Integration, the VLSI Journal 62 (2018) 14–23

Contents lists available at ScienceDirect

Integration, the VLSI Journal

journal homepage: www.elsevier.com/locate/vlsi

ALAMO: FPGA acceleration of deep learning algorithms with a modularized RTL compiler

Yufei Ma [a,*], Naveen Suda [a], Yu Cao [a], Sarma Vrudhula [b], Jae-sun Seo [a]

[a] School of Electrical, Computer and Energy Engineering, Arizona State University, Tempe, USA
[b] School of Computing, Informatics, Decision Systems Engineering, Arizona State University, Tempe, USA

ARTICLE INFO

Keywords:
Convolutional neural networks
FPGA
RTL
Hardware acceleration
Compiler

ABSTRACT

Deploying Convolutional Neural Networks (CNNs) on a portable system is still challenging due to the large volume of data, the extensive amount of computation and frequent memory accesses. Although existing high-level synthesis tools (e.g. HLS, OpenCL) for FPGAs dramatically reduce the design time, the resulting implementations are still inefficient with respect to resource allocation for maximizing parallelism and throughput. Manual hardware-level design (i.e., RTL) can improve the efficiency and achieve greater acceleration but that requires an in-depth understanding of both the algorithm structure and the FPGA system architecture. This work presents a scalable solution that achieves the flexibility and reduced design time of high-level synthesis and the near-optimality of an RTL implementation. The proposed solution is a compiler that analyzes the algorithm structure and parameters, and automatically integrates a set of modular and scalable computing primitives to accelerate the operation of various deep learning algorithms on an FPGA. Integrating these modules together for end-to-end CNN implementations, this work quantitatively analyzes the complier's design strategy to optimize the throughput of a given CNN model under the FPGA resource constraints. The proposed RTL compiler, named ALAMO, is demonstrated on Altera Stratix-V GXA7 FPGA for the inference tasks of AlexNet and NiN CNN models, achieving 114.5 GOPS and 117.3 GOPS, respectively. This represents a 1.9*X* improvement in throughput when compared to the OpenCL-based design. The results illustrate the promise of the automatic compiler solution for modularized and scalable hardware acceleration of deep learning.

1. Introduction

Convolutional Neural Networks (CNNs) have become the *de facto* standard in many computer vision applications such as character recognition [1], image/video classification [2–6], face detection [7], and scene analytics [8], because of their ability to achieve accuracy close to or even better than human-level perception. CNNs typically consist of multiple layers of computationally intensive convolution operations followed by memory intensive classification layers, which still challenge the state-of-the-art computing platforms to achieve real-time performance with high energy efficiency.

In practice, GPUs are widely used to accelerate the training tasks of CNNs by implementing convolution as highly optimized matrix-matrix multiplication [20]. However, their cost and power consumption (>100 W) is too high for embedded applications, where energy-efficiency is critical. Even though GPUs can achieve very high throughput by computing batches of images together, their inference processing time of one image is relatively high [27], which is not suitable for latency-critical applications that require real-time classification results. Instead of GPUs, various hardware accelerators have been recently proposed based on FPGAs [9–15], SoCs [16] and ASICs [17,18] to accelerate CNN inference processes. FPGA-based CNN accelerators in particular, have become increasingly popular by virtue of their high re-configurability, fast turn-around time (compared to ASICs), good performance and better energy efficiency (compared to GPUs) [19].

Previous FPGA implementations based on high-level synthesis (HLS) tools have achieved good flexibility, easy programmability and short design time, but their hardware and memory utilization is inefficient and may not allow exploitation of low-level hardware structures to achieve higher performance and throughput [10,11,13,28].

On the other hand, previous RTL based FPGA efforts implemented customized CNN accelerators optimized for a specific CNN model [14,31,

* Corresponding author. 660 S Mill Ave, Tempe, AZ 85281, USA.
E-mail addresses: yufeima@asu.edu (Y. Ma), nsuda@asu.edu (N. Suda), yu.cao@asu.edu (Y. Cao), vrudhula@asu.edu (S. Vrudhula), jaesun.seo@asu.edu (J.-s. Seo).

https://doi.org/10.1016/j.vlsi.2017.12.009
Received 1 August 2017; Received in revised form 30 October 2017; Accepted 15 December 2017
Available online 5 January 2018

32]. By this means, the fine-grained hardware level optimization can be exploited to achieve high performance and energy efficiency. However, the customized RTL accelerator requires long development time, e.g. several months, which makes it difficult to catch up with the rapidly evolving CNN algorithms and diverse applications. A generic RTL based CNN accelerator, which is agnostic to the model configuration (i.e., the feature maps and kernel sizes) [9,16], is another design option that increases the accelerator flexibility for different CNNs at the cost of inefficient hardware utilization. This approach is necessary for the ASIC design to serve different CNN algorithms on the same hardware, but not that important for the FPGA platform that is highly reconfigurable. Furthermore, the rapidly changing field of deep learning algorithms and tasks makes it even more difficult for a generic accelerator to efficiently cater a wide range of CNN algorithms.

On the software side, machine learning researchers have been able to efficiently develop and explore deep learning algorithms through flexible open-source frameworks such as Caffe [20], Torch7 [21], Theano [22], and TensorFlow [23], which run on CPUs or GPUs. These software frameworks have simple expression and modularity, which allowed researchers to efficiently explore various algorithms and network structures. Unfortunately, the hardware design community does not yet have such a flexible modular framework for hardware implementation of CNN and other deep learning algorithms, inevitably spreading out the hardware research efforts instead of coalescing them.

In this context, there is a timely need to reform the strategy and its implementation for mapping CNNs to physical hardware, and to support modular and scalable hardware customization for specific applications without sacrificing design flexibility. To this end, we propose a scalable FPGA framework, referred to as ALAMO, that stands for Acceleration of deep Learning Algorithms with a MOdularized RTL compiler [30]. ALAMO targets and provides an automatic means to map CNN inference processes to efficient RTL codes that can be used for high-performance FPGA and ASIC implementations.

This review article provides a detailed review of ALAMO compiler from our previous conference paper [30]. It also includes new analysis and technique proposed in our more recent publications [31,32] for customized CNN accelerator, whose design can be applied to further improve the performance, efficiency and flexibility of ALAMO. The main contributions include:

(a) The proposed ALAMO RTL compiler is shown in Fig. 1. It analyzes a given CNN model's structure and sizes, as well as the degree of desired parallelism set by users, to generate and integrate variants of CNN modules based on the hand coded parameterized RTL module templates.
(b) The acceleration system generated by the compiler is validated by implementing two deep CNNs each requiring over a billion operations per input image – AlexNet [4] and Network in Network (NiN) [5] on DE5-Net board with Altera Stratix-V FPGA, achieving a throughput of 114.5 GOPS and 117.3 GOPS, respectively.

The rest of the paper is organized as follows. Section 2 provides a brief overview of operations and structure of a typical CNN model. Section 3 highlights the practical challenges in FPGA implementation of large-scale CNNs and studies the implementation of CNN acceleration modules with limited precision. Section 4 describes the strategy to accelerate convolution which dominates CNN operations. Section 5 presents the design of scalable modules for CNN building blocks e.g. convolution, pooling, normalization and fully-connected inner-product, generated by the compiler. The accelerator system that integrates the scalable CNN RTL modules is discussed in Section 6. Section 7 presents the experimental results of ALAMO implementation of AlexNet [4] and NiN [5] models on an FPGA and compares them with prior works. The paper is concluded in Section 8.

2. Overview of CNN operations and structures

Convolutional neural networks typically incorporate multiple layers of convolution, pooling/subsampling, and normalization that extract low-level to high-level features from the input. These features can be categorized into a finite number of output classes by the final classification layers such as the multi-layer perceptron or fully-connected layers.

Convolution involves 3-dimensional multiply and accumulate (MAC) operations of N_{if} input feature maps (or channels) with N_{if} convolution kernels (or filters) of size $K_x \times K_y$ to compute an output pixel (or feature), as described in Eq. (1) and illustrated in Fig. 2.

$$out(f_o,x,y)=\sum_{f_i=0}^{N_{if}}\sum_{k_x=0}^{K_x}\sum_{k_y=0}^{K_y} wt(f_o,f_i,k_x,k_y)\times in(f_i,x+k_x,y+k_y) \quad (1)$$

where $out(f_o,x,y)$ and $in(f_i,x,y)$ represent the pixel or neuron values at (x,y) position in the feature maps f_o and f_i, respectively, $wt(f_o,f_i,k_x,k_y)$ are the kernel weights at position (k_x,k_y) that are convolved with the input feature map f_i to get the output feature map f_o. A special convolution layer, named 'cccp', is used in NiN with $K=1$. Convolutions can constitute more than 90% of the total operations for complex CNNs models [4–6,26].

Pooling or subsampling is commonly employed after convolution to reduce the dimensionality of the input features while preserving key information. This is done by replacing $K_p \times K_p$ input neurons with their maximum or average value as shown in Fig. 2, depending on the model definition.

Normalization or local response normalization (LRN) implements a form of lateral inhibition [4] by normalizing the neuron value by a factor depending on its neighboring features, as shown in Eq. (2).

$$out(f_o,x,y)=\frac{in(f_o,x,y)}{\left(1+\frac{\alpha}{K}\sum_{f_i=f_o-K/2}^{f_o+K/2} in^2(f_i,x,y)\right)^{\beta}} \quad (2)$$

Fully-connected (FC) or inner-product layers are final classification layers where the output features are computed as matrix-vector multiplications of the fully-connected weight matrix and the input feature

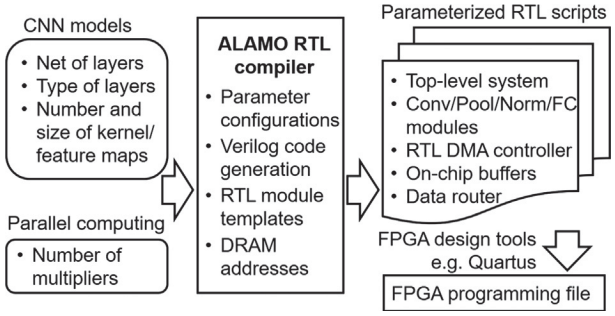

Fig. 1. The compilation flow with the proposed ALAMO RTL compiler.

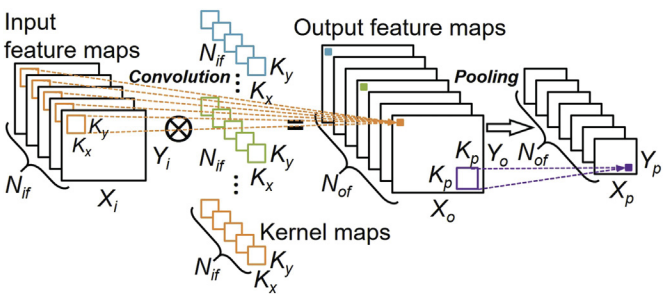

Fig. 2. 3-D convolution and pooling with their corresponding parameters.

vector. Convolution and fully-connected layers are often followed by non-linear activation functions such as tanh, sigmoid or Rectified Linear Unit (ReLU). ReLU, defined as $y = max(x,0)$, has become a popular choice for the activation function, due to faster convergence in training [4] as well as compact hardware implementation.

Structures of two representative CNN models, AlexNet and NiN, for image classification task in the ImageNet challenge [2] are shown in Fig. 3. They have relatively small sizes of intermediate feature results and can be stored in the FPGA on-chip memory. The dimensions of output feature maps ($N_{of} \times X_o \times Y_o$) are denoted on the right side of the layer name and the kernel sizes ($N_{if} \times K_x \times K_y$) are shown outside the box on the left. AlexNet consists of 5 convolution layers, 2 normalization layers, 3 pooling layers, 3 fully-connected classification layers and 7 ReLU-based activation layers. On the other hand, NiN consists of 4 convolution layers, 8 cccp layers, 4 pooling layers, and 12 activation layers. Note that, in NiN, the fully-connected layers are replaced by the global average pooling layer (pool4), which save a large number of weights. The eight cccp layers with 1×1 convolution are employed for better local abstraction, and there are no normalization layers.

3. Hardware design challenges

Implementing state-of-the-art CNN models on embedded platforms with performance, power, and area constraints is a challenging task, mainly because of the computational complexity in the convolution layers and the memory requirements of the fully-connected layers. For instance, AlexNet CNN [4] has over 60 million parameters and needs to perform 1.46 billion operations on each input image. Using 32-bit floating-point representation from the original algorithm, approximately 250 MB of memory is required to store all he weights. This exceeds the on-chip memory capacity of present-day commercial FPGAs. Consequently, these weights have to be stored in off-chip memory (e.g. DRAM) and transferred to the FPGA during computation, making the external memory bandwidth as a performance bottleneck, especially in the final classification layer, where a large number of weights are required to compute a single output neuron value.

The scale and complexity of the CNN algorithms keep increasing as new applications emerge, and requiring ever increasing accuracy. For example, AlexNet [4], which achieved the top classification accuracy in ImageNet challenge 2012, has 8 layers when counting only the layers with parameters. In comparison, GoogLeNet [6] (winner of the ImageNet challenge in 2014) has 22 layers, and ResNet [25] (winner of the ImageNet challenge in 2015) has 152 layers. Considering such a trend, it would be impossible to implement all the CNN layers on an FPGA without reusing the hardware resources. In order to efficiently share FPGA resources, CNN layers should be implemented as scalable hardware acceleration modules, so that the same hardware is reused by directing the appropriate data through it. The energy cost of FPGA accelerators mainly comes from computation, data movements and memory access. Memory that is higher in the hierarchy (e.g. DRAM) normally consumes greater energy per access than memory that is lower in the hierarchy (e.g. register). This makes it essential to process data as locally as possible while minimizing memory accesses for energy efficient design.

CNN models are typically trained with 32-bit floating point precision in CPU/GPU based systems using an open-source machine learning framework such as Caffe, Theano and TensorFlow. Although 32-bit floating point precision (or 16-bit fixed point with statistical rounding [24]) may be required during training for convergence, such a high precision is not typically necessary during the classification phase to achieve same accuracy levels [13,14]. Using weights with reduced precision alleviates the performance limitation due to the external memory bandwidth bottleneck, on-chip memory footprint, and power consumption for external memory transfers. Moreover, it enables the use of fixed-point arithmetic operations, which require considerably fewer FPGA logic resources compared to floating point operations, thus improving the hardware efficiency.

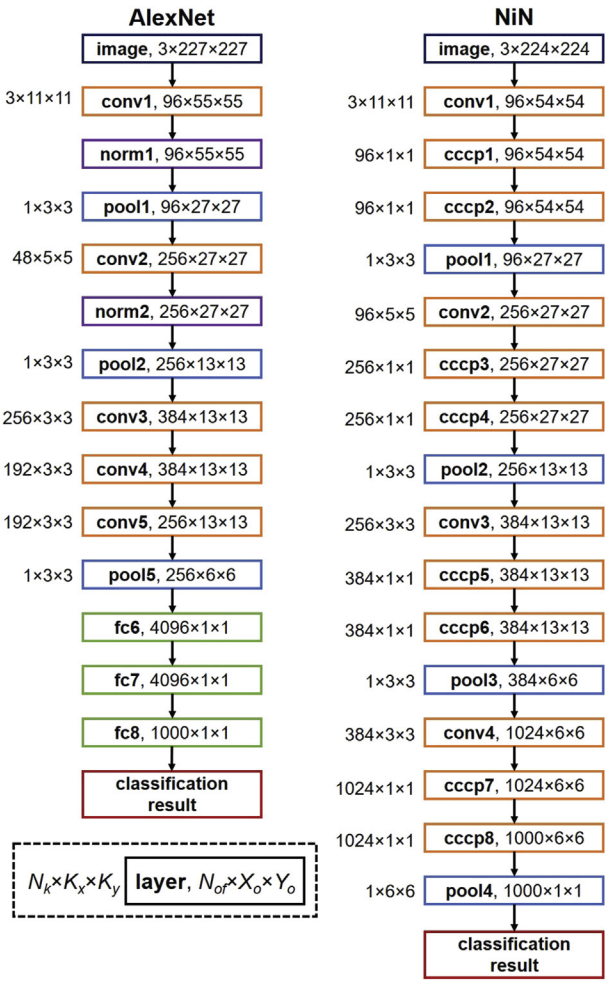

Fig. 3. Structures of AlexNet and NiN CNN models.

4. Strategy to accelerate convolution

4.1. Four levels of convolution loops

Convolving feature maps with kernels is essentially a 3-D MAC operation, which is comprised of four levels of loops as shown in the pseudo codes in Fig. 4. Loop-1 is the inner most loop that computes the MAC within a kernel window of dimension $K_x \times K_y$. Loop-2 iterates across different input feature maps of dimension of N_{if}. Loop-3 slides the kernels within an input feature map of dimension $X_i \times Y_i$ and the outermost Loop-4 generates different output feature maps with dimension of N_{of}.

4.2. Convolution loop unrolling

Loop unrolling [31] directs the parallel computation of different convolution loops as illustrated in Fig. 5, which determines the computation pattern and the data flow. If Loop-1 is unrolled, $P_{kx} \times P_{ky}$ pixels and

Loop-4 Across the output feature maps of N_{of}
Loop-3 Slide within one input feature map with $X \times Y$
Loop-2 Across the input feature maps of N_{if}
Loop-1 MAC within a kernel window of $K_x \times K_y$

Fig. 4. Pseudo codes to implement iterative convolution operations.

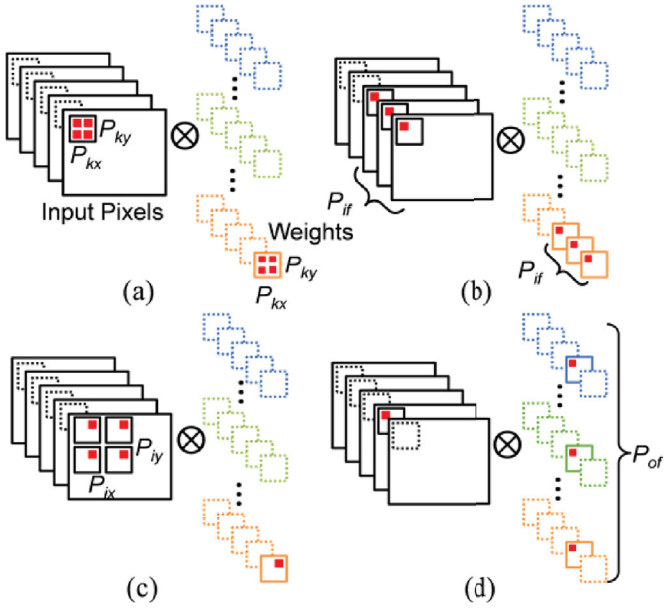

Fig. 5. Loop unrolling: (a) unroll Loop-1; (b) unroll Loop-2; (c) unroll Loop-3; (d) unroll Loop-4 [31].

weights from the same feature and kernel map are computed every cycle as shown in Fig. 5(a). If Loop-2 is unrolled by a factor of P_{if}, pixels from P_{if} different input feature maps are multiplied in parallel with the associated kernel weights as shown in Fig. 5(b). If Loop-3 is unrolled, $P_{ix} \times P_{iy}$ number of pixels in the same feature map are multiplied with the identical weight as shown in Fig. 5(c). If Loop-4 is unrolled, P_{of} adjacent output channels are computed in parallel using the shared input pixel as shown in Fig. 5(d).

In Ref. [30], the acceleration of convolution operations is realized by unrolling Loop-2 and Loop-4 as illustrated by Fig. 6. The number of parallel multipliers (N_m), which constrains the degree of acceleration (i.e. $P_{if} \times P_{of} \leq N_m$) and hardware resource usage (e.g. DSPs or logic elements), can be configured by users to fit the CNN accelerator into their FPGA chip. In order to execute the maximum number of parallel multiplications with limited hardware resources, multipliers are shared among all convolution layers. Since the dimensions of kernel and feature maps could vary significantly across all the convolution layers even in one CNN algorithm, our loop unrolling factors are dynamically adjusted for layers with different numbers of channels to increase the computing resource utilization. In particular, P_{if} and P_{of} are adjusted for each layer to make $P_{if} \times P_{of}$ close or equal to N_m.

If the multiplier number N_m is larger than the number of input

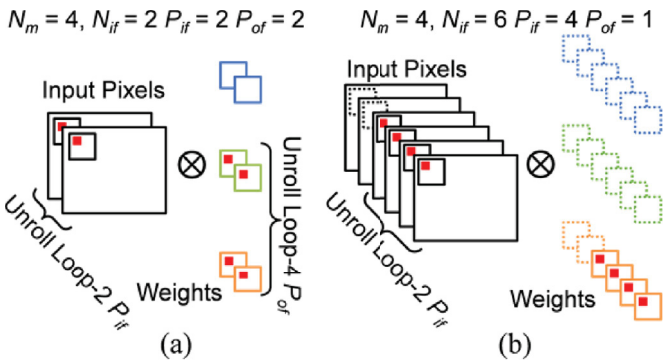

Fig. 6. Convolution acceleration strategy, (a) $N_m > N_{if}$, unroll both Loop-2 and Loop-4, (b) $N_m \leq N_{if}$, only unroll Loop-2.

channels N_{if}, then Loop-2 can be fully unrolled (i.e. $P_{if} = N_{if}$) as well as Loop-4. This is shown in Fig. 6(a). The multiplication results are summed up by the following adder trees with fan-in of P_{if}, and subsequently the pixels at the next location are fetched and computed. After the sequential sliding within one kernel window $K_x \times K_y$ is completed, we add the bias and compute the final output pixel value. Therefore, $K_x \times K_y$ cycles are required to compute an output pixel and in total $X_o Y_o K_x K_y$ cycles to compute one output feature map. With $N_m > N_{if}$, we can unroll Loop-4 to compute $P_{of} = \lfloor N_m / N_{if} \rfloor$ output feature maps in parallel, and we need $\lceil N_{of}/P_{of} \rceil$ iterations to compute the entire output feature maps of N_{of}. Therefore, the total latency for one convolution layer is $X_o Y_o K_x K_y \times \lceil N_{of}/P_{of} \rceil$. If N_m is smaller than N_{if}, Loop-2 can only be partially unrolled (i.e. $N_m = P_{if} < N_{if}$, $P_{of} = 1$) as shown by Fig. 6(b). Then, the compiler simply repeats the $K_x \times K_y$ kernel window sliding $\lceil N_{if}/P_{if} \rceil$ times, and repeats the memory read operations $\lceil N_{if}/P_{if} \rceil$ times to read pixels from the same (x,y) feature map location but from the next group of input feature maps. In this case, the total delay for one layer in the number of cycles is $X_o Y_o K_x K_y N_{of} \times \lceil N_{if}/P_{if} \rceil$.

If both Loop-2 and Loop-4 are fully unrolled, $P_{if} \times P_{of} = N_{if} \times N_{of}$ number of parallel multiplications are required, which is greater than 8000 for all but the first the convolution layer in recent CNN models [4–6,25,26]. This is well beyond the capability (the number of DSP blocks and logic elements) of the most advanced present-day FPGAs. Therefore, only Loop-2 and Loop-4 are unrolled in Ref. [30].

In order to improve the hardware utilization and throughput of the first convolution layer, Loop-3 and Loop-4 are unrolled in Ref. [31] instead, which can provide enough parallelism. In addition, the kernel weight can be reused by multiple features after fetched from buffers by unrolling Loop-3.

4.3. Sequential computation order

To obtain one final output pixel, we need to complete Loop-1 and Loop-2. As Loop-2 is already unrolled, we sequentially compute Loop-1 first to get the output pixels and consume the partial sums as early as possible to reduce its storage requirement. After finishing the sequential sliding of Loop-3 that computes output pixels within the same feature map, we can get one subset of P_{of} output feature maps, and then iterate this process across Loop-4 to compute all the output feature maps. To enable the parallel read of the pixel values at the same location (x,y) across different input feature maps, N_{of} output feature maps from the previous layer are stored into N_{of} separate memories to be used as the input of the current layer, and $X_o \times Y_o$ pixels of each feature map are continuously stored within one memory bank.

5. Compilation and parallel computation of scalable CNN modules

In this section, we present the detailed designs of the key RTL module templates customized for different types of layers, e.g. convolution, pooling and LRN. These modules are highly parameterized and designed to maximize the amount of parallel computation constrained by the limited hardware resources and memory bandwidth.

The ALAMO compiler that we developed analyzes the given CNN structure and the parameters of each layer to configure the module templates and their connections accordingly.

5.1. Scalable convolution module (CONV)

The convolution RTL modules (CONV) that can be scaled by changing the parameters, e.g. K_x, K_y, X_i, Y_i, X_o, Y_o, N_m, N_{if}, and N_{of}, of each convolution layer, are automatically generated and configured by the compiler based on the acceleration strategy described in Section 4.

This module mainly consists of dedicated control logic and parallel adder trees, whereas the multipliers, which serve as the main computing engines, are shared across all the layers as shown in Fig. 7. The input data

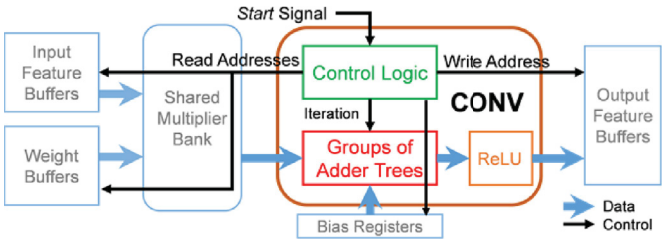

Fig. 7. Convolution acceleration module block diagram.

(i.e., the input pixels and kernel weights) for the convolution are stored in the on-chip buffers, and the multiplier results are sent to CONV to perform summation and accumulation. The outputs of CONV are conveyed to N_{of} separate on-chip memories and will be used for the next layer.

The control logic inside the convolution module consists of counters that control the sliding of the convolution within the kernel window and the feature map. Each counter is parameterized with respect to the kernel sizes (K), feature map dimensions (X, Y) and the number of iterations $\lceil N_{of}/P_{of} \rceil$, which correspond to the sliding of Loop-1, Loop-3 and Loop-4, respectively. For different layers, these counter parameters are adjusted and generated by the compiler. The combination of these counters along different dimensions generates the memory address to read the input pixels, which is $k_x + k_y \times (X_i{+}2 \times \text{PAD}) + \text{STRIDE} \times x + \text{STRIDE} \times y \times (X_i{+}2 \times \text{PAD})$. Both k_x and k_y are signals generated by the counters as they iterate from 0 to $K-1$ to slide within a kernel window in X and Y directions, and k_y increases by one only when k_x equals to $K-1$. Similarly, x and y iterate from 0 to $X_i - 1$ and $Y_i - 1$, respectively, for computations within an input feature map. PAD denotes the size of zero padding, and STRIDE is the step size by which input feature map slides. The write address simply iterates from 0 to $X_o \times Y_o - 1$ as the output features are continuously stored. Between the input read and output write operations of CONV, there are several cycles of delay caused by memory read, MAC operation and data routing pipelined by DFFs. An additional counter that counts the iteration number from 0 to $\lceil N_{of}/P_{of} \rceil - 1$ is used to control Loop-4 unrolling across output feature maps and generates *end* signal to initialize the next layer or the transfer of data from external memory.

Different layers may have different N_{of} or N_{if} that makes N_{of}/P_{of} or N_{if}/P_{if} not necessarily to be integers across all the convolution layers. Therefore, some layers cannot fully utilize all the existing multipliers for every iteration. To that end, the compiler supports the configurability of allocating a portion of the multiplier bank for a specific CONV module, while minimizing the number of unused multipliers.

As Loop-2 is unrolled in Ref. [30], the CONV module requires parallel adder trees followed by an accumulator, which reads the output from multipliers and accumulates them within one kernel window and subsequently through all input feature maps as illustrated in Fig. 8(a). As the kernel window is sequentially slid across the input pixels, the fan-in of the adder tree equals the number of parallel computed input channels (P_{if}), and the number of adder trees is determined by the number of output feature maps computed in parallel (P_{of}). Therefore, if the convolution layers have different sizes of P_{if}, e.g. in Table 1, ALAMO must compile various CONV modules with different adder trees and control logic for different layers, which significantly increases the overhead of hardware resources. In Ref. [31], only Loop-3 and Loop-4 are unrolled, which uses MAC units to independently compute each output without using adder trees as shown in Fig. 8(b). By this means, it is possible to design one CONV module used for different convolution layers with uniform unrolling factors.

The ReLU module, which follows the adder trees, checks the sign bit of the pixel values and only retains the positive ones, while converting the negative values to be zero.

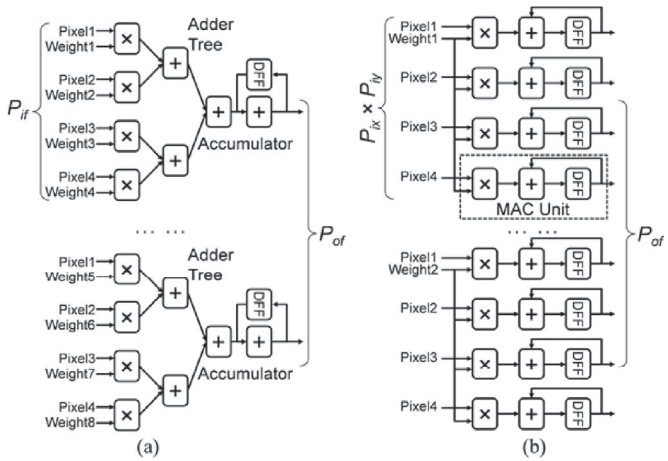

Fig. 8. Convolution arechitecture: (a) unroll Loop-2 with P_{if} and Loop-4 with P_{of} [30]; (b) unroll Loop-3 with $P_{ix} \times P_{iy}$ and unroll Loop-4 with P_{of} [31].

Table 1
Unrolling Configurations of each layer.

AlexNet	N_{if}	P_{if}	P_{of}	# used Mult.	Utilization ($N_m = 1152$)
conv1	3	3	96	228	25.0%
conv2	48	48	24	1152	100%
conv3	256	256	4	1024	88.9%
conv4	192	192	6	1152	100%
conv5	192	192	6	1152	100%
NiN	N_{if}	P_{if}	P_{of}	# used Mult.	Utilization ($N_m = 768$)
conv1	3	3	96	288	37.5%
cccp1	96	96	8	768	100%
cccp2	96	96	8	768	100%
conv2	96	96	8	768	100%
cccp3	256	256	3	768	100%
cccp4	256	256	3	768	100%
conv3	256	256	3	768	100%
cccp5	384	384	2	768	100%
cccp6	384	384	2	768	100%
conv4	384	384	2	768	100%
cccp7	1024	512	1	512	66.7%
cccp8	1024	512	1	512	66.7%

5.2. Pooling module (POOL)

The structure of the pooling module (POOL) is similar to CONV with its own control logic and computing engine as illustrated in Fig. 9. Two POOL variants are designed for max-pooling (POOL-MAX) and average-pooling (POOL-AVE). The control logic for both variants is the same, which generates memory read/write address to scan the input feature maps. The computing engine is either a comparator for max-pooling or an accumulator followed by a multiplier for average-pooling. The division in the average operation is replaced by constant coefficient multiplication. As POOL module only requires input pixels from the previous layer, it can be executed directly after its previous layer without accessing to external memory. Similar to convolution, the parallel computation in pooling is across different feature maps that uses a common read/write address

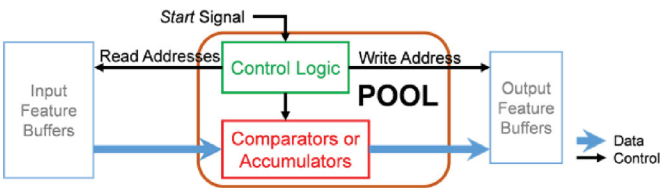

Fig. 9. Pooling module block diagram.

signal. Module POOL allows for a trade-off between throughput and hardware resources by changing the number of feature maps that are computed in parallel (N_p). The latency of one pooling layer is $X_o Y_o K^2 \times N_{of}/N_p$.

5.3. Normalization module (NORM)

Local response normalization (LRN) computes the nearby pixels within a local region. The structure of the module NORM, shown in Fig. 10, assumes this local region spans only the neighboring feature maps. As both CONV and POOL are designed to output the pixels across the adjacent feature maps in parallel, this allows the NORM module to directly read input data from the previous layer and overlap its operation with the previous layer's computation. If the previous layer is CONV, this will require unrolling of Loop-4. This way the memory to store the intermediate pixels can also be saved. The square operations of input pixels from adjacent feature maps are computed in parallel and the results are shared by its nearby pixels during the sum operation. After that, the nonlinear operation is performed using a look-up table that associates the input to a corresponding output value, which is then multiplied with the original pixel. The NORM module generates the memory write address signal in a similar manner to CONV module by adding its own pipeline delay.

5.4. Inner-product module (FC)

The inner-product or fully-connected (FC) layer essentially performs the matrix-vector multiplication, which also consists of MAC operations as in the convolution layers. Therefore, the FC module shares the multiplier bank with CONV modules, while having its own adders to perform accumulation that are shared across all fully-connected layers. The parallel MAC operations are performed to compute multiple output pixels in parallel so that one input pixel can be shared by multiple rows of the kernel weight matrix and no parallel adder trees are needed. As the fully-connected layers do not share the kernel weights for different features, the data volume of the weights is significantly larger than that of convolution layers, even though the FC layer has much less operations. This makes the throughput of the FC layer primarily dependent on the transfer speed of the external memory to on-chip buffers. Due to this, dual on-chip buffer banks are employed to temporarily store the FC kernel weights to allow overlapping the DMA transfer operation with the inner product computation, such that the latency of the FC layers equals to the kernel weights transfer time.

5.5. DMA configuration module

A Direct Memory Access (DMA) engine is used to transfer data between external and on-chip memories. The Modular Scatter-Gather DMA (mSGDMA) IP provided by Altera is used to efficiently handle small, frequent and non-continuous data movements with preload instructions. A custom DMA configuration module is designed to control the mSGDMA behavior by writing the instructions with different transfer sizes, source and destination addresses for different CONV and FC modules. It also generates the *start* signal for CONV and FC upon the completion of DMA data transactions.

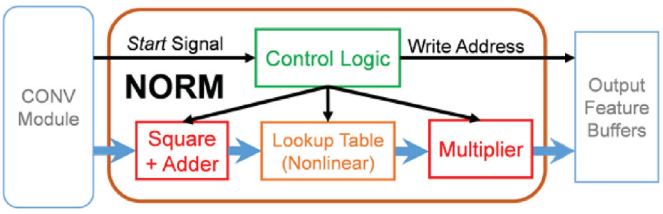

Fig. 10. Local response normalization module block diagram.

6. FPGA implementation

In this section, the top-level FPGA system architecture and the compiler strategy to integrate the generated scalable CNN modules are described.

6.1. System overview

The top-level FPGA-based CNN accelerator system is shown in Fig. 11. The system includes NIOS-II, a processor IP from Altera, which is only used to transfer the weights and input images from the on-board flash memory to the external DDR3 SDRAM. DDR3 controller and mSGDMA IPs from Altera's IP library are used to perform memory transfers from external DDR3 to FPGA on-chip memory. All these standard IPs are connected to Altera's Avalon memory mapped interface bus. After all the kernel weights are loaded into DDR3, the DMA configuration module present in the accelerator initializes and controls the mSGDMA and starts the CNN acceleration process.

6.2. Integration of CNN acceleration modules

The integration of the overall acceleration system is shown in Fig. 12, including the proper connections between the building block modules, and the control of the data flow to sequentially perform the iterations in each layer. Furthermore, the memory system is designed to store kernel weights and features on external or on-chip memories, and the feature data routers are configured to convey feature data and address signals from different modules into feature buffers, and fetch the stored data to the multiplier bank or directly into the corresponding modules (e.g. POOL).

6.2.1. Controller

The layer by layer sequential computation is controlled by the *start* and *end* signals generated by computation modules and the DMA configuration module. If the next module requires data from DDR3, the *end* signal of the current module initializes the DMA transfer, and after the transfer is finished, a *start* signal from DMA configuration module is sent to the next module. Alternatively, the *end* signal can directly start the next module.

6.2.2. Weight storage

The kernel weights for both CONV and FC modules are stored in the external DDR3 DRAM. As described in Section 5.4, the transfer of FC kernel weights is performed in parallel with the FC MAC operations that use kernel weights from the previous DMA transfer. On the other hand, considering the relatively small size of convolution kernel weights, only the necessary kernel weights of the next convolution layer are transferred

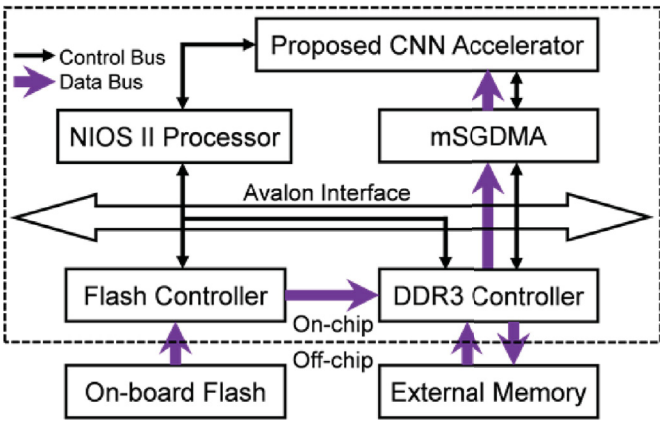

Fig. 11. Top-level CNN accelerator system [30].

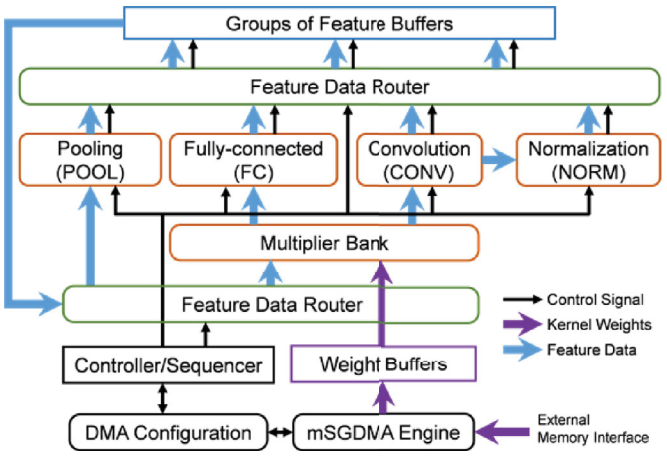

Fig. 12. Integration of scalable CNN modules [30].

to the on-chip CONV weight buffers between the executions of two convolution layers. The capacity of CONV weight buffers only needs to be able to cover the convolution layer that has the largest kernel size. However, if the kernel size of one layer is still too large, the compiler is designed to tile the execution of this layer and iterate the corresponding DMA transfer multiple times to reduce the size of CONV weight buffers. The CONV module is divided across Loop-4 and can be treated as different convolution layers yet having the same N_{if} number, so that these variant CONV modules with multiple sets of control logic could still share and re-use the adder trees.

6.2.3. Feature buffers

After quantization to form fixed point data, the intermediate pixels of many practical CNN models [4,5] can be fit into the on-chip block memory (BRAM) of many middle to high end FPGAs, e.g. exceeding 4 MB. Therefore, [30] stores the pixel results only in on-chip buffers to eliminate the external memory access. The output pixel data of each module are stored in the feature buffers in Fig. 12, and these data are fetched by the following layers as the input. The feature buffers are implemented as groups of on-chip memories. Each module stores its N_{of} output feature maps into N_{of} separate on-chip memories with the memory depth of $X_o \times Y_o$. The on-chip memories are implemented using M20K memory blocks, which has 20 K bits capacity per block. Only when the memory depth exceeds than the maximum depth of one M20K block, one more M20K block will be added. This way, the on-chip memories are stacked with shallow depth to achieve the maximum utilization of M20K blocks. Moreover, layers that are not consecutive can also share the memories. If the memory bandwidth requirement is high but the total capacity demand is low, the memory logic array block (MLAB) RAMs using logic elements are employed instead of M20K. In Ref. [30], pixel results are only stored in on-chip RAMs, which limits the flexibility of ALAMO to CNNs with only small volumes of intermediate results. In Ref. [31], the pixel outputs are stored in external DRAM, and the loop tiling technique is used to divide the entire data of one layer into multiple tiles, which can be accommodated in the on-chip BRAMs. By this means, the large-scale CNNs, e.g. VGG [26] and ResNet [25], can also be processed as in Ref. [31,32], which could significantly improve the flexibility of ALAMO.

6.2.4. Data router

The feature data router, implemented by groups of multi-level multiplexers, differentiates the read/write signals from two adjacent modules into the feature buffers by the *start* signal of the latter module. The multiplier bank in the CNN accelerator, which is the main computing resource, is shared among all CONV and FC modules. Another feature data router is used to select the feature buffer output data of different computing modules to the shared multipliers or POOL modules, and then the output results of the multipliers are allocated to corresponding CONV and FC modules. As many signals are merged and distributed by the feature data router, signal congestion and timing failure could occur. Therefore, the data paths in the router are pipelined, where the number of pipeline stages is parametrized, to reduce the combinational logic critical path and complete timing closure.

7. Experimental results

7.1. Experimental setup

The proposed ALAMO RTL compiler [30] is demonstrated by implementing AlexNet and NiN CNN models on a DE5-Net board with two 4 GB DDR3 DRAMs as the external memory and Altera Stratix-V GXA7 FPGA, which consists of 622 K logic elements, 256 DSP blocks and 2560 M20K RAMs. First, the compiler analyzes the connection and dimensions of all the layers in the AlexNet and NiN CNNs with the number of multipliers (N_m) to generate and configure the parameterized CNN computing modules and integrates them as discussed in Section 6. The entire CNN acceleration system is synthesized by Altera Quartus tool and then downloaded to the FPGA. After that, the weights and input images, which are stored initially in the on-board flash memory, are transferred to the external DDR3 memory and starts the acceleration process. After going through the layered computations, the final outputs of the CNN (i.e., the sorted top-5 predictions) are read from the output of the on-chip memory and compared against the validation data to verify the functionality of the accelerator.

7.2. Parallelism configurations

The configurations for various amounts of unrolling are listed in Table 1 for all the convolution layers in AlexNet and NiN, respectively. As described in Section 4.2, we first unroll the computation of P_{if} input channels, and if there are extra multipliers, P_{of} output channels are further unrolled. Since the number of input and output channels vary significantly across different layers, P_{if} and P_{of} are dynamically adjusted to make the number of used multipliers ($P_{if} \times P_{of}$) close or equal to the number of existing multiplier (N_m). As Stratix-V GXA7 has only 256 DSP blocks, which can be implemented as 512 18-bit × 18-bit multipliers, to achieve higher throughput with more parallel MAC operations, multipliers are implemented not only by DSP blocks (Fig. 15) but also by logic elements (Fig. 14). By this means, our N_m can be set to be > 512. As NiN has more convolution layers than AlexNet, more logic elements are demanded for different CONV modules. Therefore, the number of multipliers in NiN ($N_m = 768$) must be decreased compared to that of AlexNet ($N_m = 1152$) so that the accelerator can be fit into to the FPGA chip at the cost of less parallelism. Since conv1 has only three input channels, notably limiting the multiplier utilization and parallel degree, we plan to unroll other loops as in Ref. [31], to improve the throughput of conv1.

7.3. Detailed experimental results

7.3.1. Performance

The timing breakdown of different CNN layers of AlexNet and NiN models is shown in Fig. 13. The DMA_CONV delay shown in Fig. 13 corresponds to the convolution weight transfer time, whereas the FC weight transfer is overlapped with matrix-vector multiplication. The DMA transfer delay is measured by counting the number of clock cycles from real experiments and then multiplying with the clock periods. The performance and throughput numbers are also reported in Table 3.

7.3.2. Hardware utilization

The utilization breakdown of logic elements, DSP blocks, and on-chip RAM for both AlexNet and NiN are shown in Figs. 14–16, respectively. As the multipliers are implemented not only by DSP blocks but also by logic

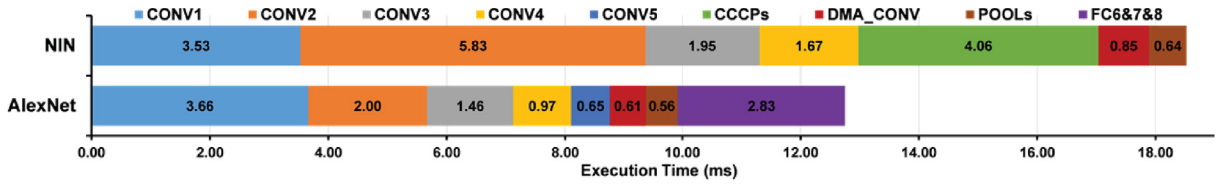

Fig. 13. Timing breakdown of different layers of AlexNet and NiN [30].

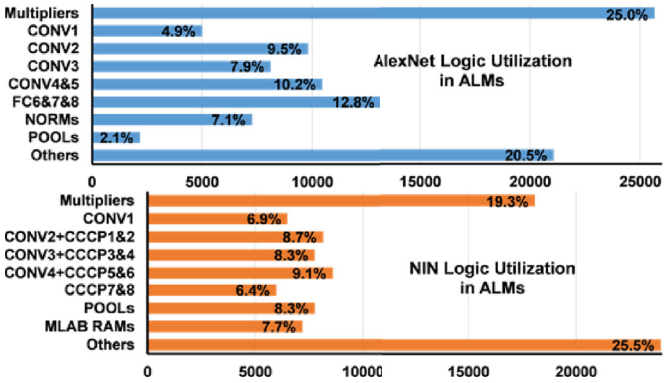

Fig. 14. Logic utilization breakdown of AlexNet and NiN modules.

elements, a significant portion of the overall logic is consumed by the hardware expensive multipliers as shown in Fig. 14. CONV4 and CONV5 in AlexNet with the same N_{if} are combined to be one module to share the adder trees, and this occurs more frequently in NiN such as the combination of CONV2, CCCP1 and CCCP2 layers. The feature data routers, DMA configuration module, mSGDMA engine, DDR3 controller, and NIOS II soft processor are included in "Others" in Fig. 14. CONV_Kernel and FC_Kernel in Fig. 16 denote the weight buffers to receive kernel weights from external memory. As outputs from CONV1 and CONV2 in AlexNet directly go into NORMs, there is no need to store these results. RAMs are stacked for modules with shallow word depths (e.g. NORM1 and POOL1 in AlexNet) and reused by non-consecutive modules (e.g. CONV1 and CCCP2 in NiN). The results of CONV4, CCCP7, and CCCP8 that have small data size but require high memory bandwidth by the next layer in NiN are stored in MLAB RAMs (Fig. 14).

7.3.3. Accuracy

Aimed at accelerating the inference process, fixed point data representation is used to reduce the FPGA hardware resource usage at the cost of minimal accuracy loss. With less data width, the logic elements used for multipliers, adder trees, accumulators and data buses can be reduced, as well as the on-chip RAMs used for buffers. Consequently, the data width can be decreased until considerable accuracy drop occurs. All the feature values after ReLU are 10-bit unsigned binary numbers and the kernel weights are 8-bit signed binary numbers. The multiplier outputs are 18-bit, and the adder tree and following accumulator outputs are 25-bit. The data width and the portion of integer and fractional bits can be dynamically adjusted by the compiler for each layer. If one layer has small data, more bits are allocated to fractional part while keeping the total data width equivalent to that of the other layers. This way all the data widths of the computing hardware, memory bandwidth and data bus can be fully utilized. The top-1 and top-5 ImageNet accuracies of FPGA implementation of AlexNet and NiN for the first 5 K images from ImageNet 2012 validation database are summarized in Table 2, compared to the implementation in software with full-precision. The accuracy degradation of the proposed modular FPGA design with fixed-point precision is less than 1%.

7.3.4. Power

The power consumptions of the DE5-Net board including the Stratix-V FPGA chip when running the AlexNet and NiN are measured as 19.5 W and 19.1 W, respectively, while the initial power consumption of the board at power on is 16.5 W. Using data activity information (*.vcd) obtained from the top-level testbench simulation on FPGA-mapped netlists, we analyzed the power consumption among different modules when running AlexNet, which is shown in Fig. 17. Note that the power analysis in Fig. 17 excludes the power of the off-chip SDRAM and other board-level components.

7.4. Comparsion with related works

The performance of the accelerator is summarized and compared to prior CNN accelerators in Table 3 along with the percentage of the FPGA capacity.

7.4.1. Parallelism strategy

In Ref. [33], the accelerator architecture allows flexible dataflow and unrolling factors to improve the computing resource utilization for varying layer dimensions, which is realized by simplifying the PE interconnections and designing a data distribution structure.

7.4.2. Automation-based accelerator

With ~3*X* more DSP blocks, considerable logic elements can be saved in Ref. [14,15] and this improves the performance. As the parallelism is exploited within a kernel window, the varying kernel sizes significantly degrade the hardware utilization. Therefore, the more regular structure of the VGG model [26] (compared to AlexNet and NiN) with uniform

Table 2
ImageNet accuracy comparison.

Model accuracy comparison	Software (Caffe tool)		FPGA implementation	
CNN model	Top-1	Top-5	Top-1	Top-5
AlexNet	56.78%	79.72%	55.64%	79.32%
NiN	56.14%	79.32%	55.74%	78.96%

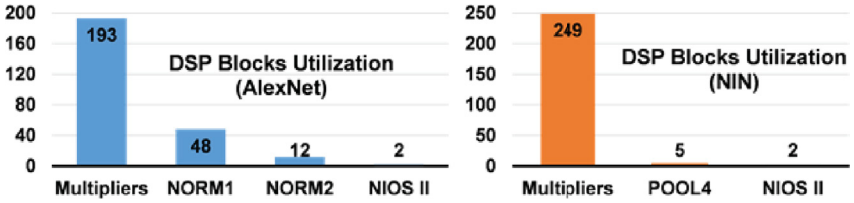

Fig. 15. DSP block breakdown in AlexNet and NiN implementations.

Table 3
Comparison with previous implementations.

	[11] AlexNet	[14,15] VGG	[29] AlexNet	[13] AlexNet	This work: AlexNet	This work: NiN
FPGA	Virtex-7 VX485T	Zynq XC7Z045	Zynq-7045	Stratix-V GXA7	Stratix-V GXA7	Stratix-V GXA7
Design Entry	C-language	RTL	RTL	OpenCL	RTL	RTL
Main Technique	Roofline model and loop transform	Data quantization and compilation tool	Library-based compiler	Design space exploration	Modularized RTL compiler and loop optimization	
Frequency	100 MHz	150 MHz	100 MHz	193.6 MHz	100 MHz	100 MHz
# operations per image	1.33 GOP	30.76 GOP	1.46 GOP	1.46 GOP	1.46 GOP	2.2 GOP
# weights	2.33 M	50.18 M	60.95 M	60.95 M	60.95 M	7.59 M
Precision	floating (32b)	fixed (16b)	fixed	fixed (8–16b)	fixed (8–16b)	fixed (8–16b)
DSP Utilization	2240 (80%)	780 (89%)	144 (16%)	256 (100%)	256 (100%)	256 (100%)
Logic Utilization[a]	186 K (61%)	183 K (84%)	37.8 K (17%)	114 K (49%)	121 K (52%)	112 K (48%)
BRAM Utilization[b]	1024 (50%)	486 (87%)	–	1893 (74%)	1552 (61%)	2330 (91%)
Convolution time	21.61 ms	163.42 ms	~ 20 ms	19.86 ms	9.92 ms	18.75 ms
FC layer time	N/A	61.18 ms		4.40 ms	2.83 ms	–
Convolution throughput	61.6 GOPS	187.80 GOPS	–	67.5 GOPS	134.1 GOPS	117.3 GOPS
Overall throughput	N/A	136.97 GOPS	~ 73 GOPS	60.2 GOPS	114.5 GOPS	117.3 GOPS

[a] Xilinx FPGAs in LUTs and Altera FPGAs in ALMs.
[b] Xilinx FPGAs in BRAMs (36 Kb) and Altera FPGAs in M20K RAMs (20 Kb).

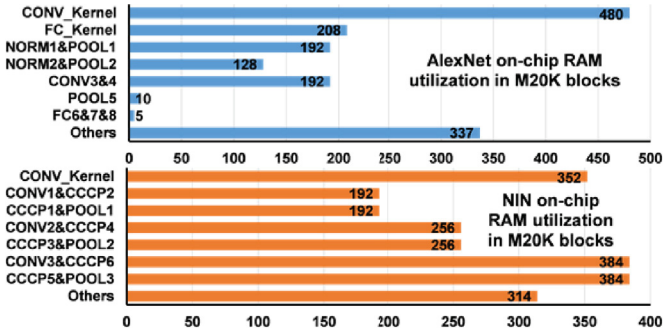

Fig. 16. On-chip RAM breakdown of AlexNet and NiN modules.

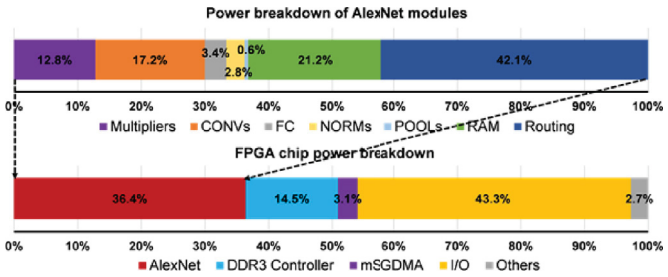

Fig. 17. Simulation based power breakdown of FPGA chip when running AlexNet.

kernel size *K* in convolution layers benefits the throughput [29]. presents DeepBurning to automatically generate FPGA-based neural network accelerators using RTL module library. Their implementations of AlexNet and NiN on Zynq-7045 SoC device show about 20 ms and 50 ms latency, respectively, which are about 1.6*X* and 2.7*X* worse than ours, respectively [28]. presents fpgaConvNet framework to automatically map neural networks onto FPGA based on HLS method. However, their framework is only demonstrated using several relatively small CNN models, e.g. LeNet-5.

7.4.3. Customized accelerator

A throughput-optimized OpenCL implementation is presented in Ref. [13] that also uses logic elements along with DSP blocks, but the number of multipliers is still much smaller than that of our RTL implementation. Our modular and scalable RTL implementation of AlexNet outperforms the OpenCL design (on the same board with similar FPGA utilization) by 1.9X for the overall throughput and the HLS design [11] by > 2X for convolution layers [12]. presents a pipelined accelerator to map different convolution layers onto different FPGA computing hardware to increase the resource utilization. However, with the increased number of CNN layers, it becomes more difficult to efficiently allocate different resources to hundreds of layers while keeping the balance between each pipeline stage.

Overall, the RTL implementation of CNN accelerator provides considerable performance benefit over high-level synthesis based implementations, which do not have good hardware efficiency. CNN RTL compiler with parameterized scalable acceleration modules also allows quick turn-around time that is comparable to high-level synthesis methodologies.

8. Conclusions

In this paper, ALAMO RTL compiler is proposed to accelerate CNNs on FPGA platforms, where the computing primitives could be easily compiled from the parametrized hardware library. Representative CNN algorithms of AlexNet and NiN have been demonstrated on an Altera Stratix-V FPGA board, which show an end-to-end throughput of 114.5 GOPS and 117.3 GOPS, resulting in 1.9X improvement compared to an optimized OpenCL design on the same FPGA board. Future work includes adopting techniques in Ref. [31,32] to increase the compiler's generality and efficiency of data and weight transfer for larger state-of-the-art CNN models [6,25,26].

Acknowledgment

This work was in part supported by the National Science Foundation within the Directorate for Engineering under Grants 1230401 and 1237856, the NSF I/UCRC Center for Embedded Systems through NSF grants 1361926 and 1432348, NSF grant 1652866, and Samsung Advanced Institute of Technology.

References

[1] Le Cun, B. Boser, J.S. Denker, D. Henderson, R.E. Howard, W. Hubbard, L.D. Jackel, Handwritten digit recognition with a back-propagation network, in: Neural Information Processing Systems (NIPS), 1990, pp. 396–404.

[2] O. Russakovsky, J. Deng, H. Su, J. Krause, et al., ImageNet large-scale visual recognition challenge, in: Int. J. Computer Vision, 2015.
[3] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, Li Fei-Fei, Large-scale video classification with convolutional neural networks, in: IEEE Conf. On Computer Vision and Pattern Recognition (CVPR), 2014, pp. 1725–1732.
[4] A. Krizhevsky, I. Sutskever, G.E. Hinton, ImageNet classification with deep convolutional neural networks, in: Neural Information Processing Systems (NIPS), 2012, pp. 1097–1105.
[5] M. Lin, Q. Chen, S. Yan, Network in network, in: Int. Conf. On Learning Representations (ICLR), 2014.
[6] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich, Going deeper with convolutions, in: IEEE Conf. On Computer Vision and Pattern Recognition (CVPR), 2015, pp. 1–9.
[7] H. Li, Z. Lin, X. Shen, J. Brandt, G. Hua, A convolutional neural network cascade for face detection, in: IEEE Conf. On Computer Vision and Pattern Recognition (CVPR), 2015, pp. 5325–5334.
[8] C. Farabet, C. Couprie, L. Najman, Y. LeCun, Learning hierarchical features for scene labeling, in: IEEE Trans.. On Pattern Analysis and Machine Intelligence, vol. 35, Aug. 2013, pp. 1915–1929.
[9] C. Farabet, B. Martini, P. Akselrod, S. Talay, Y. LeCun, E. Culurciello, Hardware accelerated convolutional neural networks for synthetic vision systems, in: IEEE Int. Symp. On Circuits and Systems (ISCAS), 2010, pp. 257–260.
[10] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, Y. LeCun, NeuFlow: a runtime reconfigurable dataflow processor for vision, in: Computer Vision and Pattern Recognition Workshops, 2011, pp. 109–116.
[11] C. Zhang, P. Li, G. Sun, J. Cong, Optimizing FPGA-based accelerator design for deep convolutional neural networks. In ACM, in: Symp. On Field-programmable Gate Arrays (FPGA), 2015, pp. 161–170.
[12] C. Zhang, D. Wu, J. Sun, G. Sun, G. Luo, J. Cong, Energy-efficient CNN implementation on a deeply pipelined FPGA cluster, in: ACM Int. Symp. On Low Power Electronics and Design (ISLPED), 2016, pp. 326–331.
[13] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-S. Seo, Y. Cao, Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks, in: Int. Symp. On Field-programmable Gate Arrays, 2016, pp. 16–25.
[14] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, Y. Wang, H. Yang, Going deeper with embedded FPGA platform for convolutional neural network, in: ACM Int. Symp. On Field-programmable Gate Arrays, 2016, pp. 26–35.
[15] K. Guo, L. Sui, J. Qiu, J. Yu, J. Wang, S. Yao, S. Han, Y. Wang, H. Yang, Angel-Eye: a complete design flow for mapping CNN onto embedded FPGA, in: IEEE Trans. Computer-aided Design of Integrated Circuits and Systems, May 2017.
[16] V. Gokhale, J. Jin, A. Dundar, B. Martini, E. Culurciello, A 240 G-ops/s mobile coprocessor for deep neural networks, in: IEEE Conf. On Computer Vision and Pattern Recognition Workshops, 2014, pp. 696–701.
[17] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, O. Temam, DaDianNao: a machine-learning supercomputer, in: *IEEE/ACM Int. Symp. on Microarchitecture*, 2014, pp. 602–622.
[18] Y.-H. Chen, T. Krishna, J.S. Emer, V. Sze, Eyeriss: an energy-efficient reconfigurable accelerator for deep convolutional neural networks, in: IEEE Int. Solid-state Circuits Conf. (ISSCC), 2016.
[19] A. Putnam, A.M. Caulfield, E.S. Chung, et al., A reconfigurable fabric for accelerating large-scale datacenter services, in: Int. Symp. On Computer Architecture (ISCA), 2014, pp. 13–24.
[20] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, T. Darrell, Caffe: Convolutional Architecture for Fast Feature Embedding, 1408, p. 5093 arXiv.
[21] R. Collobert, et al., Torch7: a matlab-like environment for machine learning, in: Big Learning Workshop, NIPS, 2011.
[22] F. Bastien, et al., Theano: new features and speed improvements, in: Deep Learning Workshop, NIPS, 2012.
[23] M. Abadi, et al., TensorFlow: Large-scale Machine Learning on Heterogeneous Systems, White paper available online at: http://download.tensorflow.org/paper/whitepaper2015.pdf.
[24] S. Gupta, A. Agrawal, K. Gopalakrishnan, P. Narayanan, Deep learning with limited numerical precision, in: Int. Conf. On Machine Learning, 2015, pp. 1735–1746.
[25] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, in: IEEE Conf. On Computer Vision and Pattern Recognition (CVPR), 2016.
[26] K. Simonyan, A. Zisserman, Very deep convolutional networks for large-scale image recognition, in: International Conference on Learning Representations (ICLR), 2015.
[27] NVIDIA Corporation, GPU-based Deep Learning Inference: a Performance and Power Analysis. White Paper.
[28] S.I. Venieris, C. Bouganis, fpgaConvNet: A framework for mapping convolutional neural networks on FPGAs, in: IEEE Int. Symp. Field-programmable Custom Computing Machines (FCCM), 2016.
[29] Y. Wang, J. Xu, Y. Han, H. Li, X. Li, DeepBurning: automatic generation of FPGA-based learning accelerators for the Neural Network family, in: Design Automation Conference (DAC), 2016.
[30] Y. Ma, N. Suda, Y. Cao, J. Seo, S. Vrudhula, Scalable and modularized RTL compilation of convolutional neural networks onto FPGA, in: IEEE Int. Conf. Field Programmable Logic and Applications (FPL), 2016.
[31] Y. Ma, Y. Cao, S. Vrudhula, J. Seo, Optimizing loop operation and dataflow in FPGA acceleration of deep convolutional neural networks, in: ACM Int. Symp. On Field-programmable Gate Arrays (FPGA), 2017.
[32] Y. Ma, Y. Cao, S. Vrudhula, J. Seo, End-to-end scalable FPGA accelerator for deep residual networks, in: IEEE Int. Symp. On Circuits and Systems (ISCAS), 2017.
[33] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, X. Li, FlexFlow: a flexible dataflow accelerator architecture for convolutional neural networks, in: IEEE Int. Symp. On High Performance Computer Architecture (HPCA), 2017.