Throughput-Optimized FPGA Accelerator for Deep Convolutional Neural Networks

ZHIQIANG LIU, YONG DOU, JINGFEI JIANG, JINWEI XU, SHIJIE LI, YONGMEI ZHOU, and YINGNAN XU, National University of Defense Technology

Deep convolutional neural networks (CNNs) have gained great success in various computer vision applications. State-of-the-art CNN models for large-scale applications are computation intensive and memory expensive and, hence, are mainly processed on high-performance processors like server CPUs and GPUs. However, there is an increasing demand of high-accuracy or real-time object detection tasks in large-scale clusters or embedded systems, which requires energy-efficient accelerators because of the green computation requirement or the limited battery restriction. Due to the advantages of energy efficiency and reconfigurability, Field-Programmable Gate Arrays (FPGAs) have been widely explored as CNN accelerators. In this article, we present an in-depth analysis of computation complexity and the memory footprint of each CNN layer type. Then a scalable parallel framework is proposed that exploits four levels of parallelism in hardware acceleration. We further put forward a systematic design space exploration methodology to search for the optimal solution that maximizes accelerator throughput under the FPGA constraints such as on-chip memory, computational resources, external memory bandwidth, and clock frequency. Finally, we demonstrate the methodology by optimizing three representative CNNs (LeNet, AlexNet, and VGG-S) on a Xilinx VC709 board. The average performance of the three accelerators is 424.7, 445.6, and 473.4GOP/s under 100MHz working frequency, which outperforms the CPU and previous work significantly.

Categories and Subject Descriptors: C.3 [**Special-Purpose and Application-Based Systems**]: Neural Network Processing Systems

General Terms: Design, Experimentation, Performance

Additional Key Words and Phrases: FPGA architecture, convolutional neural networks, optimisation, high performance computing, application mapping

ACM Reference Format:
Zhiqiang Liu, Yong Dou, Jingfei Jiang, Jinwei Xu, Shijie Li, Yongmei Zhou, and Yingnan Xu. 2017. Throughput-optimized FPGA accelerator for deep convolutional neural networks. ACM Trans. Reconfigurable Technol. Syst. 10, 3, Article 17 (July 2017), 23 pages.
DOI: http://dx.doi.org/10.1145/3079758

1. INTRODUCTION

In recent years, deep convolutional neural networks (CNNs) have gained great popularity not only in many computer vision applications including image classification [Krizhevsky et al. 2012; Szegedy et al. 2015], object detection [Girshick et al. 2014; Razavian et al. 2014], and video analysis [Xu et al. 2015; Karpathy et al. 2014] but also a wide range of fields such as natural language processing [Collobert and Weston

This work is funded by National Science Foundation of China under Grant No. U1435219, 61303070, 61402507, and 61402499.
Authors' addresses: Z. Q. Liu, Y. Dou, J. F. Jiang, J. W. Xu, S. J. Li, Y. M. Zhou, and Y. N. Xu, National University of Defense Technology, Changsha, Hunan, China 410073; emails: {liuzhiqiang, yongdou, jingfeijiang, xujinwei13}@nudt.edu.cn, {lishijienudt, yongmei0102, yingnanpearl}@163.com.

DOI: http://dx.doi.org/10.1145/3079758

2008], speech recognition [Abdel-Hamid et al. 2014], and text classification [Lai et al. 2015]. Many succeeding works on CNNs focus on improvements of network structure and fine-tuned network super-parameters, improving the accuracy and performance significantly.

A trained CNN model 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, and thus customized accelerators based on Field-Programmable Gate Arrays (FPGA) have attracted increasing attention because of their good performance, high-energy efficiency, and capability of reconfiguration. However, some previous works only considered small CNNs for simple tasks [Chen et al. 2014; Farabet et al. 2009; Chakradhar et al. 2010]. Some works implemented only convolutional layers while fully connected layers were not examined in depth [Chakradhar et al. 2010; Zhang et al. 2015]. Complete accelerators for large-scale CNNs are implemented in Suda et al. [2016] and Qiu et al. [2016]. We will make comparisons with their work in Section 8.

Modern FPGAs posses larger capacity and higher memory speed than before, which provides larger design space than in the past. In this article, we present a systematic methodology to explore the design space and find the optimal solution that maximizes the throughput of an FPGA-based accelerator under given resources. Our main contributions are the following:

- We propose a parallel framework for FPGA-based CNN accelerators that exploits four levels of parallelism, including task level, layer level, loop level, and operator level. Basic computation units such as convolver complex in the convolutional layers are implemented in fixed-point arithmetic.
- We put forward a systematic design space exploration methodology to search for the optimal solution with maximum throughput under the FPGA constraints. Execution time, logic resources, and system throughput are analytically modeled based on some critical design variables.
- We demonstrate the methodology by implementing three representative CNNs on a Xilinx VC709 board. The average performance of LeNet, AlexNet, and VGG-S is 424.7, 445.6, and 473.4GOP/s under 100MHz, which outperforms the CPU and previous work significantly.

2. BASIC OPERATIONS AND COMPLEXITY ANALYSIS OF CNN

A typical CNN consists of multiple convolutional layers and fully connected layers, interspersed by normalization, pooling, and non-linear activation functions. In this section, we briefly review the operations of each CNN layer type. Analysis of computation complexity and memory footprint are also presented. The variables used in this section, along with their definitions, are listed as follows:

- AOs: amount of operations, a multiply-and-accumulation (MAC) is treated as two operations;
- DS: data size of weights or intermediates
- WL: word length
- n: number of input feature maps
- m: number of output feature maps
- X: an input feature
- Y: an output feature
- $K_{i,j}$: jth kernel in ith group in convolutional layers
- W: weight matrix in fully connected layers
- R_i: number of rows in an input feature map
- R_o: number of rows in an output feature map
- k: length of a square convolution window

2.1. Convolutional Layer

In a convolutional layer, each input feature map is convolved by a shifting window with a $k \times k$ kernel. 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. Mathematically, the output feature maps in a convolutional layer are computed as follows where $\otimes$ representing an image-kernel convolution:

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

Each pixel in an output feature map is the addition of n pixels that requires $k \times k$ MACs. Notice that a MAC includes two operations, multiplication and addition. Suppose the non-linear function attached is $ReLU$, which needs only one comparison operation; then the amount of operations (AOs) in a convolutional layer is given by

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

In a convolutional layer, $m \times n$ groups of kernels exist, with $k \times k$ kernels in each group. Therefore the data size of kernels is given by the following equation:

$$DS_{kernel} = m \times n \times k \times k \times WL. \tag{3}$$

The data size of intermediate results generated by a convolutional layer is given by Equation (4). This equation is also fit for normalization layers and pooling layers,

$$DS_{inters} = m \times R_o \times R_o \times WL. \tag{4}$$

2.2. 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 can be computed as follows:

$$Y_i = \sum_{j=1}^{n} X_j * W_{ij} \qquad (i = 1, 2, \ldots, m). \tag{5}$$

In fact, the computation in a fully connected layer can be seen as m groups of inner-products, with vector length of n. Thus the amount of operations can be computed as shown in Equation (6). Some fully connected layers are attached to a non-linear activation function. In this case, m additional operations exist,

$$AOs = m \times n \times 2. \tag{6}$$

Similarly, the data size of weights and intermediate results can be computed as shown in Equations (7) and (8), respectively. The data size of weights is n times the data size of intermediate results,

$$DS_{weight} = m \times n \times WL, \tag{7}$$

$$DS_{inters} = m \times WL. \tag{8}$$

2.3. 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 [Krizhevsky et al. 2012]. Two LRN modes exist: across neighboring

Table I. Summary of Each Layer Type in Different CNNs ($WL = 4B$)

	LeNet			AlexNet			VGG-S		
layer type	conv layers	fc layers	others	conv layers	fc layers	others	conv layers	fc layers	others
$AOs(GOP)$	0.028	9.5×10^{-6}	9×10^{-5}	1.33	0.12	0.007	5.08	0.19	0.019
$DS_{kernels}(MB)$	0.41	4.20	0	8.9	223.6	0	24.89	367.63	0
$DS_{inters}(MB)$	0.18	2.9×10^{-4}	0.057	3.07	0.04	2.29	7.45	0.037	0.89

features and within the same feature, which can be computed as shown in Equations (9) and (10), respectively. μ, α, β and s are constants with values that are determined using a validation set in the training phase and are fixed in the prediction phase,

$$Y_{i,j}^{t} = X_{i,j}^{t} \Bigg/ \left(\mu + \alpha \sum_{l=max(0,t-\frac{s}{2})}^{min(n-1,t+\frac{s}{2})} \left(X_{i,j}^{l}\right)^2 \right)^{\beta}, \tag{9}$$

$$Y_{i,j}^{t} = X_{i,j}^{t} \Bigg/ \left(\mu + \alpha \sum_{l_i=i-\frac{s}{2}}^{i+\frac{s}{2}} \sum_{l_j=j-\frac{s}{2}}^{j+\frac{s}{2}} \left(X_{i+l_i,j+l_j}\right)^2 \right)^{\beta}. \tag{10}$$

The amount of operations in a normalization layer is given by Equation (11). λ in Equation (11) is a constant factor that depends on the implementation of arithmetic units.

$$AOs = m \times R_o \times R_o \times \lambda. \tag{11}$$

2.4. 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 (12), pooling averages neighboring elements or selects the maximum elements to produce a single element; these processes are called average-pooling and max-pooling, respectively [Boureau et al. 2010],

$$Y_{i,j} = \underset{0 \le l_i, l_j \le s}{max/average}(X_{i+l_i,j+l_j}). \tag{12}$$

The amount of operations in a pooling layer can also be computed according to Equation (11). The constant factor λ depends on the pooling method and the size of pooling window.

3. CNN MODEL STUDY AND DESIGN DIRECTIONS

3.1. CNN Model Study

Table I lists the computation complexity and memory footprint of each layer type in three representative CNNs in 32-bit floating point implementation. All the values are computed according to the equations in Section 2. By comparison we can find that:

(1) The total data size of network parameters is very large, 230MB in AlexNet and 391MB in the VGG-S network. Among all the sub-layers, fully connected layers are the most memory-intensive that constitute over 90% in all three CNNs. The total data size of network parameters far exceeds the on-chip capacity so all parameters can only be stored in external memory. Thus optimization of memory transfer and scheduling is required to fully utilize external bandwidth. Besides, we can also reduce data access volume by using limited precision parameters. A precision study is required to determine bit width and reduce impact on the classification accuracy.

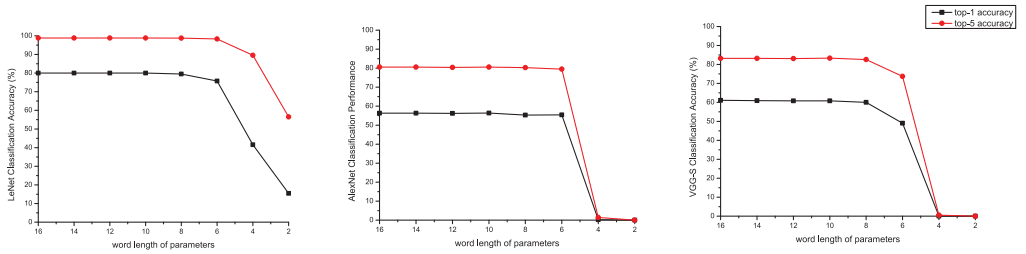

Fig. 1. Classification accuracy of three CNNs (LeNet, AlexNet, and VGG-S) as word length decreases.

(2) The operations in CNNs are computationally expensive to perform classification of a single input image, requiring roughly 1.46GOPs in AlexNet and 5.29GOPs in VGG-S. Among all the sub-layers, convolutional layers are the most computation-intensive that constitute over 90% of total operations in all three CNNs. We have to exploit the potential parallelism in CNNs to make the execution process more efficient, especially the convolutional layers.
(3) The total data size of intermediate results is smaller compared to the parameters, 0.24MB in LeNet, 5.39MB in AlexNet, and 8.37MB in VGG-S network, which can be even smaller when data quantization technique is adopted. This enables storage of all intermediate results on chip. In this article, we focus on the method to map one to multiple entire CNN models layer by layer on a single FPGA chip with large memory capacity.

3.2. 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 [Denil et al. 2013]. Therefore we adopt fixed-point arithmetic units in hardware design. Since the kernels and weights are stored in external memory, shorter word length is preferred as it brings benefits both to memory footprint and memory bandwidth. Experiments are conducted to explore the precision requirements of convolutional kernels and fully connected weights based on a MatConvNet framework [Vedaldi and Lenc 2015]. The MatConvNet framework is reformed with fixed-point representation and arithmetic to exactly mimic the hardware behaviours. Specific settings in the experiments include the following:

- For pixels in input feature maps and output feature maps, we adopt 16-bit representation. The integer width and fraction width are decided by the value scope. For example, the value scope of original map is $(-255, 255)$, and thus we assemble 1 sign bit, 8 integer bits, and 7 fraction bits.
- For intermediate results used in accumulations, we adopt 32-bit representation with 1 sign bit, 16 integer bits, and 15 fraction bits. Once accumulation finished, the intermediate results will be truncated to 16 bits according to the data format of output feature maps. When overflow or underflow occurs, it saturates to the upper bound or the lower bound, respectively.
- The kernels in convolutional layers and weights in fully connected layers are all fractional numbers within $(-1, 1)$, and thus there are 0 integer bit in fixed-point representation. In our experiments, we round the kernels and weights to different fraction bits and observe the final classification accuracy.

The experimental results are reported in Figure 1. For all the three networks, there is no obvious degradation in classification accuracy until the word length drops to 6

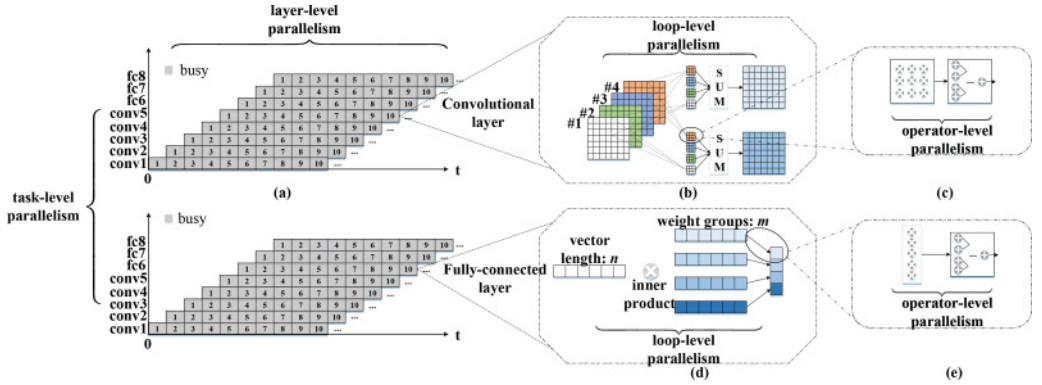

Fig. 2. Parallel framework exploiting four levels of parallelism.

or 4 bits. Therefore, we adopt 8-bit word length with 1 sign bit and 7 fraction bits to represent kernels and weights in accelerators implementation.

3.3. Parallel Framework

As analyzed, the operations in CNNs are computationally intensive. To get better throughput, we propose a parallel framework as Figure 2 shows, which can exploit the potential parallelism of CNNs in four levels.

Task-level parallelism. We can assemble multiple processing elements (PEs) to execute multiple image predication tasks simultaneously when there is abundant on-chip memory capacity. Thus task-level parallelism is explored as shown in Figure 2. To be mentioned, the kernels and weights of one trained CNN are fixed in predication phase. Therefore the kernels and weights can be shared by different tasks. All the tasks are triggered by a same signal so the parameters they need at any time are identical. The only increase in memory access brought by additional tasks lies in loading original images, which constitutes very little compared to the parameters.

Layer-level parallelism. Due to the feed-forward nature of the prediction phase, data dependencies occur between successive layers, which preclude parallel execution of all layers with a single image. However, pipeline across layers enables parallel execution of all layers with different images, thereby exploiting the layer-level parallelism. Figure 2(a) shows an ideal pipeline across layers, in which the execution time of all layers are balanced.

Loop-level parallelism. A convolutional layer contains $m \times n$ image-kernel convolutions. All the convolutions are independent and can 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: intra-output parallelism and inter-output parallelism [Chakradhar et al. 2010] according to different loop unrolling strategies as shown in Algorithm 1. The loop-level parallelism degree is the product of intra-degree and inter-degree. Once loop-level parallelism degree is specified, we have to evaluate all (intra-degree, inter-degree) combinations in terms of execution time and on-chip memory capacity and select the optimal combination. For example, suppose the parallelism degree is 4; then we have to evaluate all the (intra-degree, inter-degree) combinations including (1, 4), (2, 2), and (4, 1) and select the optimal one.

A total of m vector inner-products are found in a fully connected layer, which can be performed in parallel. We can exploit loop-level parallelism by performing multiple inner-products simultaneously as illustrated in Figure 2(d).

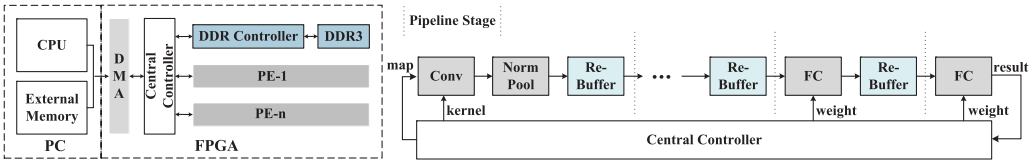

Fig. 3. Overall architecture of our accelerator (left) and PE architecture (right).

ALGORITHM 1: Pseudo Code for Convolutional Layer

```
for i from 1 to m            — inter-loop
do
    Y_i = 0;
    for j from 1 to n        — intra-loop
    do
        Y_tmp =image-kernel-convolution(X_j, K_i,j);
        Y_i = Y_i + Y_tmp;
    end
end
```

Operator-level parallelism. Image-kernel convolution is the basic computation pattern in convolutional layers. During the process of an image-kernel convolution, a $k \times k$ window scans the image rightwards and downwards. In each scan, a sub-convolution is performed with $k \times k$ MACs. We reform a classical convolver design [Bosi et al. 1999] to exploit the inherent parallelism in a sub-convolution, as shown in Figure 2(c). The details of convolver will be presented in Section 4.

Vector inner-product is the basic computation pattern in fully connected layers. A vector inner-product contains n MACs that can be performed in parallel theoretically. However, available memory bandwidth is a constraint that limits the amount of parallelism we can exploit. We design an inner-product complex that contains several multipliers and an accumulation tree to execute inner-products in pipeline, as shown in Figure 2(e).

4. SYSTEM DESIGN

In this section, we briefly introduce the system architecture of our accelerator. Then the PE architecture and the designs of basic building blocks are presented.

4.1. Overall Architecture

The overall architecture of our accelerator is shown in Figure 3 (left). It consists of two main parts: a desktop CPU that functions as host and an accelerator FPGA board that functions as development platform. The FPGA accelerator board is integrated into the PCIe slot of the desktop CPU. The logic on the FPGA chip can be divided into two parts according to their functions: One is memory system, which is responsible for data transfer between on-chip memory and off-chip DDR3 memory, and the other one is computation complex, which consists of one or multiple PEs. A PE can execute an image prediction task independently thus by assembling multiple PEs we explore the task-level parallelism. The working process of our accelerator includes the following three stages:

(a) **Data configuration.** At this stage, the CPU fetches all the required data including original images, kernels, and weights from the external memory and sends the data through DMA to DDR3 memory integrated on the FPGA board. The central

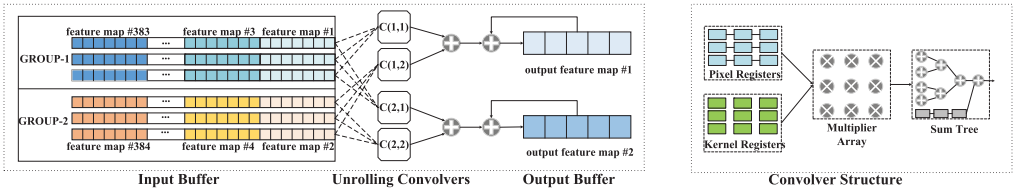

Fig. 4. The structure of a convolutional block (left) and a convolver (right).

controller monitors the data flow and triggers data processing immediately after data configuration.

(b) **Data processing.** At this stage, feed-forward prediction of one or multiple images are executed on the FPGA accelerator board. The central controller pre-fetches kernels and weights from DDR3 memory and stores them in on-chip caches, allowing the PEs to fetch required data in high speed. The results generated by PEs are stored back to the DDR3 memory.

(c) **Result recall.** After all the image prediction tasks are handled, the central controller sends a signal to the CPU, which triggers the stage of result recall. The central controller then fetches the results from DDR3 memory and sends them to the CPU through DMA.

4.2. PE Architecture

The PE architecture is shown in Figure 3 (right), composed by some basic building blocks including convolutional block, fully connected block, normalization block, pooling block and re-buffer block. The first four building blocks are closely correspondent to the CNN layer types. The re-buffer block stores all the intermediate feature maps and separates the building blocks into pipeline stages. The PE is then pipelined across layers that exploits the layer-level parallelism. Further, we will present the structure of the basic building blocks.

4.2.1. Convolutional Block. Figure 4 (left) shows the structure of a convolutional block, which consists of three parts: input buffer, convolver complex and output buffer. By configuring multiple convolvers, we exploit the loop-level parallelism. The specific configurations of the three parts are closely related to the unrolling methods. Suppose p indicates the loop-level parallelism degree, p_a indicates the intra-degree, and p_e indicates the inter-degree, and then we configure p_a groups of block RAMs in the input buffer to provide p_a input feature maps at the same time, p convolvers to perform p image-kernel convolutions in parallel, and p_e block RAMs in the output buffer to store the p_e partial results that generated simultaneously. In addition, there are k separated block RAMs in each group of the input buffer to offer k pixels every clock cycle, as the convolution window reads k new pixels every time it shifts rightwards. In the example shown in Figure 4, $p = 4$, $p_a = 2$, $p_e = 2$, and $k = 3$, thus there are two groups of block RAMs in the input buffer, four unrolling convolvers and two separated block RAMs in the output buffer.

For convolver complex, we adopt a two-dimensional (2D) convolver design as Figure 4 (right) shows. Suppose the size of the convolution window is $k \times k$, the convolver configures a $k \times k$ multiplier array, and a $(k \times k)$-item accumulation tree accordingly. In addition, there are $k \times k$ kernel registers and $k \times k$ pixel registers to store kernels and pixels, respectively. Particularly, the pixel registers are shift-registers, which means each pixel register shifts the stored data to its right neighbor every clock cycle. An inter-connect lies between the block RAMs and the pixel registers, which reconnects block RAMs and pixel registers every time the convolution window shifts downwards.

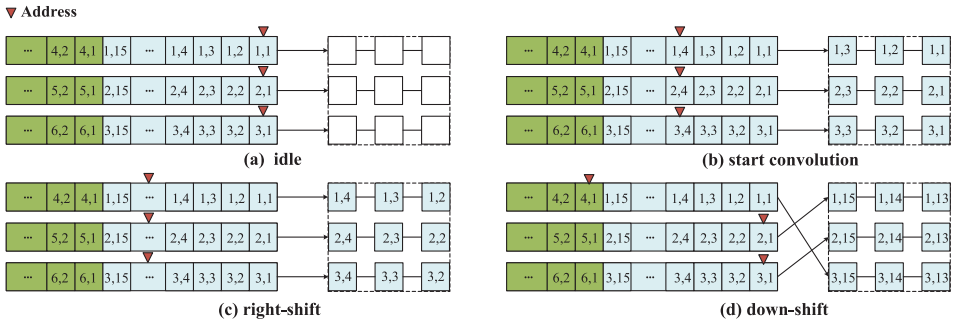

Fig. 5. The right-shift and down-shift process of the convolution window.

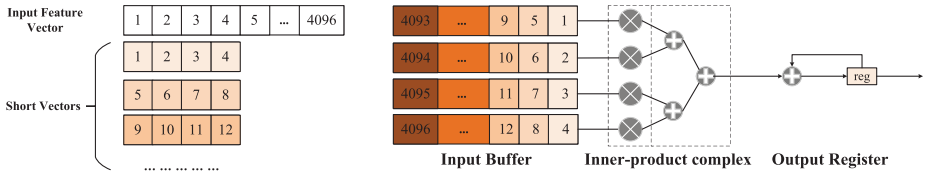

Fig. 6. The structure of a fully connected block.

Figure 5 shows an example of how right-shift and down-shift of the convolution window during the image-kernel convolution process are achieved in our design. In the example, $k = 3$ and $R_i = 15$.

(a) Before an image-kernel convolution, pixels have been stored in the block RAMs, and kernels have been fetched from cache and stored in the kernel registers. All the addresses of the block RAMs point to 0 and the pixel registers are still empty at this moment.
(b) After 3 clock cycles, the pixel registers fetched all nine pixels in the first convolution window. The convolver starts the first valid convolution. All the addresses of the block RAMs point to 4 and are ready for the next read.
(c) In the next cycle after (b), the fourth column of pixels are read from block RAMs and written to the left three pixel registers. Meantime, the other pixel registers fetch pixels from their left neighbors. All the addresses of the block RAMs point to 5 and are ready for the next read. Thus the right-shift of the convolution window is achieved.
(d) When the convolution window shifts to the right edge, it shifts downwards first and then rightwards again. The inter-connect reconnects the block RAMs and pixel registers: RAM-1 to line-3, RAM-2 to line-1, and RAM-3 to line-2. Meanwhile, the address of RAM-1 moves forward to 15 while the addresses of RAM-2 and RAM-3 move back to 0. Thus the down-shift of the convolution window is achieved.

Notice that the kernel registers hold their values during the process of image-kernel convolution. When the image-kernel convolution finish, kernels are re-fetched and the kernel registers are updated. Since we store complete input feature maps on-chip rather than a tiling of them, each kernel will be fetched only once and then aborted during one prediction task.

4.2.2. Fully Connected Block. Figure 6 shows the structure of a fully connected block, which consists of three parts: input buffer, inner-product complex, and output register. The input feature vector is cut into short vectors to perform inner-product in fully

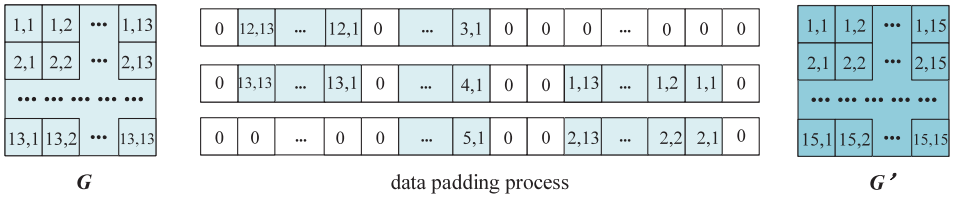

Fig. 7. The data padding process in a re-buffer block.

connected layers, as shown in Figure 6 (left). Suppose the length of each short vector is p; then we configure p separated block RAMs in the input buffer. The pixels in each short vector are distributed in different block RAMs. The inner-product complex configures p multipliers and a p-item accumulation tree to perform inner-products of short vectors. The output register accumulates every partial inner-product result and sends the result to the re-buffer block behind once the final value are obtained. No loop-level parallelism is exploited in fully connected layers limited by memory bandwidth, and hence p is defined as the total parallelism degree of a fully connected block. In the example shown in Figure 6, $p = 4$.

4.2.3. Normalization Block. 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. In addition, the computations in normalization layers are much less complex than convolutional and fully connected layers, thus only one set of arithmetic units is assembled in a normalization layer, which consumes a total of 20 DSP48 slices.

4.2.4. Pooling Block. 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 of input feature maps are fed into the pooling block 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. Changes include replacing the comparison operations with accumulations and adding an average operation behind. In addition, the average function unit is implemented by shifting registers instead of DSP48 slice, since the divisor is a constant.

4.2.5. Re-buffer Block. Re-buffer blocks are designed to reorganize data and transfer data between successive building blocks. The only module in a re-buffer block is a buffer.

(a) **Data padding.** In some convolutional layers, each input feature map is padded to form a larger map before image-kernel convolution. Figure 7 shows how data padding is achieved in a re-buffer block. In this example, $k = 3$, hence there are 3 separated block RAMs. The data of all block RAMs are reset to 0 at the beginning. The input buffer receives data from the last building block and starts to write from the second block RAM. Besides, addresses $15 * i$ and $15 * i + 14$ ($i = 0 : 4$) in each block RAM are skipped. Therefore the original feature map G with 13×13 pixels turns into a new feature map G' with 15×15 pixels after data padding.

(b) **Data transferring.** During the execution of image-kernel convolution, re-buffer blocks receive results from the output buffer of the building block ahead and store them in the buffer, as the phase-1 in Figure 8 shows. We configure the buffer of a

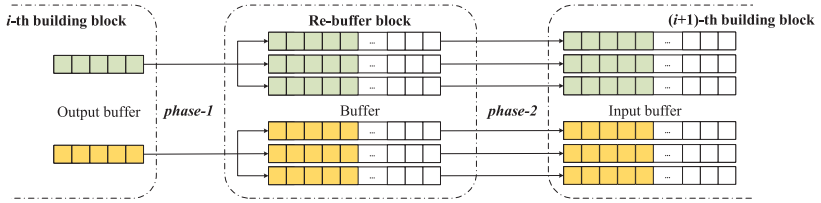

Fig. 8. Re-buffer blocks transfer data between successive building blocks: receiving and storing results from the output buffer of the ith building block in phase-1; transferring data to the input buffer of the $(i+1)$th building block in phase-2.

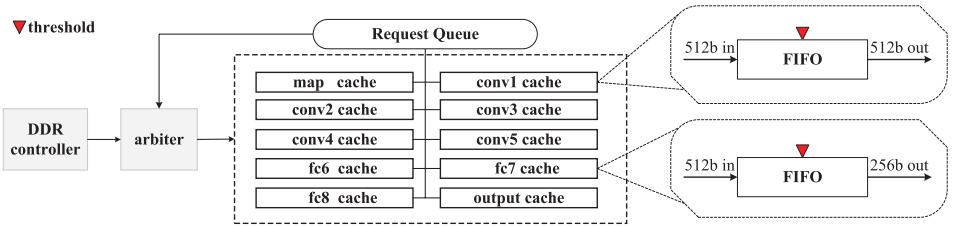

Fig. 9. The structure of memory-cache system.

re-buffer block the same as the input buffer of the next building block, hence data reorganizations are required in this phase. After all the image-kernel convolutions complete, re-buffer blocks transfer the feature maps to the building blocks behind. As the buffer in a re-buffer block is same as the input buffer of the next building block, all the block RAMs can be accessed concurrently, as the phase-2 in Figure 8 shows. The total clock cycles equal to the depth of each block RAM. Compared to the computation of image-kernel convolutions, the communication cost is very small.

5. MEMORY SYSTEM

As analyzed in Section 3, kernels and weights are extremely large in terms of data size and thus are stored in the off-chip DDR3 memory in our implementation. To effectively feed data to convolutional blocks and fully connected blocks, we adopt a memory-cache mechanism based on two considerations. First, by loading kernels and weights from the DDR3 memory to on-chip cache in advance, convolutional blocks and fully connected blocks can obtain the needed data in high speed. Second, under this mechanism, we load a batch of data rather than a single datum from the DDR3 memory every time, which helps to fully utilize memory bandwidth. In the following part, the memory-cache design will be introduced. After that, the data arrangement for convolutional kernels and fully connected weights are presented.

5.1. Cache Design

As shown in Figure 9, we configure an independent cache for each convolutional and fully connected block. In addition, there are two caches for original images and prediction results. As we store all input feature maps on chip, the convolution window scans the feature map completely during the image-kernel convolution process rather than a tiling or a part. Hence each group of kernels are loaded only once and then aborted during one image prediction task. Meanwhile, each weight is loaded and used only once. So we adopt FIFOs to build these caches in our implementation.

5.1.1. Memory Bandwidth. As analyzed in Section 3, CNN applications are memory-intensive, especially the fully connected layers. To maximize available memory

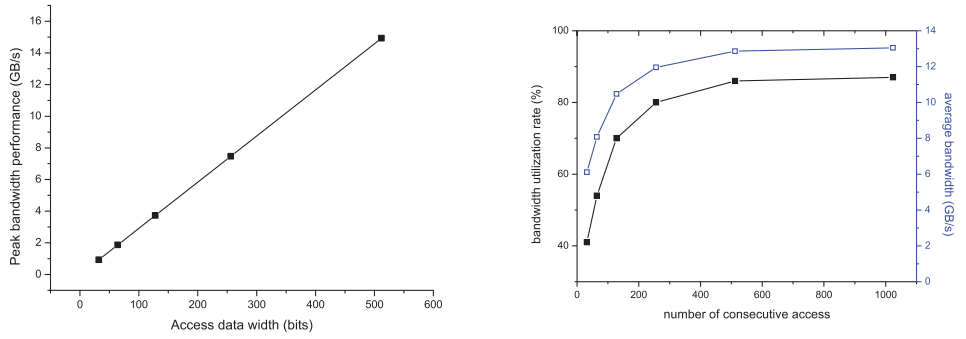

Fig. 10. The peak memory bandwidth as access bit width increases (left) and the average memory bandwidth and utilization as consecutive access number increases (right).

bandwidth, measures are taken to lift peak memory bandwidth and bandwidth utilization rate in our implementation.

(a) **Peak memory bandwidth.** The access data width of DDR3 directly determines the peak memory bandwidth. Figure 10 (left) shows the peak memory bandwidth of a DDR3 memory working under 233.33MHz. We can find from the figure that the peak memory bandwidth increases linearly as the access data width increases. The peak memory bandwidth achieves only 0.93GB/s when accessing 32-bit data, and increases to 14.9GB/s when accessing 512-bit data. To maximize memory bandwidth, the access data width is set to 512 in our implementation. Kernels and weights are finely organized to support 512-bit access mode.

(b) **Utilization rate.** Memory bandwidth utilization rate is closely related to the access mode. Figure 10 (right) shows the average bandwidth performance of a DDR3 memory working under 233.33MHz and 512-bit data width. We can find from the figure that the average bandwidth increases as the consecutive access number increases. The value is 6.11GB/s at 32, with a utilization rate of 41%, and increases to 11.96GB/s at 256, with utilization rate of 80%. In our implementation, the consecutive access number is configured to 128, and thus we load 128 consecutive 512-bits data per time step, with an average bandwidth of 10.48GB/s. Since the VC709 board we use in our implementation has two DDR3 slices, the total average bandwidth can achieve up to 20.96GB/s, which is sufficient enough to provide kernels and weights to the PEs.

5.1.2. Cache Capacity. When deciding the cache capacity, the main principle is to avoid or minimize cache misses while consuming as few hardware resources as possible. As the caches are implemented with FIFOs, a cache miss occurs when reading an empty FIFO. To avoid FIFO becoming empty, we configure a threshold for each cache. When the available data count of one cache reduces to the threshold, the cache submits a request to load data from DDR3 memory and fill it. The threshold should be large enough so the cache can receive the returned data before it turns empty. In addition, every request loads a batch of 512-bit data from DDR3 memory and the size of each batch equals the consecutive access number. Therefore, the input data width of each cache is 512 and the depth of each cache should be larger than the sum of the consecutive access number and the threshold. In our implementation, the threshold is set to 128, the same as the consecutive access number, and thus the data depth of each cache is 256 and the capacity of each cache is 16 KB.

The inputs and outputs of each FIFO are driven by independent clocks. The inputs are driven by the clock related to the DDR logic while the outputs are driven by the

Addr.	Data (512 bits)					
1	$K_{1,1}(1:9)$	$K_{1,2}(1:9)$	$K_{1,3}(1:9)$	…	$K_{1,7}(1:9)$	8 bits
2	$K_{1,8}(1:9)$	$K_{2,1}(1:9)$	$K_{2,2}(1:9)$	…	$K_{2,6}(1:9)$	8 bits
3	$K_{2,7}(1:9)$	$K_{2,8}(1:9)$	368 bits			
4	$K_{1,9}(1:9)$	$K_{1,10}(1:9)$	$K_{1,11}(1:9)$	…	$K_{1,15}(1:9)$	8 bits
5	$K_{1,16}(1:9)$	$K_{2,9}(1:9)$	$K_{2,10}(1:9)$	…	$K_{2,14}(1:9)$	8 bits
6	$K_{2,15}(1:9)$	$K_{2,16}(1:9)$	368 bits			
…	…….					

Addr.	Data (512 bits)				
1	$W_{1,1}$	$W_{1,2}$	$W_{1,3}$	…	$W_{1,64}$
2	$W_{1,65}$	$W_{1,66}$	$W_{1,67}$	…	$W_{1,128}$
…	…….				
64	$W_{1,4033}$	$W_{1,4034}$	$W_{1,4035}$	…	$W_{1,4096}$
65	$W_{2,1}$	$W_{2,2}$	$W_{2,3}$	…	$W_{2,64}$
66	$W_{2,65}$	$W_{2,66}$	$W_{2,67}$	…	$W_{2,128}$
…	…….				

Fig. 11. Arrangement of kernels (left) and arrangement of weights in external memory and caches (right).

clock related to the user logic. Hence the output data width of each cache may differ from the input data width, mainly decided by the corresponding convolutional or fully connected block. For example, in the first convolutional block of VGG-S, the size of the convolution window is 7×7, thus the conv-1 block reads $7 \times 7 \times 8b = 392b$ to load a group of kernels. In the second fully connected block of VGG-S, the parallelism degree is configured to 32 and thus the fc-7 block reads $32 \times 8b = 256b$ every clock. Accordingly, the output data width of conv-1 cache is 512 bits while the output data width of fc-7 cache is 256 bits, as shown in Figure 9.

5.1.3. Cache Management. As introduced above, a cache submits a request when the data count drops to the threshold. There may be conflicts when multiple caches submit requests simultaneously. We adopt a priority-based request queue to address this problem. The caches are assigned with different priorities, as well as the requests. Once the requests are submitted to the request queue, they are sorted and then handled in sequence according to their priorities.

In terms of priority, fully connected blocks own higher priority than convolutional blocks, as they read weights every clock cycle while convolutional blocks read kernels every span of time. Besides, the slower blocks own higher priority than the faster ones to minimize pipeline interval and maximize throughput, as the first convolutional block owns the lowest priority to avoid data overrides in the pipeline.

5.2. Data Arrangement for Convolutional Kernels

To support 512-bit data access mode, we optimize the storage pattern of convolutional kernels. As a convolutional block loads $p_e \times p_a$ groups of kernels every time, we divide the $m \times n$ groups of kernels into $p_e \times p_a$ tiles and reorganize the tiles in row-major layout. Therefore, we can load all the $p_e \times p_a$ groups of kernels continuously every time. Data combination and data padding are required to align data in 512-bit format. Figure 11 (left) shows a brief example of how we combine and pad kernels for a convolutional layer. In this example, $p = 16$, $p_e = 2$, $p_a = 8$, and $k = 3$. A group of kernels require $3 \times 3 \times 8b = 72b$ to present thus a line with 512 bits can store as more as 7 groups of kernels. Therefore, the 16 groups of kernels are divided into 3 lines. The first and second lines combine 7 groups of kernels and pad 8 invalid bits, while the third line combines 2 groups of kernels and pads 368 invalid bits to achieve 512 bits.

5.3. Data Arrangement for Fully Connected Weights

Restricted by the memory bandwidth, no loop-level parallelism is exploited in fully connected blocks. A fully connected block loads p weights in in the same row of the weight matrix every clock cycle. Hence weights are combined and stored in external memory and caches one by one in row-major layout. The parallelism degree of the fully connected block only affects the output data width of its corresponding cache, while it does not affect the data arrangement. Figure 11 (right) shows a brief example of how we combine and store weights for a fully connected layer. In the example, $m = 4{,}096$ and $n = 4{,}096$. A line with 512 bits can store exactly 64 weights, thus each row of the

weight matrix is distributed to 64 lines. Generally, n is divisible by 64, and therefore no data padding is required.

6. DESIGN SPACE EXPLORATION

Based on our design above, how to build a CNN accelerator is directly decided by a group of design variables including the number of tasks P_{task}, inter-degree p_e and intra-degree p_a of each convolutional block, and parallelism degree p of each fully connected block. Finding the best design variables that maximizes the performance of the CNN accelerator is a non-trivial task. In this section, we first model the execution time, resource utilization, memory bandwidth, and system throughput of a CNN accelerator based on the design variables. Then we present an exploration methodology to address the searching problem with these models. Finally, we introduce an example of searching for the optimal solution for an AlexNet accelerator on the VC709 board.

6.1. Performance Modeling

6.1.1. Execution Time Model. The convolver complex in convolutional blocks is fully pipelined, and thus it generates one result per cycle once the pipeline is established. The pipeline sets up in $k-1$ cycles and it restarts when the convolution window shifts downwards. Therefore to generate one map with $R_o \times R_o$ pixels requires $Ro \times (R_o + k - 1)$ cycles. The actual number of output feature maps is ceiled to multiple of p_e and the actual number of input feature maps is ceiled to multiple of p_a. In summary, the execution time of a convolutional layer is modeled as

$$exe\_time = \left\lceil \frac{m}{p_e} \right\rceil \times \left\lceil \frac{n}{p_a} \right\rceil \times R_o \times (R_o + k - 1) \times \frac{1}{f}, \tag{13}$$

where f indicates clock frequency.

The inner-product complex in fully connected blocks is also pipelined, which can perform p MACs per clock cycle once the pipeline is established. The length of the input vector is ceiled to multiple of p. Since no loop-level parallelism is exploited in fully connected blocks, the execution time of a fully connected block is modeled as

$$exe\_time = m \times \left\lceil \frac{n}{p} \right\rceil \times \frac{1}{f}. \tag{14}$$

During the working process of a pipelined PE, prediction results of an image are generated for every span of time. This time span is called pipeline interval, which directly reflects the system throughput. The pipeline interval is decided by the longest pipeline stage as follows:

$$T_p = \max_{i=1}^{l} exe\_time_i, \tag{15}$$

where l indicates the number of pipeline stages.

6.1.2. Logic Resource Model. We use the on-board DSP48 slice to implement multipliers. As pixels are represented in 16 bits while kernels and weights are represented in 8 bits, a $16b \times 8b$ multiplier consumes only one DSP48 slice. Therefore the number of DSP48 slices consumed by a convolutional block is given by $p \times k \times k$ and the number consumed by a fully connected block is given by p. The total number of DSP48 slices consumed by a single PE is modeled as

$$\#DSP = \sum_{i=1}^{l_c} p_i \times k_i \times k_i + \sum_{i=l_c+1}^{l} p_i, \tag{16}$$

where l_c indicates the number of convolutional blocks. With P_{task} PEs, the total number of DSP48 slices consumed by the accelerator should multiply a factor of P_{task}.

In terms of on-chip memory resource, convolutional blocks and re-buffer blocks consume most of block RAMs. There are p_a groups of block RAMs in the input buffer, and k block RAMs in each group. Therefore the capacity of the input buffer is modeled as

$$Memory_{input} = p_a \times \left\lceil \frac{n}{p_a} \right\rceil \times k \times \left\lceil \frac{R_i}{k} \right\rceil \times R_i \times WL. \tag{17}$$

The output buffer of convolutional blocks contains p_e block RAMs, and each block RAM stores a complete feature map, and thus the capacity of the output buffer is modeled as

$$Memory_{output} = p_e \times R_o \times R_o \times WL. \tag{18}$$

For fully connected blocks, the length of the input feature vector is ceiled to multiple of p as there are p block RAMs in the input buffer. Therefore the capacity of the input buffer of fully connected blocks is modeled as

$$Memory_{input} = p \times \left\lceil \frac{n}{p} \right\rceil \times m \times WL. \tag{19}$$

6.1.3. Memory Bandwidth Model. During the process of a complete image prediction task, all kernels and weights are loaded from the DDR3 memory to on-chip caches once. As the pipeline interval is T_p, the average memory bandwidth can be estimated by

$$BW_{avg} = \frac{DS_{total}}{T_p} = \frac{\sum_{i=1}^{l} DS_i}{T_p}, \tag{20}$$

where DS_{total} indicates the data size of all kernels and weights and DS_i indicates the data size of kernels in the ith convolutional block or the data size of weights in the ith fully connected block. BW_{avg} should be less than BW_{max}. By transformation, we obtain the constraint of pipeline interval as follows:

$$T_p \geq \frac{DS_{total}}{BW_{max}}. \tag{21}$$

6.1.4. Throughput Model. The throughput is evaluated by GOP/s, which reflects the number of operations the accelerator performs in a second. The throughput of the accelerator can be computed as follows:

$$Thr = \frac{P_{task} \times AOs_{total}}{T_p}, \tag{22}$$

where AOs_{total} indicates the total amount of operations to perform a complete image prediction task.

6.2. Exploration Methodology

The target of design space exploration is to find the best combination of the design variables (P_{task}, $p_1(p_{a1}, p_{e1})$, $p_2(p_{a2}, p_{e2}) \ldots, p_{l-1}, p_l$) that maximizes the accelerator throughput under the restrictions of on-chip memory capacity, available memory bandwidth, and computation resources. Here P_{task} indicates the number of PEs, $p_i(p_{ai}, p_{ei})$ indicates the parallelism degree; intra-degree and inter-degree of the ith convolutional block; and p_i indicates the parallelism degree of the ith fully connected block. Therefore the searching problem can be formulated by

$$\max_{(P_{task}, p_1(p_{a1}, p_{e1}), p_2(p_{a2}, p_{e2}), \ldots, p_{l-1}, p_l)} \frac{P_{task} \times AOs_{total}}{T_p} \tag{23}$$

ALGORITHM 2: Incremental Searching Algorithm

```
for P_task from 1 to P_max do
    calculate DSP_max and Tp_min
    initial: p_i = 1, i = 1, 2, ..., l
    while #DSP ≤ DSP_max and T_p ≥ Tp_min
        p_i = p_i + 1                                   //(i-th block is the bottleneck)
        if i-th block is a convolutional block
            decomposition of p_i
            find the (p_a, p_e) combination with minimum execution time and memory capacity
        end
        update #DSP and T_p
    end
    calculate throughput Thr
    if P_task = 1 or Thr ≥ Thr_tmp
        Thr_tmp = Thr
        save current solution
    end
end
```

subject to

$$\begin{cases} \sum_{i=1}^{l} Memory_i < Memory_{max} \\ \sum_{i=1}^{l} \#DSP_i \leq DSP_{max} \\ T_p \geq DS_{total}/BW_{max} \end{cases}. \tag{24}$$

Since the design variables are all integers that narrow the exploration space significantly, we propose an incremental searching algorithm to solve the searching problem. It first finds the optimal solution for every possible P_{task}. Then it compares all the solutions with different P_{task} and chooses the one with largest throughput as the final result. If two solutions are same in throughput, then the solution with larger P_{task} is preferred because of its potential gain by lifting working clock frequency. The pseudocode in Algorithm 2 depicts the incremental searching algorithm. For every possible P_{task}, we first calculate the upper bound of DSP48 slices available for a single PE by Equation (25). *uRate* is introduced to restrict the resource utilization to avoid potential problems in place and route phase when all the resources are fully exploited. In our implementation, we empirically set the utilization rate to 80%,

$$DSP_{max} = (DSP_{total} \times uRate)/P_{task}. \tag{25}$$

Meanwhile the lower bound of pipeline interval is given by

$$Tp_{min} = DS_{total}/BW_{max}. \tag{26}$$

The searching process begins with all parallelism degrees initialized to 1. While the total number of DSP48 slices consumed by a single PE is less than DSP_{max} and the pipeline interval is larger than Tp_{min}, the algorithm chooses the bottleneck block and adds 1 to its parallelism degree. Decomposition of p_i is performed to find the optimal (p_a, p_e) combination that minimizes the execution time and memory capacity in convolutional blocks. Then the number of DSP48 slices and pipeline interval are updated. This process repeats until the judgement conditions turn invalid. After that, the optimal solution for current P_{task} is obtained. The throughput of current solution is calculated and then compared to the preserved value. If the throughput is higher than the preserved one, then the current solution will be saved.

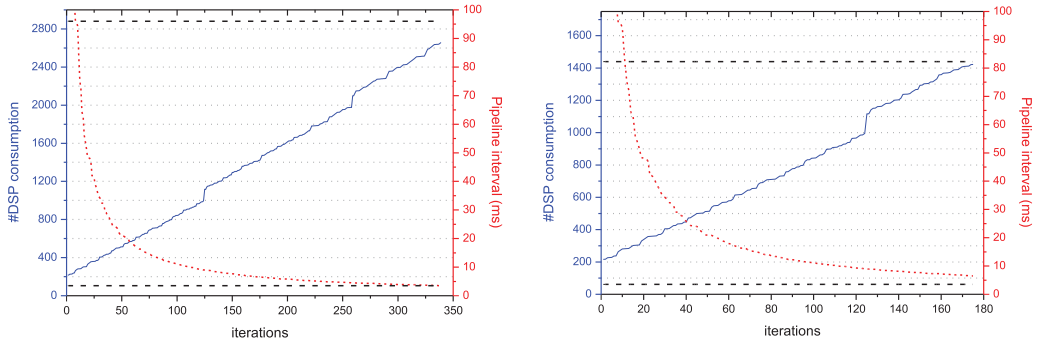

Fig. 12. The process of searching for the optimal solution to build an AlexNet accelerator on VC709 board: $P_{task} = 1$ (left); $P_{task} = 2$ (right).

6.3. A Case Study: Finding the Optimal Solution for AlexNet Accelerator on the VC709 Board

As a case study, we present the process of searching for the optimal solution for AlexNet accelerator on the VC709 board using the incremental searching algorithm. The VC709 board contains 3,600 DSP48 slices, 6.5MB on-chip capacity, and 2 DDR3 memory with a peak bandwidth of 14.9GB/s [Xilinx 2015]. As all the intermediate feature maps are stored on-chip, there can be at most 2 PEs for AlexNet according to the on-chip capacity. The access data width is set to 512 bits and the consecutive access number is configured to 128, and thus the available memory bandwidth achieves up to 20.96GB/s. The total data size of kernels and weights is roughly 63.1MB after 8-bit data quantization. Therefore, the constraints are computed as follows:

$$\begin{cases} Tp_{min} = 63.1/20.96 = 3.01\ ms \\ DSP^{1}_{max} = 3600 \times 80\% = 2880 \qquad (P_{task} = 1)\ . \\ DSP^{2}_{max} = 3600 \times 80\%/2 = 1440 \quad (P_{task} = 2) \end{cases}$$

Figure 12 shows the searching process for the optimal solution when $P_{task} = 1$ (left) and when $P_{task} = 2$ (right) under 100MHz working clock frequency. When $P_{task} = 1$, the searching process halts at the 339-th iteration limited by the memory bandwidth. The throughput of this solution can achieve 411.6GOP/s ideally, with a pipeline interval of 3.54ms. When $P_{task} = 2$, the searching process halts at the 175th iteration limited by the available DSP48 slices. The throughput of this solution can achieve 448.9GOP/s ideally, with pipeline interval of 6.49ms. As the 2-PE solution achieves higher throughput, it is then chosen as the final solution.

7. EXPERIMENTAL RESULTS

In this section, we present the validation results of the proposed architecture by implementing three representative CNN accelerators on the VC709 board. Then we verify the performance models by comparing the measured results with theoretical estimations. Finally, comparisons with other platforms (CPU and GPU) are presented.

7.1. Accelerator Implementations

We implement three CNN accelerators on the VC709 board for LeNet, AlexNet, and VGG-S, respectively. We find the optimal solution for each accelerator through the incremental searching algorithm. The design variables of each accelerator are listed in Table II. Limited by the on-chip memory capacity, we can configure at most 1 PE for

Table II. The Design Variables of the Three CNN Accelerators on the VC709 Board

	P_{task}	conv1	conv2	conv3	conv4/fc4	conv5/fc5	fc6	fc7	fc8
LeNet	6	3 (3, 1)	10 (5, 2)	6 (2, 3)	3	1	–	–	–
AlexNet	2	2 (1, 2)	16 (8, 2)	33 (11, 3)	24 (6, 4)	16 (8, 2)	64	32	8
VGG-S	1	5 (1, 5)	30 (5, 6)	40 (8, 5)	80 (5, 16)	80 (16, 5)	72	16	4

Table III. Summary of Utilization Resource

	Slice LUT		Slice Register		Block RAMs		DSP48 Slices	
LeNet	233215	54%	307617	35%	477	32.4%	2907	80.8%
AlexNet	206821	48%	323092	37%	1021	69.5%	2872	79.8%
VGG-S	223927	52%	298591	34%	1305	88.7%	2950	81.9%

Table IV. Summary of Execution Time of Each Block in Each Accelerator (ms)

		conv1	conv2	conv3	conv4/fc4	conv5/fc5	fc6	fc7	fc8
LeNet	model	0.369	0.358	0.366	0.219	0.006	–	–	–
	measured	0.386	0.367	0.363	0.228	0.007	–	–	–
AlexNet	model	4.52	6.49	6.33	6.24	6.15	6.10	6.02	5.86
	measured	4.63	6.54	6.47	6.37	6.36	6.31	6.23	6.02
VGG-S	model	7.52	10.50	10.65	10.65	10.65	10.49	10.49	10.24
	measured	7.71	10.67	10.98	11.18	11.18	10.65	10.65	10.28

VGG-S, and thus no task-level parallelism is exploited in the VGG-S accelerator. All three accelerators are implemented with fixed-point arithmetic. Kernels and weights are represented by 8-bit fixed-point numbers, and pixels in input and output feature maps are represented by 16-bit fixed-point numbers. Table III shows the hardware resource utilization summary of the three accelerators. We can see from the table that our searching algorithm helps to maximize the computation resource utilization, since the DSP48 slice utilization rates of all three accelerators have reached the *uRate*, 80%.

The execution time of each pipeline stage in the three accelerators are summarized in Table IV. We can see from the table that the pipeline stages of each accelerator are well balanced in execution time due to the updating scheme in the incremental searching algorithm: choosing the bottleneck and increasing its parallelism degree by 1. The size of the convolution window in the first convolutional layer of AlexNet and VGG-S is obviously larger than others, 11×11 and 7×7, respectively. Hence the execution time of conv1 stage decreases dramatically every time its parallelism degree increases by 1. That's why the execution time of conv1 stage is not well matched with the other pipeline stages in AlexNet and VGG-S accelerators. The pipeline interval T_p of each accelerator equals to the slowest pipeline stage, thus T_p of each accelerator is 0.386ms, 6.54ms, and 11.18ms, respectively. We can compute the throughput accordingly, 424.7GOP/s in LeNet accelerator, 445.6GOP/s in AlexNet accelerator, and 473.4GOP/s in VGG-S accelerator.

7.2. Performance Model Evaluation

Based on the design variables of the accelerator, our performance models predict the computation resource and execution time. Table V displays the predicted DSP48 slices of each building block by our models and the real values in the implementation report. By comparison we can find that the measured values are exactly the same as the model predictions. Table IV shows the predicted execution time of each pipeline stage by our models and the measured values on FPGA. By comparison, we can find that the measured values are slightly larger than the model predictions. The reason is the execution time model considers only the computation part of convolutional and fully

Table V. Summary of DSP48 Slices of Each Block in Each Accelerator

		conv1	**conv2**	**conv3**	**conv4/fc4**	**conv5/fc5**	**fc6**	**fc7**	**fc8**
LeNet	$k \times k$	5×5	5×5	5×5	1×1	1×1	–	–	–
	p	3	10	6	3	1	–	–	–
	model	75	250	150	3	1	–	–	–
	measured	75	250	150	3	1	–	–	–
AlexNet	$k \times k$	11×11	5×5	3×3	3×3	3×3	1×1	1×1	1×1
	p	2	16	33	24	16	64	32	8
	model	242	400	297	216	144	64	32	8
	measured	242	400	297	216	144	64	32	8
VGG-S	$k \times k$	7×7	5×5	3×3	3×3	3×3	1×1	1×1	1×1
	p	5	30	40	80	80	72	16	4
	model	245	750	360	720	720	72	16	4
	measured	245	750	360	720	720	72	16	4

Table VI. Performance and Power Efficiency Comparison

		accuracy		power	performance		power efficiency	
	platform	top-1	top-5	(watt)	(GOP/s)	compare	(GOP/s/W)	compare
LeNet	CPU	80.01%	98.81%	88	28.61	1x	0.33	1x
	GPU	80.01%	98.81%	225	133.87	4.68x	0.59	1.83x
	FPGA	79.64%	98.77%	25.2	424.7	14.84x	16.85	51.84x
AlexNet	CPU	56.64%	79.71%	88	64.05	1x	0.73	1x
	GPU	56.65%	79.71%	225	463.47	7.24x	2.06	2.83x
	FPGA	56.50%	79.61%	24.8	445.6	6.96x	17.97	24.69x
VGG-S	CPU	61.25%	82.37%	88	98.85	1x	1.12	1x
	GPU	61.26%	82.36%	225	834.23	8.44x	8.44	3.3x
	FPGA	61.20%	82.31%	25.6	473.4	4.79x	4.79	16.46x

connected blocks. It takes time to load input feature maps from re-buffer blocks to convolutional and fully connected blocks. Besides, although normalization and pooling blocks are pipelined, the time latency contributes to the execution time of the whole pipeline stage. Nevertheless, the model predictions are close enough to the real values and the performance models are quite reasonable.

7.3. Comparison with CPU and GPU

We conduct the prediction phase of LeNet, AlexNet, and VGG-S with MatConvNet tool running on Intel Core i7-4790K CPU (4.0GHz) and NVIDIA GTX-770 GPU (1,536 CUDA cores, 2GB GDDR5, 224.3GB/s memory bandwidth) for comparison. The dataset used in LeNet is the CIFAR10 with 10K images, while the dataset used in AlexNet and VGG-S is the ImageNet2012 validation dataset with 50K images. The batch size is configured to 100 in our experiments. Table VI summarized the top-1 and top-5 accuracies of each network on the three platforms. Compared to CPU and GPU, the FPGA implementations with fixed-point arithmetic achieve nearly the same accuracy. The accuracy degradation resulting from lower precision is less than 0.5% for top-1 accuracy and 0.1% for top-5 accuracy, which is quite acceptable in practice.

Table VI also shows the performance and power efficiency of the three networks. Compared to the CPU implementations, our accelerators for LeNet, AlexNet, and VGG-S achieve 14.84×, 6.96×, and 4.79× in performance, and 51.84×, 24.69×, and 16.46× in power efficiency. Compared to the GPU implementations, our accelerators achieve better performance in the small-scale network LeNet, comparable performance in the medium-scale network AlexNet, and worse performance in the large-scale

network VGG-S. However, our accelerators achieve higher power efficiency than the GPU implementations in all three networks.

8. RELATED WORK

Many previous works are focused on optimizing convolution engines while recent works have turned to optimize complete networks. In this section, we discuss different design methods with reference to three representatives and compare our design with them.

8.1. Design Comparison

Suda et al. [2016] implements an OpenCL-based FPGA accelerator. The 3D convolutions in convolutional layers are mapped as matrix multiplication operations by flattening and rearranging the input feature maps. For example, in the first convolutional layer of AlexNet, the three input feature maps with dimensions 224×224 were flattened and rearranged to a matrix with dimensions of $(3 \times 11 \times 11) \times (55 \times 55)$. This work provides a general solution for all CNN implementations. However, the straightforward transformation does not consider the potential optimization in convolution operations, and the overhead of memory footprint and execution time caused by flattening and rearranging input feature maps are obvious. Zhang et al. [2015] implements an HLS-based FPGA accelerator. Operator-level and loop-level parallelism in convolutional layers are exploited, and special buffers are designed for data reuse. In addition, a novel method based on roofline model is proposed to identify the optimal solution with best performance and lowest FPGA resource requirements. Qiu et al. [2016] implements a verilog-based FPGA accelerator. The design structure is similar to that of Zhang et al. [2015]. Operator-level parallelism is exploited via specially designed convolver complex, and loop-level parallelism is also well exploited. Besides, storage pattern of data in convolutional and fully connected layers are finely optimized to reduce memory access latency.

In Zhang et al. [2015] and Qiu et al. [2016], the image data are divided into tiles, and different tiles are swapped in and out from the data buffer during execution. In this article, we find that it is possible to store all the input feature maps of one single layer on chip after data quantization, so map tiling is not always necessary in CNN implementations, especially on FPGA with medium to large memory capacity. As a result of this change, kernel buffer is no longer needed as every group of kernels is loaded and used only once during one image prediction, and the memory overhead caused by swapping map tiles no longer exists. Zhang et al. [2015] and Qiu et al. [2016] implement CNN accelerators by optimizing a general convolutional block to execute different layers sequentially. However, the size of the convolution window in different convolutional layers may vary. It may cause decrease of multiplier efficiency when using general convolver complex to perform convolutions with different window size. Our work first exploits the layer-level parallelism so we can assemble customized convolver complex for different convolutional layers. Further, our work exploits the task-level parallelism by configuring multiple PEs.

Table VII shows the performance details of all the three referenced works and our AlexNet accelerator. By comparison, our design achieves the highest in performance, power efficiency, and resource efficiency. This is partly due to more DSP48 slices that our accelerator consumes. The customized convolver complex also contributes to the result.

8.2. Discussion

Our design is based on the premise that all the input feature maps of the largest layer in a CNN can be stored on chip. Hence the implementation method is closely related to

Table VII. Performance Comparison with Other FPGA Implementations

		[Zhang et al. 2015]	[Suda et al. 2016]	[Qiu et al. 2016]	ours
Year		2015	2016	2016	2016
Platform Board		Virtex7 VX485T	Straix-V GSD8	Zynq XC7Z045	Virtex7 VX690T
FPGA Capacity	Slices	75,900	173,750	54,650	108,300
	DSP	2,800	1,963	900	3,600
Frequency (MHz)		100	120	150	100
CNN model		AlexNet	VGG	VGG16	AlexNet
Precision		float(32b)	fixed(16b)	fixed(16b)	fixed(8-16b)
Power (W)		18.61	19.1	9.63	24.8
Performance (GOP/s)		61.6	117.8	136.97	445.6
Power Efficiency (GOP/s/W)		3.31	6.16	14.22	17.97
Resource Efficiency (GOP/s/slice)		8.16×10^{-4}	6.78×10^{-4}	2.61×10^{-3}	4.1×10^{-3}

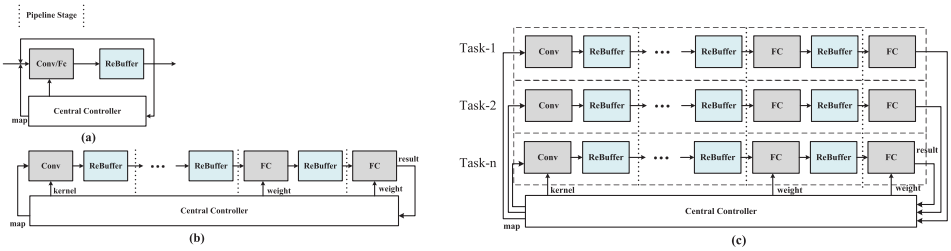

Fig. 13. Possible accelerator structures based on our design.

the CNN scale and the on-chip memory capacity of the selected FPGA chip. There are five possible cases as follows:

1) The input feature maps of the largest layer are too large to store on chip. Our design fails in this case. We have to turn to the solution in Qiu et al. [2016].
2) The on-chip memory can store only one layer. In this case, we configure one convolutional block, a fully connected block, and a re-buffer block in our accelerator, as shown in Figure 13(a). A general convolver complex is applied, and different layers are performed sequentially. No layer-level or task-level parallelism is exploited.
3) The on-chip memory can store a few layers but not all of them. In this case, we configure multiple building blocks in our accelerator, as shown in Figure 13(b). The convolutional layers with same convolution window size are folded and share the same arithmetic units. Layer-level parallelism is partly exploited, and no task-level parallelism is exploited.
4) The on-chip memory can store all layers of one CNN. In this case, we configure a building block for each layer in our accelerator. Layer-level parallelism is fully exploited, and still no task-level parallelism is exploited.
5) The on-chip memory can store all layers of multiple CNNs. In this case, we configure multiple PEs to exploit the task-level parallelism, as shown in Figure 13(c). The LeNet accelerator and AlexNet accelerator are all examples, which exploit four levels of parallelism.

We have demonstrated cases (4) and (5) in this work through the three accelerators. For cases (2) and (3), we will demonstrate the solution by implementing larger CNNs such as VGG16 and VGG19 in our future work. Optimizations including folding layers, data reuse, and data re-arrangement are required in implementations. For case (1), modern

FPGAs posses much larger capacity than before, and thus we can choose alternative FPGAs with larger memory capacity to turn case (2) into case (5).

9. CONCLUSION

In this article, we present an in-depth investigation of computation complexity and memory footprint of each CNN layer type. Then a parallel framework is proposed that exploits four levels of parallelism. We further put forward a systematic methodology to search for the optimal configuration under specific constraints of FPGA chips. We validate the framework and exploration methodology by implementing three representative CNNs on a Xilinx VC709 board. The average performance of the three accelerators for LeNet, AlexNet, and VGG-S is 424.7, 445.6, and 473.4GOP/s under 100MHz working frequency, which outperforms the CPU and previous works significantly.

REFERENCES

Ossama Abdel-Hamid, Abdel-Rahman Mohamed, Hui Jiang, Li Deng, Gerald Penn, and Dong Yu. 2014. Convolutional neural networks for speech recognition. *IEEE/ACM Trans. Aud. Speech Lang. Process.* 22, 10 (2014), 1533–1545.

Bernard Bosi, Guy Bois, and Yvon Savaria. 1999. Reconfigurable pipelined 2-D convolvers for fast digital signal processing. *IEEE Trans. VLSI Syst.* 7 (1999), 299–308.

Y.-Lan Boureau, Jean Ponce, and Yann LeCun. 2010. A theoretical analysis of feature pooling in visual recognition. In *Proceedings of the 27th International Conference on Machine Learning (ICML'10)*, Vol. 7. 111–118.

Srimat Chakradhar, Murugan Sankaradas, Venkata Jakkula, and Srihari Cadambi. 2010. A dynamically configurable coprocessor for convolutional neural networks. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*. ACM, 247–257.

Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. 2014. Dadiannao: A machine-learning supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, Vol. 49. IEEE Computer Society, 609–622.

Ronan Collobert and Jason Weston. 2008. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th International Conference on Machine Learning*. ACM, 160–167.

Misha Denil, Babak Shakibi, Laurent Dinh, Marc Aurelio Ranzato, and Nando de Freitas. 2013. Predicting parameters in deep learning. In *Advances in Neural Information Processing Systems*, Vol. 7. 2148–2156.

Clément Farabet, Cyril Poulet, Jefferson Y. Han, and Yann LeCun. 2009. Cnp: An fpga-based processor for convolutional networks. In *Proceedings of the International Conference on Field Programmable Logic and Applications, 2009 (FPL'09)*, Vol. 49. IEEE, 32–37.

Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. 2014. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 580–587.

Andrej Karpathy, George Toderici, Sanketh Shetty, Thomas Leung, Rahul Sukthankar, and Li Fei-Fei. 2014. Large-scale video classification with convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 1725–1732.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*. 1097–1105.

Siwei Lai, Liheng Xu, Kang Liu, and Jun Zhao. 2015. Recurrent convolutional neural networks for text classification. In *Proceedings of Twenty-Ninth AAAI Conference on Artificial Intelligence*. 2267–2273.

Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, Yu Wang, and Huazhong Yang. 2016. Going deeper with embedded fpga platform for convolutional neural network. In *Proceedings of the ACM International Symposium on Field-Programmable Gate Arrays*. ACM, 16–25.

Ali Razavian, Hossein Azizpour, Josephine Sullivan, and Stefan Carlsson. 2014. CNN features off-the-shelf: An astounding baseline for recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*. 806–813.

Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-sun Seo, and Yu Cao. 2016. 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, 16–25.

Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 1–9.

Andrea Vedaldi and Karel Lenc. 2015. MatConvNet: Convolutional neural networks for matlab. In *Proceedings of the 23rd Annual ACM Conference on Multimedia Conference*, Vol. 7. ACM, 689–692.

Xilinx. 2015. Virtex7-product-table.pdf. https://www.xilinx.com/support/documentation/selection-guides.

Zhongwen Xu, Yi Yang, and Alex G. Hauptmann. 2015. A discriminative CNN video representation for event detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 1798–1807.

Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. 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, 161–170.

Received June 2016; revised November 2016; accepted March 2017