Automatic Code Generation of Convolutional Neural Networks in FPGA Implementation

Zhiqiang Liu, Yong Dou, Jingfei Jiang and Jinwei Xu
National Laboratory for Parallel and Distributed Processing
National University of Defense Technology
Changsha, 410073, China
Email: liuzhiqiang@nudt.edu.cn

***Abstract*—Convolutional neural networks (CNNs) have gained great success in various computer vision applications. However, state-of-the-art CNN models are computation-intensive and hence are mainly processed on high performance processors like server CPUs and GPUs. Owing to the advantages of high performance, energy efficiency and reconfigurability, Field-Programmable Gate Arrays (FPGAs) have been widely explored as CNN accelerators. In this paper, we propose parallel structures to exploit the inherent parallelism and efficient computation units to perform operations in convolutional and fully-connected layers. Further, an automatic generator is proposed to generate Verilog HDL source code automatically according to high-level hardware description language. Execution time, DSP consumption and performance are analytically modeled based on some critical design variables. We demonstrate the automatic methodology by implementing two representative CNNs (LeNet and AlexNet) and evaluate the execution time models by comparing estimated and measured values. Our results show that the proposed automatic methodology yields hardware design with good performance and saves much developing round time.**

I. Introduction

In recent years, convolutional neural networks (CNNs) have gained great popularity not only in many computer vision applications including image classification[1][2], object detection[3][4] and video analysis[5][6], but also a wide range of fields such as natural language processing[7], speech recognition[8] and text classification[9]. For instance, AlexNet gained leadership with a top-5 accuracy of 84.7% in ImageNet Large-Scale Vision Recognition Challenge (ILSVRC) 2012[10], which led to significant advances in image classification accuracy compared with other traditional image classification methods. Many succeeding works on CNN applications focus on improvements of AlexNet model for improved network structure and fine-tuned network super-parameters that accommodate specific application requirements[11][12].

A trained CNN model which is employed in real applications is computationally expensive, requiring over a billion operations per input image. General processors are not efficient for CNN implementation and can hardly meet the performance requirement, thus customized accelerators based on FPGA, GPU and ASIC have attracted more and more attentions because of their good performance, high-energy efficiency and capability of reconfiguration. In particular, FPGA technology shows extraordinary progress with examples of Xilinx series 7 FPGAs and the forthcoming 3D stacking FPGAs, Altera Stratix 10 series. Some of previous works only considered small CNN models for simple tasks[13][14][15][16]. Some works implemented only convolutional layers while fully-connected layers were not examined in depth[16][17]. In the OpenCL-based FPGA accelerator proposed in work[18], 3-D convolutions were mapped as matrix multiplication operations, which brought benefit of portability but also overheads caused by padding and rearranging data. Folding technique was adopted in implementing accelerator for VGG16 with uniform 3×3 convolutional windows[19], therefore hardware resources are efficiently shared by different convolutional layers. However, this may encounter problems in implementing CNN with non-uniform convolutional windows.

High capacity and improved memory speed of FPGA chips provide larger design space than they did in the past. This paper focuses on efficiently mapping feed-forward classification phase of CNN onto reconfigurable FPGA platforms with large capacity and abundant resources. Our main contributions are the following:

- We propose parallel structures for CNN layers to exploit the inherent parallelism. Besides, efficient computation engines to carry out operations in convolutional and fully-connected layers are implemented in fixed-point arithmetic.
- We propose an automatic generator which could generate Verilog HDL source code automatically according to high-level descriptions, which include some critical variables of one CNN layer. Execution time, DSP consumption and performance are analytically modeled based on these variables.
- We demonstrate the automatic methodology by implementing two representative CNNs on Xilinx VC709 board. The maximum performance of AlexNet could achieve 222.1 GOP/s under 100 MHz, showing rather good performance.

The rest of the paper is organized as follows. Section 2 briefly introduces the basic operations of CNN. Section 3 presents the parallel structures and computation engines for convolutional and fully-connected layers. In section 4, the automatic generator is introduced and the critical design

978-1-5090-5602-6/16/$31.00 © 2016 IEEE

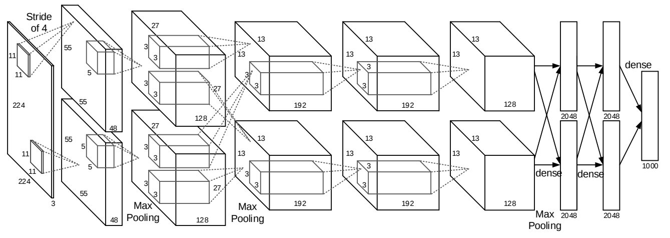

Fig. 1. Architecture of AlexNet, a typical CNN for image classification.

variables are described. In section 5, execution time, DSP consumption and performance are analytically modeled. Section 6 presents the experiments results and analysis. Finally section 7 concludes the paper.

II. CNN Basics

A typical CNN consists of multiple convolutional layers and fully-connected layers, interspersed by normalization, pooling and non-linear activation functions. Fig. 1 shows the architecture of AlexNet, a famous model used in large-scale image classification. In this section, we briefly review the operations in each layer type of CNN.

A. Convolutional Layer

A convolutional layer receives n feature maps as input and generates m output feature maps. Each input feature map is convolved by a shifting window with a shifting stride of s and window size of $k \times k$. Then the individual convolutional results are summed or aggregated. A $bias$ value is added to each pixel in the aggregated output and a suitable non-linear activation function f is used to limit the pixel value to a reasonable range. Commonly used activation functions include $tanh$, $sigmoid$ and $ReLU$. Mathematically, output feature maps in a convolutional layer is given as follows with $\otimes$ representing an image-kernel convolution:

$$Y_i = f(bias + \sum_{j=1}^{n} X_j \otimes K_{ij}) \qquad (i = 1, 2, ..., m) \tag{1}$$

Fig. 2 presents the pseudo code of a convolutional layer, where R_o represents the number of rows in output feature maps. The main computations in convolutional layers are the codes from line 5 to 10 and are actually $k \times k$ MACs (multiplications and accumulations).

B. Fully-connected Layer

In fully-connected layers, all neurons in the input and output layers are fully connected with one another. A weight matrix reflects the strength of each joint. Mathematically, an output vector in a fully-connected layer is given as follows:

$$Y_i = \sum_{j=1}^{n} X_j * W_{ij} \qquad (i = 1, 2, ..., m) \tag{2}$$

In fact, the computation in a fully-connected layer can be seen as m groups of vector inner-products, with vector length n. Fig. 3 presents the pseudo code of a fully-connected layer.

Algorithm 1 Pseudo code for a convolutional layer

```
1:  for $i$ from 1 to $m$ do                    —inter-output
2:      for $j$ from 1 to $n$ do                —intra-output
3:          for $r$ from 1 to $R_o$ do
4:              for $c$ from 1 to $R_o$ do
5:                  $tmp = 0$
6:                  for $ii$ from 1 to $k$ do
7:                      for $jj$ from 1 to $k$ do
8:                          $tmp = tmp + K[ii][jj] \times X[j][s \times (r-1) + ii][s \times (c-1) + jj]$
9:                      end for
10:                 end for
11:                 $Y[i][r][c] = Y[i][r][c] + tmp$
12:                 if $j == n$
13:                     $Y[i][r][c] = f(Y[i][r][c] + bias)$
14:                 end if
15:             end for
16:         end for
17:     end for
18: end for
```

Fig. 2. Pseudo code of a convolutional layer

Algorithm 2 Pseudo code for a fully-connected layer

```
1: for $i$ from 1 to $m$ do
2:     $tmp = 0$
3:     for $j$ from 1 to $n$ do
4:         $tmp = tmp + W[i][j] \times X[j]$
5:     end for
6:     $Y[i] = tmp$
7: end for
```

Fig. 3. Pseudo code of a fully-connected layer

C. Normalization Layer

Local response normalization (LRN) or normalization conducts a form of lateral inhibition by normalizing over local input regions, inspired by the type found in real neurons[1]. Two LRN modes exist: across neighboring features and within the same feature, which could be computed as shown in Equation (3) and (4) respectively. μ, α and s are constants with values that are determined using a validation set.

$$Y_{i,j}^{t} = X_{i,j}^{t} / \left(\mu + \alpha \sum_{l=max(0,t-\frac{s}{2})}^{min(n-1,t+\frac{s}{2})} (X_{i,j}^{l})^2 \right)^{\beta} \tag{3}$$

$$Y_{i,j}^{t} = X_{i,j}^{t} / \left(\mu + \alpha \sum_{l_i=i-\frac{s}{2}}^{i+\frac{s}{2}} \sum_{l_j=j-\frac{s}{2}}^{j+\frac{s}{2}} (X_{i+l_i,j+l_j})^2 \right)^{\beta} \tag{4}$$

D. Pooling Layer

Pooling layers in CNNs summarize the outputs of neighboring neurons in the same feature map to reduce feature dimensions. As shown in Equation (5), pooling averages neighboring elements or selects the maximum elements to produce a single element. These processes are called average-pooling and max-pooling respectively[20].

$$Y_{i,j} = \underset{0 \le l_i, l_j \le s}{max/average}(X_{i+l_i,j+l_j}) \tag{5}$$

III. Hardware Design of CNN Layers

In this section, we present the computation units and parallel structures for each layer type in CNN. Previous study[17] has proved that convolutional and fully-connected layers occupy more than 90% of the computation time. So this work focuses on accelerating convolutional and fully-connected layers. Parallelism in other layers including pooling and normalization layer is not fully exploited since it will never become the bottleneck in the feed-forward classification phase of CNN.

A. Convolutional Layer

Image-kernel convolution is the basic computation pattern in convolutional layers. During the process of an image-kernel convolution, the convolution windows scans the images from left to right and top to bottom, as line 3-4 in Fig. 2 describes. Since the main computations in each scan are $k \times k$ MACs (line 5-10), we adopt a $k \times k$ multiplier array connected with an accumulation tree as the computation unit, named as **convolver**. A convolver has the potential to generate one result of $k \times k$ MACs per clock cycle once the pipeline is established. To emulate right shift of the convolution window, we assemble a register in each multiplier to store the current pixel operand. A bypass occurs between two neighboring multipliers in the same row through which one multiplier sends the stored pixel operand to its right neighbor. We adopt k separate block RAMs to store input feature maps, which can offer k pixels per cycle. During the execution, the block RAMs send pixels to the left k multipliers while the remaining multipliers fetch the operands from their left neighbors, which is a vivid mirror of right shift of the convolution window. Besides, we assemble an inter-connect between block RAMs and convolvers, which can emulate down shift of the convolution window by adjusting the connections and address offsets. Fig. 4 presents a 3×3 convolver example. In fact, the size of convolution window may vary in different convolutional layers. Typical options include 11×11, 7×7, 5×5 and 3×3.

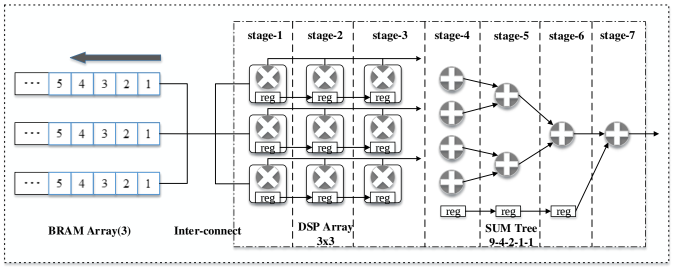

Fig. 4. A 3×3 convolver example

A convolutional layer contains $m \times n$ image-kernel convolutions (line 1-2). All the convolutions are independent and could be performed in parallel theoretically. Practical considerations such as computation resources and memory bandwidth limit the amount of parallel convolutions. We can exploit loop-level parallelism in two ways: inter-output parallelism (line 1) and intra-output parallelism (line 2)[16] according to different loop unrolling strategies. Suppose p_{inter} and p_{intra} indicate inter-output parallelism degree and intra-output parallelism degree respectively, then we have the total parallelism degree $p = p_{inter} \times p_{intra}$. In our parallel structure as shown in Fig. 5, we assemble p_{inter} separate block RAMs to store output feature maps, p_{intra} groups of block RAMs to store input feature maps with k separate block RAMs in each group and p convolvers to carry out image-kernel convolutions in parallel. In Fig. 5 $p_{inter} = 2$, $p_{intra} = 2$ and $p = 4$.

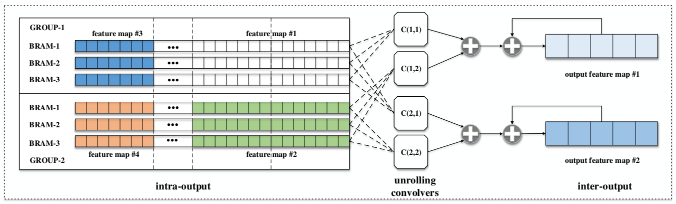

Fig. 5. Parallel structure for a convolutional layer.

B. Fully-connected Layer

Vector inner-product is the basic computation pattern in fully-connected layers. A vector inner-product contains n MACs that could be performed in parallel theoretically. However, available memory bandwidth is a constraint factor that limits the amount of parallelism we can exploit therefore the long vector will be cut into short ones. Suppose the parallelism degree in fully-connected layer is p, then we adopt a p multiplier array connected with an accumulation tree as the computation unit as shown in Fig. 6, named as **FCcore**. The input vector is stored in p separate block RAMs to offer p operand pixels per cycle.

C. Normalization Layer

There are two LRN modes in normalization layers: across neighboring features and within the same feature. In the first mode, to generate an output feature map, multiple adjacent input feature maps will be involved. Hence all the adjacent

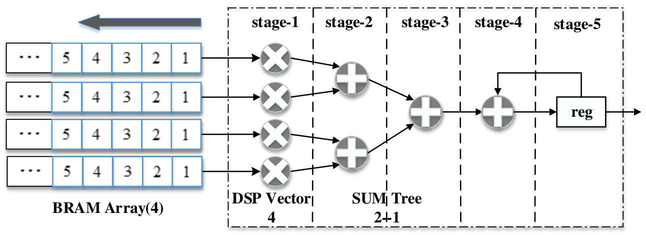

Fig. 6. Parallel structure for a fully-connected layer.

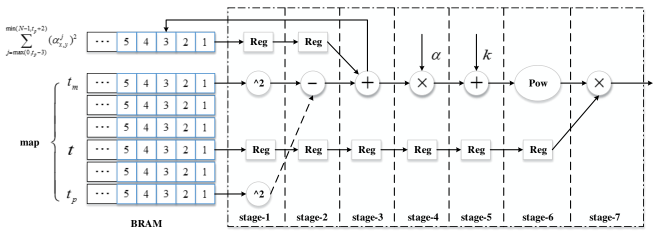

Fig. 7. The computation flow of a normalization layer.

feature maps are stored in separate block RAMs. By contrast, only the current feature map needs to be stored in the second mode. Fig. 7 shows the pipelined computation flow of a normalization layer with the first mode. The computation flows in the two modes are similar.

An exponent operation exists in normalization layers, which is expensive to precisely implement in hardware. Considering that the parameters of exponents are constants, we adopt a linear approximation function as an alternative. To ensure precision, single floating-point representation and arithmetic are adopted in implementation. As the computations in normalization layers are much less complex than convolutional and fully-connected layers, only one set of arithmetic units is assembled in a normalization layer, which consumes a total of 20 DSPs.

D. Pooling Layer

There are two modes in pooling layers: max-pooling and average-pooling. In implementation of max-pooling, the selection process is divided into two phases. In the first phase, the pixels in input features are fed into the pooling layer one-by-one in pipeline, these pixels are compared sequentially and only the maximum pixel in the adjacent columns is selected. In the second phase, the pixels are compared across rows and only the maximum pixel in the adjacent rows is selected as output. The implementation of average-pooling is similar. The comparison operations are replaced with accumulations. Since the operations in pooling layers are within the same feature map, pooling operations of different feature maps can be performed in parallel by assembling multiple logic units.

IV. Automatic Code Generation Methodology

In this section, we present the processing flow of a CNN's automatic code generation. As a first step, a high-level description of a CNN is provided from which we can extract the basic parameters of each layer. Next, precision study will be performed to determine the fixed-point data format in each layer. Then a hardware description file by means of domain-specific language we defined is provided to the automatic generator and Verilog HDL source code is finally generated.

A. High-level Description of CNN

A typical high-level description file describes a sequence of layers that form a CNN. The parameters of each layer are supplied with a label field indicating the physical meaning. Table I presents an example that describes the first convolutional, fully-connected, normalization and pooling layers respectively by means of a tuple of parameters.

TABLE I
High-level description of CNN.

```
layer {
    name: "conv1"
    type: "Convolution"
    convolution_param {
        num_output: 96
        kernel_size: 11
        stride: 4
        pad: [0 1 0 1]
    }
}
```
Convolutional Layer

```
layer {
    name : "fc6"
    type : "InnerProduct"
    inner_product_param {
        num_output : 4096
    }
}
```
Fully-connected Layer

```
layer {
    name: "norm1"
    type: "LRN"
    lrn_param {
        mode : 1
        local_size : 5
        alpha : 0.0001
        beta : 0.75
    }
}
```
Normalization Layer

```
layer {
    name: "pool1"
    type: "Pooling"
    pooling_param {
        mode : MAX
        kernel_size: 3
        stride: 2
        pad: [1 1 1 1]
    }
}
```
Pooling Layer

B. Precision Study

Previous studies have emphasized that high precision is not necessarily required in the feed-forward prediction phase because of the redundancy in the over-parameterized CNN models[21]. Thus we adopt fixed-point arithmetic units in hardware design. Data of input feature maps, output feature maps are represented in 16-bit fixed-point number while the intermediate results are represented in 18-bit fixed-point number to avoid data overflow during computation process. However, the value scopes in different layers are different therefore experiments are required to confirm the value scope of input and output feature maps in each layer. Further the length of fractional and integer part are determined. For example, the value scope of original maps is $[-255, 255]$ thus we assemble 1 sign bit, 8 integer bits and 7 fractional bits.

The kernels in convolutional layers and weights in fully-connected layers are all fractional numbers within $(-1, 1)$. Certainly we can use 16-bit fixed-point number with 15 fractional bits to represent them. However, the data size of kernels and weights in state-of-the-art deep CNN is very large, for example, this value in AlexNet is roughly 230MB in 32-bit single-precision presentation. Therefore the kernels and weights can only be stored in external memory. Lower precision representation of kernels and weights reduces memory footprint and alleviate memory bandwidth requirement. Compared to the 32-bit single-precision number, the data size of 16-bit fixed-point number decreased by 50% and that of 8-bit fixed-point number decreased by 75%. Experiments need to be conducted to explore the precision requirements of convolutional kernels and fully-connected weights. By rounding kernels and weights within specific precision and observing the degradation of the final classification accuracy, we can determine the optimal length of fractional bits.

As a case study, we conduct experiments to explore the precision requirement of two representative CNNs, LeNet and AlexNet, based on MatConvNet framework[22]. We round the kernels and weights of different precision and assign the number of integer part according to the value scope of intermediate results. We test the LeNet network using the CIFAR-10 data-set and test the AlexNet networks using the ImageNet-2012 validation data-set. Note that, all the computations in

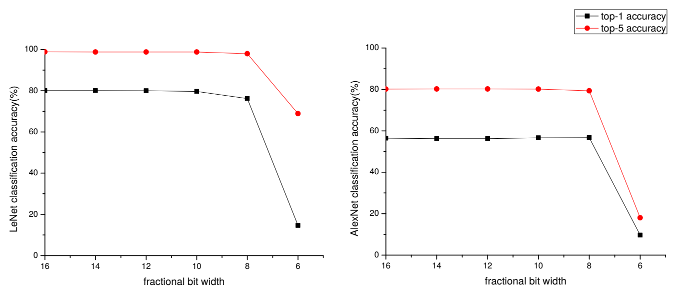

Fig. 8. Classification accuracy of LeNet and AlexNet as fractional bit width decreases.

MatConvNet are reformed with 16-bit fixed-point arithmetic to provide a vivid simulation of the hardware. The experimental results are reported in Fig. 8. For LeNet and AlexNet, there is no obvious degradation in classification accuracy until the fractional bit width drops to 6. Therefore we adopt 8-bit precision number to represent kernels and weights in FPGA implementation.

C. Hardware Description of CNN Layers

To generate the Verilog source code of each layer, we design a domain-specific language that describes the hardware implementation. The domain-specific language is actually a tuple of parameters. Some of these parameters are extracted or inferred from the high-level description file of CNN, some are fixed-point number configurations based on the precision study and the rest are specified by the developer.

- Convolutional Layers

$$< m, R_o, n, R_i, k, s, pad, p_{inter}, p_{intra}, fl_{in}, fl_{out} >$$

where

  * m is the number of output feature numbers;
  * R_o is the height of each output feature map, assuming output feature maps of dimensions $R_o \times R_o$;
  * n is the number of input feature maps;
  * R_i is the height of each input feature map, assuming input feature maps of dimensions $R_i \times R_i$;
  * k is the height of each filter in the convolutional layer, assuming kernels of dimensions $k \times k$;
  * s is the stride which defines the step between successive convolution windows;
  * pad indicates how many rows and columns of zeros will be padded into the input feature maps before image-kernel convolution.
  * p_{inter} is the inter-output parallelism degree;
  * p_{intra} is the intra-output parallelism degree;
  * fl_{in} is the length of fractional bits in input feature maps;
  * fl_{out} is the length of fractional bits in output feature maps.

- Fully-connected Layers

$$< m, n, p, fl_{in}, fl_{out} >$$

where

  * m is the length of output feature vector;
  * n is the length of input feature vector;
  * p is parallelism degree;
  * fl_{in} is the length of fractional bits in input feature vector;
  * fl_{out} is the length of fractional bits in output feature vector.

- Normalization Layers

$$< T, s, \mu, \alpha, \beta, n, R_i, fl_{out} >$$

where

  * T is the LRN mode, 0 represents LRN across neighboring features while 1 represents LRN within the same feature;
  * s is the number of feature maps used for LRN computation in the first mode and the number of neighboring neurons in two directions in the same feature map in the second mode;
  * μ, α and β are constants with values that are determined using a validation set;
  * n is the number of input feature maps as well as output feature maps;
  * R_i is the height of each input feature map as well as each output feature map, assuming feature maps of dimensions $R_i \times R_i$;
  * fl_{out} is the length of fractional bits in output feature maps.

- Pooling Layers

$$< T, k, s, pad, fl_{in}, fl_{out} >$$

where

  * T is the pooling mode, 0 represents max-pooling while 1 represents average-pooling;
  * k is the pooling size which defines a $k \times k$ pooling window;
  * s is the stride which defines the step between successive pooling windows;
  * pad indicates how many rows and columns of zeros will be padded into the input feature maps before pooling operation;
  * fl_{in} is the length of fractional bits in input feature maps;
  * fl_{out} is the length of fractional bits in output feature maps.

D. Automatic Code Generation

The hardware description of a CNN layer contains all the hardware implementation details. The automatic generator extracts the parameters and generates Verilog code automatically. There are two main parts in Verilog source code: control logic and instants of hardware modules including storage modules and computation modules. Next we will take convolutional layer as an example to show the process of code generation.

As introduced in Section 3, the hardware structures of convolutional layers is illustrated in Fig. 5. The automatic code generator firstly instantiates storage modules and computation modules according to the parameters above. There will be p_{intra} groups of block RAMs, and k separate block RAMs in each group to store input feature maps. Suppose the height of input feature maps after padding is R, then we have $R = R_i + pad(0) + pad(1)$. Thus the depth of each block RAM is given by the following equation:

$$depth = \lceil \frac{n}{p_{intra}} \rceil \times R \times \lceil \frac{R}{k} \rceil \tag{6}$$

Meanwhile, there will be p_{inter} separate block RAMs to store intermediate output feature maps and the depth of each block RAM is $R_o \times R_o$. Besides, there will be $p_{inter} \times p_{intra}$ convolvers with dimensions $k \times k$ to perform computations. fl_{in} and fl_{out} will input as parameters of computation units.

In terms of control logic, the FSM (Finite State Machine) in convolutional layers is illustrated in Fig. 9. The FSM starts at state *IDLE* and jumps to state *READ_MAP* triggered by $start$ signal. After all the pixels in input feature maps are read and stored, the image-kernel convolutions are then performed. There will be two level of loops, inter-loop and intra-loop, as the code (line 1-2) in Fig. 2 shows. During the process of image-kernel convolutions, the FSM moves in the states cycle (*READ_KERNEL*, *CONV_EXECUTE* and *JUDGE*). After all the output feature maps are generated, the FSM jumps to state *OVER* and then back to state *IDLE*. The monitor registers used for state control are listed as follows:

- $count\_map$ is used to record the number of pixels in input feature maps that have been read and stored. The critical value is given by $n \times R_i \times R_i$;
- $count\_kernel$ is used to record the number of kernels that have been read and assigned before the image-kernel convolution. Since all the $k \times k$ kernels in the same groups are read simultaneously, the critical value is given by $p_{inter} \times p_{intra}$;
- $count\_conv$ is used to record the number of generated intermediate results, thus the critical value is given by $R_o \times R_o$ which indicates the ending of a complete image-kernel convolution;
- $count\_channel$ is used to record the loop times of intra-loop. Since the intra-output degree is p_{intra}, the critical value is given by $\lceil n/p_{intra} \rceil$;
- $count\_iter$ is used to record the loop times of inter-loop. Since the inter-output degree is p_{inter}, the critical value is given by $\lceil m/p_{inter} \rceil$.

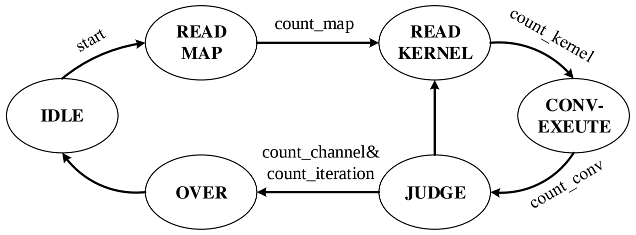

Fig. 9. FSM in convolutional layers

V. Performance Model

In this section, computation complexity, execution time and computation resource of each layer are analytically modeled based on the variables in the hardware description file.

A. Convolutional Layer

Computation complexity. In a convolutional layer, each pixel in the output feature map is the addition of n pixel which requires $k \times k$ multiply-accumulations (MACs). Considering the non-linear function attached, the amount of operations (AOs) is given by:

$$AOs = m \times n \times R_o \times R_o \times k \times k \times 2 + m \times R_o \times R_o \tag{7}$$

Execution time. As the FSM in Fig. 9 shows, the execution time includes two main parts: the time cost in reading map and image-kernel convolutions. The execution cycles of reading map is given by:

$$read\_cycles = \lceil \frac{n}{p_{intra}} \rceil \times R_i \times R_i \tag{8}$$

Considering the overheads that results from filling pipeline, padding data and reading kernel, the execution cycles of image-kernel convolutions is given by:

$$exe\_cycles = \lceil \frac{m}{p_{inter}} \rceil \times \lceil \frac{n}{p_{intra}} \rceil \times R_o \times (R_o + k + \alpha \times p) \tag{9}$$

The coefficient α is closely related to the bit width of kernels, the bit width of input port and k. Supposed the clock frequency is f, then the total execution time of a convolutional layer is given by:

$$exe\_time = (read\_cycles + exe\_cycles) \times \frac{1}{f} \tag{10}$$

Computation resource. In our implementation, we adopt 16-bit fixed-point arithmetic units thus a multiplier consumes only one DSP. For convolutional layers, the total number of DSPs consumed is $p \times k \times k$.

B. Fully-connected Layer

Computation complexity. Since the computations in a fully-connected layer are m groups of vector inner-products, with vector length n, therefore the amount of operations is given by equation (11). Some fully-connected layers are attached to a non-linear activation function. In this case, m additional operations exist.

$$AOs = m \times n \times 2 \tag{11}$$

Execution time. Since the parallelism degree is p, therefore to generate one result of a vector inner-product requires $\lceil n/p \rceil$ cycles. The total execution time of a fully-connected layer is hence given by:

$$exe\_time = \lceil \frac{n}{p} \rceil \times m \times \frac{1}{f} \tag{12}$$

Computation resource. As the parallelism degree is p, the total number of DSPs consumed is p.

C. Normalization Layer

Computation complexity. The amount of operations in a normalization layer is given by Equation (13). λ in Equation (13) is a constant factor that depends on the implementation of arithmetic units. In our implementation, $\lambda = 12$.

$$AOs = n \times R_i \times R_i \times \lambda \tag{13}$$

Execution time. The implementation of all layers are fully-pipelined. Once the pipeline is established, the output results will be generated one by one sequentially. Therefore the total execution time of a normalization layer is given by:

$$exe\_time = n \times R_i \times R_i \times \frac{1}{f} \tag{14}$$

Computation resource. As introduced in Section 3, we adopt single precision floating-point representation and arithmetic in implementation, and only one set of arithmetic units is assembled. The total number of DSPs consumed is 20.

D. Pooling Layer

Computation Complexity. The amount of operations in a pooling layer can also be computed according to Equation (13). The constant factor λ depends on the pooling method and the size of pooling window. For a 3×3 pooling window, $\lambda = 8$ when max-pooling mode is adopted while $\lambda = 9$ when average-pooling is adopted in our implementation.

The execution time of pooling layers is identical to that of normalization layers. There is no multiplication or division in max-pooling mode. Although there is an average operation in average-pooling mode, the division can be implemented by shifting operation. Therefore no DSP will be consumed in pooling layers.

VI. Experimental Results

To validate the automatic generator and the performance model, we implement two representative CNNs, LeNet and AlexNet, on VC709 board launched by Xilinx Inc. The XC7VX690T chip on VC709 board has large on-chip memory capacity and abundant computational resources, which allows to map the entire network of LeNet or AlexNet onto FPGA. Therefore in our implementation, we firstly generate Verilog source code of each layer by means of the automatic generator, then combine all the layers to form a complete accelerator. Buffers are inserted before each convolutional layer and fully-connected layer, making the accelerator pipelined across layers. To make the execution in all layers balanced as much as possible, the parallelism degrees in each layer are finely selected according to the execution time model. Table II lists all the parallelism degrees in convolutional layers and fully-connected layers of LeNet and AlexNet. For convolutional layers, x (y, z) indicates the parallelism degree and its (intra-degree, inter-degree) combination. The execution time of each layer predicted by the model are also listed in this table. We can find that the execution time are very close thus the accelerator is quite balanced to a large extent.

TABLE II
PARALLELISM DEGREES IN EACH LAYER OF LENET AND ALEXNET

		conv1	conv2	conv3	fc4/conv4	fc5/conv5	fc6	fc7	fc8
LeNet	p	$3\ (3,1)$	$10\ (5,2)$	$6\ (2,3)$	3	1	–	–	–
	time (ms)	0.369	0.358	0.366	0.219	0.006	–	–	–
AlexNet	p	$2\ (1,2)$	$16\ (8,2)$	$33\ (11,3)$	$24\ (6,4)$	$16\ (8,2)$	62	28	7
	time (ms)	4.52	6.49	6.33	6.24	6.15	6.10	6.02	5.86

TABLE III
RESOURCE UTILIZATION OF LENET AND ALEXNET ACCELERATORS.

	Slice LUT		Slice Register		Block RAMs		DSP Slices	
	used	rate	used	rate	used	rate	used	rate
LeNet	70628	16.3%	91996	10.6%	312	21.2%	484	13.4%
AlexNet	115036	26.6%	174412	20.1%	585	39.7%	1436	39.9%

The palcement and routing is completd with Vivado tool set and the hardware resource utilization is reported after that, as Table III displays. The LeNet accelerator utilizes less than 20% and the AlexNet accelerator utilizes less than 40% of total hardware resource. We can enhance resource utilization by mapping multiple PEs, which also improves the system throughput.

Table IV summarizes the computation complexity, execution time and DSP consumption of each layer in the AlexNet accelerator. By comparison we can find that the measured value of DSP slices utilized are highly consistent with model estimations while measured value of execution time are slightly larger than the model estimations. This is because the execution time model considers only the computation part, neglecting the overhead of reading input feature maps or logic control. Nevertheless, the results are already close enough and the performance model is quite reasonable.

As shown in Table V, we compare our CNN accelerators with previous work and other platforms. The performance of our accelerators for AlexNet can achieve 222.1 GOP/s. Compared to the Intel core i7-4790K CPU (4.0 GHz), our accelera-

TABLE IV
SUMMARY OF EACH LAYER IN ALEXNET

	AOs (GOP)	**exe_time (ms)**		**DSP Consumption**	
		model	**measured**	**model**	**measured**
conv1	0.21	4.52	4.63	242	242
conv2	0.45	6.49	6.54	400	400
conv3	0.30	6.33	6.47	297	297
conv4	0.22	6.24	6.37	216	216
conv5	0.15	6.15	6.36	144	144
fc6	0.076	6.10	6.10	62	62
fc7	0.034	6.02	6.23	28	28
fc8	0.008	5.86	6.02	7	7
pool1	5.6×10^{-4}	2.90	2.98	0	0
pool2	3.5×10^{-4}	1.87	2.06	0	0
pool5	7.4×10^{-5}	0.43	0.51	0	0
norm1	3.5×10^{-3}	2.90	2.98	20	20
norm2	2.2×10^{-3}	1.87	2.06	20	20
Total	1.46	–	–	1436	1436

TABLE V
COMPARISONS WITH PREVIOUS WORK AND CPU

		[17]	[18]	[19]	CPU	Ours
Year		2015	2016	2016	–	2016
Platform Board		Virtex7 VX485T	Straix-V GSD8	Zynq XC7Z045	Intel(R) Core i7-4790K	Virtex7 VX690T
Frequency (MHz)		100	120	150	4,000	100
FPGA Capacity	Slices	75,900	173,750	54,650	–	108,300
	DSP(used/total)	2240/2800	727/1963	780/900	–	1436/3600
Precision		float(32b)	fixed(16b)	fixed(16b)	float(32b)	fixed(8-16b)
CNN model		AlexNet	VGG	VGG16	AlexNet	AlexNet
Performance (GOP/s)		61.6	117.8	136.97	58.6	222.1
Power (W)		18.61	19.1	9.63	88	24.8
Resource Efficiency (GOP/s/Slice)		8.16×10^{-4}	6.78×10^{-4}	2.61×10^{-3}	–	2.05×10^{-3}
Power Efficiency (GOP/s/W)		3.31	6.16	14.22	0.67	8.96

tor for AlexNet gets an promotion of $3.79\times$ in throughput and $13.37\times$ in power efficiency. Compared to previous designs, our accelerator achieves the highest throughput with good resource efficiency. In terms of power efficiency, work[19] is 1.59 times of ours. This is mainly because we measure the power of the whole board while the accelerator consumes only 40% of total resources. By lifting utilization, for example, assembling 1 more PE, the power efficiency may match or even surpass that of work[19]. Besides, the working frequency is only 100 MHz, which still possesses promotion potential. We will enhance the resource utilization rate and lift the clock frequency to improve the accelerator performance in our future work.

VII. CONCLUSION

In this paper, we present parallel hardware structures to exploit the inherent parallelism in convolutional layers and fully-connected layers. Efficient computation units are implemented in fixed-point arithmetic. We further put forward an automatic code generator based on the hardware structures, which can generate Verilog HDL source code by means of high-level description languages. Besides, computation complexity, execution time and DSP consumption are analytically modeled. We validate the generator and model by implementing two representative CNN, LeNet and AlexNet, on Xilinx VC709 board. Our results show that the automatic methodology yields hardware design with good performance and saves much developing round time.

ACKNOWLEDGEMENTS

This work was supported in part by National Natural Science Foundation of China under Grant No. U1435219, 61303070, 61402507 and 61402499. The Corresponding author is Zhiqiang, Liu.

REFERENCES

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[2] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 1–9.

[3] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2014, pp. 580–587.

[4] A. Razavian, H. Azizpour, J. Sullivan, and S. Carlsson, "Cnn features off-the-shelf: an astounding baseline for recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2014, pp. 806–813.

[5] Z. Xu, Y. Yang, and A. G. Hauptmann, "A discriminative cnn video representation for event detection," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 1798–1807.

[6] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei, "Large-scale video classification with convolutional neural networks," in *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, 2014, pp. 1725–1732.

[7] R. Collobert and J. Weston, "A unified architecture for natural language processing: Deep neural networks with multitask learning," in *Proceedings of the 25th international conference on Machine learning*. ACM, 2008, pp. 160–167.

[8] O. Abdel-Hamid, A.-R. Mohamed, H. Jiang, L. Deng, G. Penn, and D. Yu, "Convolutional neural networks for speech recognition," *IEEE/ACM Transactions on audio, speech, and language processing*, vol. 22, no. 10, pp. 1533–1545, 2014.

[9] S. Lai, L. Xu, K. Liu, and J. Zhao, "Recurrent convolutional neural networks for text classification." in *AAAI*, 2015, pp. 2267–2273.

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

[11] J. Mao, W. Xu, Y. Yang, J. Wang, and A. L. Yuille, "Explain images with multimodal recurrent neural networks," *arXiv preprint arXiv:1410.1090*, 2014.

[12] W. Ding, R. Wang, F. Mao, and G. Taylor, "Theano-based large-scale visual recognition with multiple gpus," *arXiv preprint arXiv:1412.2302*, 2014.

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

[14] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *ACM Sigplan Notices*, vol. 49, no. 4. ACM, 2014, pp. 269–284.

[15] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun, "Cnp: An fpga-based processor for convolutional networks," in *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*. IEEE, 2009, pp. 32–37.

[16] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, "A dynamically configurable coprocessor for convolutional neural networks," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3. ACM, 2010, pp. 247–257.

[17] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2015, pp. 161–170.

[18] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-s. Seo, and Y. Cao, "Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2016, pp. 16–25.

[19] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, Y. Wang, and H. Yang, "Going deeper with embedded fpga platform for convolutional neural network," in *ACM International Symposium on FPGA*, 2016.

[20] Y.-L. Boureau, J. Ponce, and Y. LeCun, "A theoretical analysis of feature pooling in visual recognition," in *Proceedings of the 27th international conference on machine learning (ICML-10)*, 2010, pp. 111–118.

[21] M. Denil, B. Shakibi, L. Dinh, M. A. Ranzato, and N. de Freitas, "Predicting parameters in deep learning," in *Advances in Neural Information Processing Systems*, 2013, pp. 2148–2156.

[22] A. Vedaldi and K. Lenc, "Matconvnet: Convolutional neural networks for matlab," in *Proceedings of the 23rd Annual ACM Conference on Multimedia Conference*. ACM, 2015, pp. 689–692.