278–4284.

[7] W. Liu *et al.*, "SSD: Single shot multibox detector," in *Proc. Comput. Vis. (ECCV)*, Oct. 2016, pp. 21–37.

[8] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards real-time object detection with region proposal networks," in *Proc. Conf. Neural Inf. Process. Syst. (NIPS)*, Dec. 2015, pp. 1–9.

[9] Y. Ma, Y. Cao, S. Vrudhula, and J.-S. Seo, "Optimizing the convolution operation to accelerate deep neural networks on FPGA," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 26, no. 7, pp. 1354–1367, Jul. 2018.

[10] C. Zhang *et al.*, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *Proc. ACM/SIGDA Int. Symp. Field Programm. Gate Arrays (FPGA)*, 2015, pp. 161–170.

[11] L. Song *et al.*, "C-brain: A deep learning accelerator that tames the diversity of CNNs through adaptive data-level parallelization," in *Proc. Design Autom. Conf. (DAC)*, Jun. 2016, pp. 1–6.

[12] K. He, X. Zhang, S. Ren, and J. Sun, "Identity mappings in deep residual networks," in *Proc. Comput. Vis. (ECCV)*, Oct. 2016, pp. 630–645.

[13] Y. Jia *et al.*, "Caffe: Convolutional architecture for fast feature embedding," in *Proc. ACM Int. Conf. Multimedia*, Nov. 2014, pp. 675–678.

[14] Y. Ma, Y. Cao, S. Vrudhula, and J.-S. Seo, "An automatic RTL compiler for high-throughput FPGA implementation of diverse deep convolutional neural networks," in *Proc. Int. Conf. Field Programm. Logic Appl. (FPL)*, Sep. 2017, pp. 1–8.

[15] M. Lin, Q. Chen, and S. Yan, "Network in network," *CoRR*, vol. abs/1312.4400, 2013. [Online]. Available: http://arxiv.org/abs/1312.4400

[16] Y. Ma, N. Suda, Y. Cao, S. Vrudhula, and J.-S. Seo, "ALAMO: FPGA acceleration of deep learning algorithms with a modularized RTL compiler," *Integr. VLSI J.*, vol. 62, pp. 14–23, Jun. 2018.

[17] K. Guo *et al.*, "Angel-eye: A complete design flow for mapping CNN onto embedded FPGA," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 1, pp. 35–47, Jan. 2018.

[18] X. Wei *et al.*, "Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs," in *Proc. Design Autom. Conf. (DAC)*, 2017, pp. 1–6.

[19] Y. Wang, J. Xu, Y. Han, H. Li, and X. Li, "DeepBurning: Automatic generation of FPGA-based learning accelerators for the neural network family," in *Proc. Design Autom. Conf. (DAC)*, Jun. 2016, pp. 1–6.

[20] C. Zhang, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks," in *Proc. Int. Conf. Comput.-Aided Design (ICCAD)*, Nov. 2016, pp. 1–8.

[21] Y. Guan *et al.*, "FP-DNN: An automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates," in *Proc. IEEE Int. Symp. Field Programm. Custom Comput. Mach. (FCCM)*, Apr./May 2017, pp. 152–159.

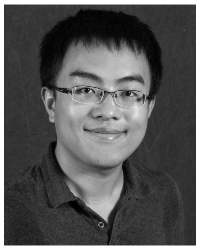
[22] H. Zeng, R. Chen, C. Zhang, and V. K. Prasanna, "A framework for generating high throughput CNN implementations on FPGAs," in *Proc. ACM/SIGDA Int. Symp. Field Programm. Gate Arrays (FPGA)*, Feb. 2018, pp. 117–126.

[23] B. Moons, R. Uytterhoeven, W. Dehaene, and M. Verhelst, "Envision: A 0.26-to-10 TOPS/W subword-parallel dynamic-voltage-accuracy-frequency-scalable convolutional neural network processor in 28nm FDSOI," in *Proc. IEEE Int. Solid-State Circuits Conf. (ISSCC)*, Feb. 2017, pp. 246–247.

[24] D. Bankman, L. Yang, B. Moons, M. Verhelst, and B. Murmann, "An always-on 3.8μJ/86% CIFAR-10 mixed-signal binary CNN processor with all memory on chip in 28nm CMOS," in *Proc. IEEE Int. Solid-State Circuits Conf. (ISSCC)*, Feb. 2018, pp. 222–224.

[25] H. Sharma *et al.*, "From high-level deep neural models to FPGAs," in *Proc. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, Oct. 2016, pp. 1–12.

[26] S. I. Venieris and C.-S. Bouganis, "fpgaConvNet: A framework for mapping convolutional neural networks on FPGAs," in *Proc. IEEE Int. Symp. Field Programm. Custom Comput. Mach. (FCCM)*, May 2016, pp. 40–47.



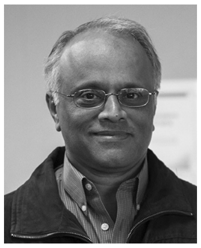
Yufei Ma (S'16) received the B.S. degree in information engineering from the Nanjing University of Aeronautics and Astronautics, Nanjing, China, in 2011 and the M.S.E. degree in electrical engineering from the University of Pennsylvania, Philadelphia, PA, USA, in 2013. He is currently pursuing the Ph.D. degree with Arizona State University, Tempe, AZ, USA.

His current research interests include high-performance hardware acceleration of deep learning algorithms on digital application-specified integrated circuit and field-programmable gate array.

Yu Cao (S'99–M'02–SM'09–F'17) received the B.S. degree in physics from Peking University, Beijing, China, in 1996 and the M.A. degree in biophysics and the Ph.D. degree in electrical engineering from the University of California at Berkeley, Berkeley, CA, USA, in 1999 and 2002, respectively.

He was a summer intern with Hewlett-Packard Labs, Palo Alto, CA, USA, in 2000, and the IBM Microelectronics Division, East Fishkill, NY, USA, in 2001. He was a Post-Doctoral Researcher with the Berkeley Wireless Research Center, Berkeley, CA, USA. He is currently a Professor of electrical engineering with Arizona State University, Tempe, AZ, USA. He has published numerous articles and two books on nano-CMOS modeling and physical design. His current research interests include physical modeling of nanoscale technologies, design solutions for variability and reliability, reliable integration of post-silicon technologies, and hardware design for on-chip learning.

Dr. Cao was a recipient of the 2012 Best Paper Award at IEEE Computer Society Annual Symposium on VLSI, the 2010, 2012, 2013, 2015, and 2016 Top 5% Teaching Award, Schools of Engineering, Arizona State University, the 2009 ACM SIGDA Outstanding New Faculty Award, the 2009 Promotion and Tenure Faculty Exemplar, Arizona State University, the 2009 Distinguished Lecturer of IEEE Circuits and Systems Society, the 2008 Chunhui Award for Outstanding Oversea Chinese Scholars, the 2007 Best Paper Award at International Symposium on Low Power Electronics and Design, the 2006 NSF CAREER Award, the 2006 and 2007 IBM Faculty Award, the 2004 Best Paper Award at International Symposium on Quality Electronic Design, and the 2000 Beatrice Winner Award at International Solid-State Circuits Conference. He has served as an Associate Editor of the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, and on the technical program committee of many conferences.



Sarma Vrudhula (M'85–SM'02–F'16) received the B.Math. degree from the University of Waterloo, Waterloo, ON, Canada, and the M.S.E.E. and Ph.D. degrees in electrical and computer engineering from the University of Southern California, Los Angeles, CA, USA.

He is a Professor of computer science and engineering with Arizona State University, Tempe, AZ, USA, and the Director of the NSF I/UCRC Center for Embedded Systems. He was a Professor with the ECE Department, University of Arizona, Tucson, AZ, USA. He was on the Faculty of the EE-Systems Department with the University of Southern California. He is recently investigating nonconventional methods for implementing logic, including technology mapping with threshold logic circuits; the implementation of threshold logic using resistive memory devices; and the design and optimization of nonvolatile logic. He was also the Founding Director of the NSF Center for Low Power Electronics, University of Arizona. His current research interests include design automation and computer-aided design for digital integrated circuit and systems, focusing on low power circuit design, and energy management of circuits and systems, energy optimization of battery powered computing systems, including smartphones, wireless sensor networks, and IoT systems that relies energy harvesting; system level dynamic power and thermal management of multicore processors and system-on-chip; statistical methods for the analysis of process variations; statistical optimization of performance, power and leakage; and new circuit architectures of threshold logic circuits for the design of ASICs and FPGAs.

Jae-Sun Seo (S'04–M'10–SM'17) received the B.S. degree in electrical engineering from Seoul National University, Seoul, South Korea, in 2001 and the M.S. and Ph.D. degrees in electrical engineering from the University of Michigan, Ann Arbor, MI, USA, in 2006 and 2010, respectively.

From 2010 to 2013, he was with IBM T. J. Watson Research Center, Yorktown Heights, NY, USA, where he was on cognitive computing chips under the DARPA SyNAPSE Project and energy-efficient integrated circuits for high-performance processors. In 2014, he joined the School of Electrical, Computer and Energy Engineering, Arizona State University, Tempe, AZ, USA, as an Assistant Professor. In 2015, he was with the Intel Circuits Research Lab, Santa Clara, CA, USA, as a Visiting Faculty. His current research interests include efficient hardware design of machine learning and neuromorphic algorithms and integrated power management.

Dr. Seo was a recipient of the Samsung Scholarship from 2004 to 2009, the IBM Outstanding Technical Achievement Award in 2012, and the NSF CAREER Award in 2017.