MALOC: A Fully Pipelined FPGA Accelerator for Convolutional Neural Networks With All Layers Mapped on Chip

Lei Gong, *Student Member, IEEE*, Chao Wang, *Senior Member, IEEE*, Xi Li, Huaping Chen, and Xuehai Zhou

***Abstract*—Recently, field-programmable gate arrays (FPGAs) have been widely used in the implementations of hardware accelerator for convolutional neural networks (CNNs). However, most of these existing accelerators are designed in the same idea as their ASIC counterparts, in which all operations from different layers are mapped to the same hardware units and working in a multiplexed way. This manner does not take full advantage of reconfigurability and customizability of FPGAs, resulting in a certain degree of computational efficiency degradation. In this paper, we propose a new architecture for FPGA-based CNN accelerator that maps all the layers to their own on-chip units and working concurrently as a pipeline. A comprehensive mapping and optimizing methodology based on establishing roofline model oriented optimization model is proposed, which can achieve maximum resource utilization as well as optimal computational efficiency. Besides, to ease the programming burden, we propose a design framework which can provide a one-stop function for developers to generate the accelerator with our optimizing methodology. We evaluate our proposal by implementing different modern CNN models on Xilinx Zynq-7020 and Virtex-7 690t FPGA platforms. Experimental results show that our implementations can achieve a peak performance of 910.2 GOPS on Virtex-7 690t, and 36.36 GOP/s/W energy efficiency on Zynq-7020, which are superior to the previous approaches.**

***Index Terms*—Convolutional neural network (CNN), design space exploration (DSE), field-programmable gate array (FPGA)-based accelerator, pipeline, programming framework, redundancy elimination.**

Manuscript received April 3, 2018; revised June 8, 2018; accepted July 2, 2018. Date of publication July 18, 2018; date of current version October 18, 2018. This work was supported in part by the NSFC under Grant 61772482, in part by the Anhui Provincial NSF under Grant 1608085QF12, in part by the Suzhou Research Foundation under Grant SYG201625, in part by the Youth Innovation Promotion Association CAS under Grant 2017497, in part by the Fundamental Research Funds for the Central Universities under Grant WK2150110003, and in part by the State Key Laboratory of Computer Architecture, Institute of Computing Technology, CAS under Grant CARCH201709. This paper was presented in the International Conference on Hardware/Software Codesign and System Synthesis 2018, and appears as part of the ESWEEK-TCAD special issue. *(Corresponding author: Chao Wang.)*

L. Gong, C. Wang, X. Li, and X. Zhou are with the School of Computer Science and Technology, University of Science and Technology of China, Hefei 230027, China (e-mail: leigong0203@mail.ustc.edu.cn; cswang@ustc.edu.cn; llxx@ustc.edu.cn; xhzhou@ustc.edu.cn).

H. Chen is with the School of Software Engineering, University of Science and Technology of China, Hefei 230027, China (e-mail: hpchen@ustc.edu.cn).

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

Digital Object Identifier 10.1109/TCAD.2018.2857078

I. Introduction

With the increasing growth of the data amount, as well as the real-time and low-power requirements, the attention of implementing convolutional neural network (CNN) by using various hardware accelerators to improve the processing performance and efficiency has been gradually increased, especially for the inference process [1]–[5]. In particular, field-programmable gate array (FPGA), by virtue of reconfigurability and customizability, is an ideal acceleration platform for those CNN applications, and has attracted widespread attention from both industry and academia [6]–[18].

However, almost all of existing FPGA accelerators are designed in the same idea as their ASIC counterparts, that is, for achieving more generality, all operations from different network layers are mapped to the same hardware units and working in a multiplexed way. This manner results in that different layers must be implemented with the same parallelism, which is not flexible enough to take full advantage of customizability of FPGAs, and led to a series of conflicts with the inherent computing features of CNNs. First, from the perspective of coarse-grained parallelism, we take AlexNet [19] and VGG16 [20], for example, the theoretical obtainable computation to communication (CTC) ratios, which describes the number of computational operations per off-chip accessed data, of different layers varies tremendously as shown in Fig. 1. This reflects the great difference of potential data-level parallelism in different layers which is hard to compensate by using the same hardware unit with fixed computational characteristics. Second, from the aspect of fine-grained computing features, as shown in Fig. 2, the same computing unit with fixed tiling size will cause the underutilization of hardware resources since the parallelism within corresponding dimensions of some layers are less than, or cannot be divided by the tiling size. This hardware underutilization is even unacceptable on mobile and embedded devices where local processing of CNN tasks is emerging [21]. Moreover, this pattern result in that the tiling can only be implemented in the dimensions of input and output feature maps to ensure the compatibility of on-chip buffer access between different layers, and therefore a large amount of data-level parallelism within other layer dimensions cannot be explored. Third, regarding data accessing and buffering, the difference in data composition of different layers in modern CNNs is obvious as Fig. 3 shows.

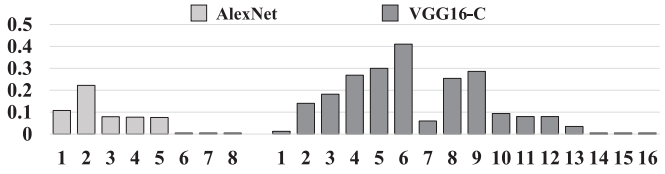


Fig. 1. CTC (GOP/MB) of each layer of modern CNNs.

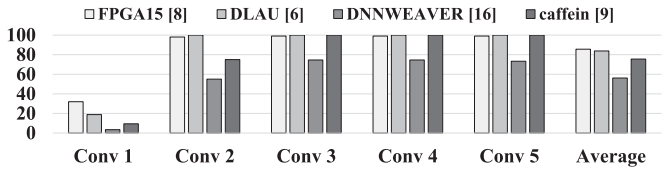


Fig. 2. Hardware utilization (%) of some previous FPGA accelerators for each CONV layer of AlexNet.

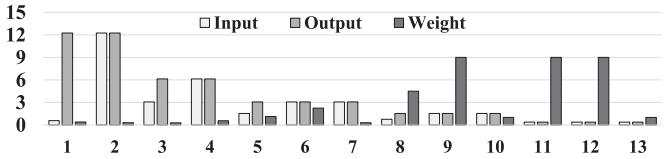


Fig. 3. Data amount (MB) of input, output, and weight for each CONV layer of VGG16-C.

While using the same on-chip buffer sets and buffering policy to treat different types of data in different layers will also lead to the decrease of on-chip buffer utilization efficiency. As a result, the existing mainstream accelerator architecture has the critical issue of computing efficiency.

Inspired by the idea of heterogeneous multicore systems which have been widely employed in deep learning domain [22], [23], considering that the amount of on-chip resources of existing FPGAs are big enough even on some embedded platforms, and this amount is still growing rapidly on the one hand, and on the other hand, with the evolution of deep learning algorithms, small-scale networks and new techniques such as SqueezeNet [24] and BNN [25] are constantly emerging, it is possible to map all the network layers to their own heterogeneous unit within one chip and optimizing independently, which is more conforming to the CNN nature and can achieve a higher computational efficiency. Previous study has tried this idea in which all convolutional (CONV) layers of AlexNet were implemented on-chip [26], but there is still a lack of completed design and optimizing method for this approach. Therefore, in this paper, we propose a new architecture with the systematic design and optimizing methodology for FPGA-based CNN accelerator that solidifies all network layers on-chip. In summary, the main contributions of this paper are as follows.

1) We present a novel CNN accelerator architecture which implements all the network layers on-chip, and the computation from different layers will be mapped to their own hardware unit with independent optimization.
2) For a given CNN model and FPGA platform, we propose a comprehensive mapping and optimizing methodology toward the proposed accelerator architecture based on establishing roofline model oriented optimization model, which can achieve maximum resource utilization as well as the optimal computational efficiency.
3) To gain more performance portability, we design a programming framework which integrates our proposed accelerator design and optimization methodology between top-level modern deep learning framework and bottom-level hardware integrated development environment (IDE), which can simplify the design complexity significantly.
4) As a case study, we implement three widely used CNN models, LeNet-5, AlexNet, and VGG16, on modern commercial FPGA platforms. It can achieve a peak performance of 910.2 GOP/s and the power efficiency with a value 36.36 GOP/s/W which are better than the previous designs.

The rest of this paper is organized as follows. Section II introduces the background of CNNs and roofline model. Section III presents the architecture and optimizing methodology of the proposed accelerator. Section IV describes the programming framework. Section V shows the experimental results. Section VI discusses related studies, and Section VII concludes this paper.

II. Background

A. Overview of the CNN Algorithm

A typical CNN is composed of three types of layers, i.e., CONV layer, pooling (POOL) layer, and fully connected (FC) layer. Traditionally, the CONV layers and the POOL layers often appear alternately at the front end of the network, and at the back end, it usually ends with several successive FC layers. The inference starts with receiving input from the first layer, and the calculation is done one by one in accordance with the layer order.

CONV layer is the chief component of CNN model, and it accounts for more than 90% of the total network computation. The main function of CONV layer is to extract feature information from N input feature maps generated from the previous layers, and outputs M output feature maps by using a set of M $K_1 \times K_2$ sized filter kernels, with convolution stride size S. The dimension of each filter kernel is same as the number of input feature maps. Besides, each element within output feature maps will be added with a bias value, and then be transformed through a nonlinear activation function such as *ReLU* before entering the next layer. The computation pattern can be described by the following expression:

$$\begin{aligned}&\text{Output}[m][r][c]\\&= f\left(\sum_{n=0}^{N}\sum_{i=0}^{K_1}\sum_{j=0}^{K_2}\text{weight}[m][n][i]\big[j\big]\right.\\&\qquad\left.\times\ \text{Input}[n][S\times r+i]\big[S\times c+j\big]+\text{bias}[m]\right).\end{aligned}\quad(1)$$

POOL layer is usually attached to the CONV layer, and is applied to achieve translation invariance and avoid computing redundancy by down-sampling the adjacent pixels in space

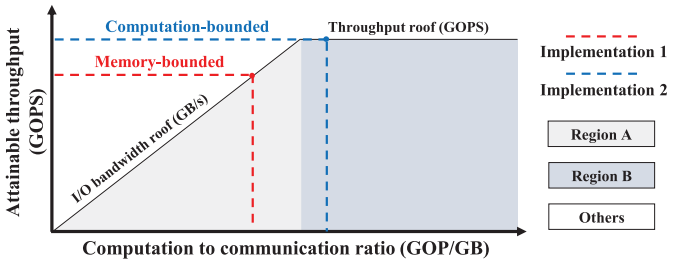


Fig. 4. Description of the Roofline model.

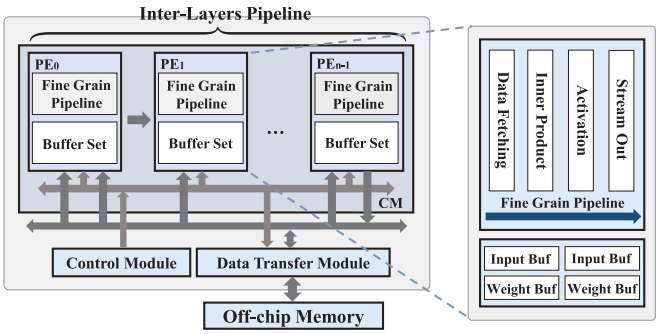


Fig. 5. Overall architecture of our proposed accelerator.

from the input feature maps. Therefore, after the processing of the POOL layer, the dimension of the output feature maps will be decreased, according to the size and the stride distance of the POOL window.

FC layer usually appears at the end of the network and is used to make the final classification. Since the neurons in this layer are connected with all the neurons in the adjacent layers, therefore, it has a huge amount of weight data and contributes to more than 90% of total data accesses. During each iteration, the FC layer will first translate the input feature maps into vectors, and then inner-product them with weights in a matrix-multiplication manner. After that, the generated result will also be added with a bias value and translated through nonlinear activation function before output. Its computation pattern can be described by the following expression:

$$\text{Output}[m] = f\left(\sum_{n=0}^{N} \text{weight}[m][n] \times \text{Input}[n] + \text{bias}[m]\right). \quad (2)$$

B. Roofline Model

For a given computing system, the performance bottleneck can occur in either on-chip computation or off-chip memory access. By visualizing the relationship between computation and memory behaviors, roofline model [27] can effectively identify the system bottleneck in accelerator design. Due to the computation of multilayers are carried out at the same time, the computation and memory behavior become more complex, therefore, in our accelerator design, it can be leveraged to solve the optimization problem related to the given CNN model and hardware platform.

In roofline model, as shown in Fig. 4, the X-coordinate represents the CTC ratio, and the Y-coordinate represents the attainable throughput. Accordingly, any coordinate point can be regarded as an accelerator implementation scheme, and its Y-coordinate and slope value represent the related throughput and the required bandwidth, respectively. Only the coordinate points falling within computation-bound and memory-bound (regions A and B) related to the given hardware platform are reasonable implementations. Besides, the throughput of implementation 2 can exceed implementation 1 in Fig. 5 because the latter is memory-bounded, which will lead to under-utilization of computational resources. Also, points with the same throughput value in region B have higher CTC ratio than that in region A, therefore, there are more feasible because they have lower requirements for memory bandwidth.

III. Accelerator Design

In this section, we first present the overall accelerator architecture, and then introduce the design and optimizing details of each layer, which include the establishment of the optimization model for the DSE of multilayers architecture, and the redundancy elimination method of FC layers.

A. Overview

The overall architecture of our proposed accelerator is shown in Fig. 5, which mainly consists of a central control module, a data transfer module, and a computational module (CM), each of which has specific features as below.

The central control module is responsible for controlling the entire acceleration process which mainly includes the initialization and synchronization for each on-chip module and starts up various types of data transmissions between off-chip memory and on-chip computation engines by specifying the address information to the corresponding data communication controllers. In practice, this module can be realized through hardware description language described register transfer level implementation (i.e., finite-state machine), or employ the soft/firm core within FPGA chip that directly runs the software controlling code. Considering the compatibility of different hardware platforms, we adopt the latter pattern, on the premise of the slightly increased performance and energy overhead.

The data transfer module is responsible for all data transmission process. It consists of an off-chip main memory controller and several on-chip DMA controllers. The main memory controller is the data communication interface between on-chip logics and off-chip memory, all off-chip data access will be aroused by it. Based on that, the DMA controllers act as intermediaries between the CM and the main memory controller, and are mainly used to transfer input feature maps and output feature maps to and from the first and the last PE, respectively, and the weights to all of the PEs within the CM. All DMA transmissions are end-to-end. As a result, the DMA controllers responsible for transmitting input/output feature maps and weights adopt dual-channel and single-channel, respectively, and the total number of DMA controllers is equal to the number of the PEs plus one.

The CM consists of multiple submodules called processing elements (PEs), and is responsible for the computation. The exact number of the PEs is determined by the number of layers

in the targeted CNN model. Since the computation of different layers is overlapped and pipelined in our accelerator, therefore, each PE is acted as one pipeline stage and all the PEs are chained one by one together to make up the whole interlayers pipeline. Within each PE, there is a finer grain pipeline to handle one specific layer and this pipeline can be optimized independently according to the features of the corresponding layer. All arithmetic units in the PEs adopt fixed-16 data type rather than conventional float-32. The main reason is that, with ignorable accuracy loss, the fixed-16 unit can save hardware cost significantly, especially for digital signal process (DSP) blocks.

Unlike previous designs that use shared on-chip buffer set between different layers, instead in our design each PE has a set of local buffers, and the advantage is that the structural parameters of each buffer including the buffer capacity, the number of read/write ports, and the partition policy can be specified independently according to the adopted parallelism of each layer. Besides, there is no output buffer in our accelerator since the computing results of each PE will be directly streamed into the input buffer of the subsequent PE. Therefore, rather than adopt traditional methods that buffer as much weight data as possible, instead we buffer as much the input feature maps of each layer as possible, in a ping-pong manner. For weights, we only buffer a small part of them which will be used in the current and next computation round, and also, in a ping-pong manner. The amount of buffered weights is determined by the parallelism of the PE. Comparing to the previous implementations, this buffering policy, on the one hand, does not decrease the reusability of the buffered weights, and on the other hand, it can also avoid frequent off-chip memory access for intermediate output feature maps. However, due to the data amount of the overall intermediate output feature maps in some modern large-scale CNN models is usually larger than the on-chip memory capacity of existing FPGA devices, therefore, for a given CNN model and hardware platform, we propose two paralleling strategies to adapt to two different scenarios that the on-chip memory resource can or cannot accommodate the overall intermediate output feature maps of the network, respectively. These two strategies, one is simple loop-tiling strategy, which is based on the traditional tiling technique [1], and its main goal is to find the optimal tiling mode directly for each PE, and another one is layer-fusion-based strategy which is introduced from the fused-layer technique [28], and its main purpose is to decrease the data transmission amount between successive CONV layers and on this basis, finding the optimal tiling mode for each PE. Different from traditional approaches, since each PE in our design is exclusive to one specific layer, therefore, tiling can be used more flexibly for both of two strategies. Another issue with implementing all layers on-chip is that the amount of off-chip memory access will increase as the more parallelism being exploited. Therefore, we apply a semibatch processing mode for both of the paralleling strategies, that is, for a given batch size, the CONV layers still process each input one by one as traditional mode, and each FC layer will process each output neuron in batches after receiving batch size inputs from the previous layer. This method, on the one hand, can achieve the reuse of weight data in FC layers, and on the other hand, can avoid buffering too much data on-chip since the input size of the FC layers is relatively small. Besides, in order to further decrease the off-chip memory access, we also apply network pruning in FC layers and enhance the corresponding PEs to support sparse networks, which will be described later.

Algorithm 1 Pseudocode of a Unrolled and Tiled CONV Layer

```
1: for (r = 0; r < R; r+=T_r)
2: for (c = 0; c < C; c+=T_c)
3: for (m = 0; m < M; m+=T_m)
4: for (n = 0; n < N; n+=T_n)
5: # Computation within unrolled and tiled loop
6: for (t_r = r; t_r < min(r + Tr, R); t_r++)
7: for (t_c = c; t_c < min(c + Tc, C); t_c++)
8: for (t_m = m; t_m < min(m + Tm, M); t_m++)
9: for (t_n = n; t_n < min(n + Tn, N); t_n++)
10: for (i = 0; i < K_1; i++)
11: for (j = 0; j < K_2; j++)
12: output[t_m][t_r][t_c] += weight[t_m][t_n][i][j] × input[t_n][S*t_r+
    i][S * t_c + j];
```

B. Parallelism Determine for Each Layer

For each paralleling strategy, determining the parallelism of each PE for specific CNN model and hardware platform is a DSE process, and can be solved by building an optimization model. Due to the computation of multilayers are carried out concurrently in our proposed accelerator, the on-chip computation and off-chip memory behaviors will become more complex which need to be coordinated elaborately to avoid bottlenecks. Therefore, in our design, we will determine the parallelism for each layer by building a roofline model oriented optimization model. In order to facilitate the description, in the following, we first analyze the constraint conditions corresponding to the different paralleling strategies and then introduce the overall objective functions.

1) Constraint Conditions for Small CNN Models: At this point, we consider that, for the target CNN model and hardware platform, the intermediate output of all layers can be buffered on-chip, and therefore, the simple loop-tiling strategy is considered. As we know, the CONV layer can be expressed as sixfold loop iteration, and traditionally loop unrolling and loop tiling technique can be applied to increase the execution parallelism as shown in Algorithm 1. As a special case, the FC layers can also be expressed in Algorithm 1 if the values of loop trip count of R, C, K_1, and K_2 as well as the stride size S are set to 1. For this loop iteration, unrolling along different loop dimensions and tiling with different sizes will result in different parallelism as well as hardware resource utilization. Besides, as we can see in Algorithm 1, loop tiling is not adopted in K_1 and K_2 since the trip counts of these two loops are usually relatively small. Also, the tiled computation is often pipelined with fixed pipelining interval (typically is 1 cycle), so that the iteration time of the layer will be proportional to the loop trip counts strictly. Our

optimization goal is to find the tiling scheme of each layer which on the one hand can fully exploit the data-level parallelism within each layer and balance their respective iteration time to avoid the pipeline stall between different layers, and on the other hand can coordinate the entire on-chip computation and off-chip memory access under specific hardware conditions.

For a CNN model with a total of α layers, of which β one are CONV layers ($\beta < \alpha$, and accordingly the number of FC layers is α–β), the independent variables of the optimization model, which includes the alternative tiling sizes in each loop dimensions of each layer and the batch size, are presented in (3). All variables are integers, and T_{r_i} and T_{c_i} in FC layers are fixed to 1. The corresponding constraint conditions of each variable are illustrated in (4), where σ is a constant (with a threshold 0.9) related to the value of tiling size which placed here to avoid excessive hardware idle, PARA_i denotes the overall parallelism of layer i after loop tiling, W_i, FM_i, and IDX_i denote the buffer size of the weight, input feature map, and index of layer i, respectively, B_{w_i}, B_{fm_i}, and B_{idx_i} denote the number of block random access memory (BRAM) blocks consumed for weight buffer, input feature map buffer, and index buffer of layer i, respectively, $\text{DSP}_{\text{total}}$, Bram_{cap}, and Bram_{num} denote total number of DSP and BRAM blocks, and total capacity of BRAM resource in the target FPGA platform, respectively, and ExeTime_i denotes the execution time of one single iteration of layer i after loop tiling. The impact of network pruning is taken into account in the expresses of IDX_i and B_{idx_i} in (9) and (10), which will be described in the following section. The BRAM in (10)–(12) denotes the capacity of a single BRAM block. Here, we assume that a DSP module can perform a multiplication in a cycle, and addition does not consume DSP resources, for the fixed-16 data type

$$\text{variables} = \left(T_{m_i}, T_{n_i}, T_{r_i}, T_{c_i}, \text{batch}\right), \quad i \in (0, \alpha] \tag{3}$$

$$\begin{cases} 1 \le T_{m_i} \le M_i \ \text{ and } \ 0.9 \le \sigma_{m_i}, \ \forall i \in (0, \alpha] \\ 1 \le T_{n_i} \le N_i \ \text{ and } \ 0.9 \le \sigma_{n_i}, \ \forall i \in (0, \alpha] \\ 1 \le T_{r_i} \le R_i \ \text{ and } \ 0.9 \le \sigma_{r_i}, \ \forall i \in (0, \beta] \\ 1 \le T_{c_i} \le C_i \ \text{ and } \ 0.9 \le \sigma_{c_i}, \ \forall i \in (0, \beta] \\ \sum_1^\alpha \text{PARA}_i < \text{DSP}_{\text{total}} \\ 2 * \left\{ \left(\sum_1^\alpha (W_i + \text{FM}_i) + \sum_{\beta+1}^\alpha \text{IDX}_i \right) \right\} < \text{Bram}_{\text{cap}} \\ 2 * \left\{ \sum_1^\alpha \left(B_{w_i} + B_{fm_i}\right) + \sum_{\beta+1}^\alpha B_{idx_i} \right\} < \text{Bram}_{\text{num}} \\ \text{ExeTime}_i \approx \text{ExeTime}_j, \ \forall i, j \in (0, \alpha] \end{cases} \tag{4}$$

where

$$\sigma_{m_i} = \frac{M/T_{m_i}}{\lceil M/T_{m_i} \rceil}, \quad \text{et cetera} \tag{5}$$

$$\text{PARA}_i = T_{m_i} \times T_{n_i} \times T_{r_i} \times T_{c_i} \tag{6}$$

$$W_i = \begin{cases} 16b \times T_{m_i} \times T_{n_i} \times K_{1_i} \times K_{2_i}, \ i \in (0, \beta] \\ 16b \times T_{m_i} \times N_i \times 0.1, \ i \in (\beta, \alpha] \end{cases} \tag{7}$$

$$\text{FM}_i = \begin{cases} 16b \times N_i \times R_i \times C_i \times S_i^2, \ i \in (0, \beta] \\ 16b \times N_i \times \text{batch}, \ i \in (\beta, \alpha] \end{cases} \tag{8}$$

$$\text{IDX}_i = 14b \times T_{m_i} \times N_i \times 0.1 \tag{9}$$

$$B_{w_i} = \begin{cases} T_{m_i} \times T_{n_i} \times \lceil \frac{16b \times K_{1_i} \times K_{2_i}}{\text{BRAM}} \rceil, \ i \in (0, \beta] \\ T_{m_i} \times T_{n_i}, \ i \in (\beta, \alpha] \end{cases} \tag{10}$$

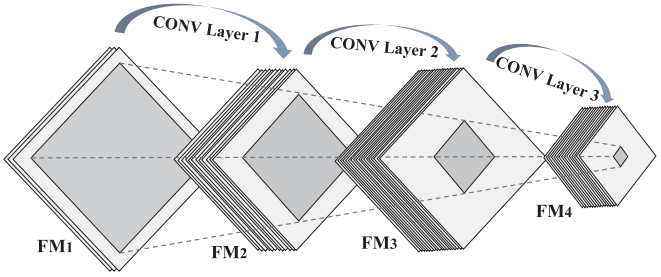


Fig. 6. Example of layer-fused technique.

$$B_{fm_i} = \begin{cases} T_{n_i} \times T_{r_i} \times T_{c_i} \times \lceil \frac{\text{FM}_i}{T_{r_i} \times T_{c_i} \times \text{BRAM}} \rceil, i \in (0, \beta] \\ T_{n_i} \times \lceil \frac{\text{FM}_i \times \text{batch}}{T_{n_i} \times \text{BRAM}} \rceil, \ i \in (\beta, \alpha] \end{cases} \tag{11}$$

$$B_{idx_i} = T_{m_i} \times T_{n_i} \tag{12}$$

$$\text{ExeTime}_i \propto \begin{cases} \lceil \frac{M_i \times N_i \times R_i \times C_i \times K_{1_i} \times K_{2_i}}{T_{m_i} \times T_{n_i} \times T_{r_i} \times T_{c_i} \times S_i^2} \rceil, i \in (0, \beta] \\ \lceil \frac{M_i \times N_i}{T_{m_i} \times T_{n_i}} \rceil, \ i \in (\beta, \alpha]. \end{cases} \tag{13}$$

2) Constraint Conditions for Large CNN Models: For some large-scale CNNs that the overall intermediate outputs are too large to be buffered on-chip simultaneously, we adopt layer fusion technique to decrease the data amount of buffered input feature maps required for each layer. The core idea of the layer fusion, as shown in Fig. 6, is that the computation related to each point in an output feature map of one CONV layer only depends on a subregion of the output feature maps of the previous layers, and therefore by adjusting the data flow and computation order, each CONV layer can only need to buffer a part of input feature maps at one time during computation. Besides, in order to decrease the iteration time, the overlapped input feature maps of each subregion sliding in a certain direction (R or C) will be buffered on-chip instead of being calculated and transmitted from the previous layer once again. However, for those CNNs with extremely large scale, the size of the subregions located at the front layers of the network will still be very large due to the influence of the POOL operation and the depth of the network, therefore, for these cases, we take partitioned layer fusion, which will first divide the CONV layers into several partitions and then fuse the layers within each partition, respectively. As a result, input feature maps of the layer, which serves as the connection between different partitions, need to be buffered on-chip completely. Actually, traditional layer-fusion is a special case of the partitioned layer fusion, in which all the layers are divided into one partition. Considering the main purpose of the partitioned layer fusion is to facilitate the network mapping, therefore, by default, to simplify the constraint conditions, here we separate the network partition from the optimization model, and agree that, from beginning of the network, successive layers with less than or equal to four times difference in the size of R and C loop dimensions will be divided into one partition when the partitioned layer fusion was adopted.

After layer fusion, the computation of the subregion within output feature maps of each layer can still be expressed as the sixfold loop iteration, and loop unrolling and loop tiling

shown in Algorithm 1 can also be applied. At this time, the independent variables are presented in (14), where R'_i and C'_i denote the size of the subregion within output feature maps after layer fusion in layer i, and other variables are the same as in (3). Correspondingly, the constraint conditions of each variable are illustrated in (15). In (15), the expression of each parameter is the same as that in (4)–(13) except FM_i, which are re-expressed in (16). In (16), A is the layer set that contains all CONV layers which served as the connection point between different layer partition, and B is another layer set that contains all the CONV layers that are not in the set A

$$\text{variables} = \left(R'_i, C'_i, T_{m_i}, T_{n_i}, T_{r_i}, T_{c_i}, \text{batch}\right), \quad i \in (0, \alpha] \tag{14}$$

$$\begin{cases} 1 \le R'_i \le R_i, \ \forall i \in (0, \beta] \\ 1 \le C'_i \le C_i, \ \forall i \in (0, \beta] \\ R'_{i+1} < R'_i, \ \forall \text{layer}_i, \text{layer}_{i+1} \in \text{the same partition} \\ C'_{i+1} < C'_i, \ \forall \text{layer}_i, \text{layer}_{i+1} \in \text{the same partition} \\ 1 \le T_{m_i} \le M_i \ \text{ and } \ 0.9 \le \sigma_{m_i}, \ \forall i \in (0, \alpha] \\ 1 \le T_{n_i} \le N_i \ \text{ and } \ 0.9 \le \sigma_{n_i}, \ \forall i \in (0, \alpha] \\ 1 \le T_{r_i} \le R_i \ \text{ and } \ 0.9 \le \sigma_{r_i}, \ \forall i \in (0, \beta] \\ 1 \le T_{c_i} \le C_i \ \text{ and } \ 0.9 \le \sigma_{c_i}, \ \forall i \in (0, \beta] \\ \sum_1^\alpha \mathrm{PARA}_i < \mathrm{DSP}_{\text{total}} \\ 2 * \left\{ \left(\sum_1^\alpha (W_i + \mathrm{FM}_i) + \sum_{\beta+1}^\alpha \mathrm{IDX}_i \right) \right\} < \mathrm{Bram}_{\text{cap}} \\ 2 * \left\{ \sum_1^\alpha \left(B_{w_i} + B_{fm_i}\right) + \sum_{\beta+1}^\alpha B_{idx_i} \right\} < \mathrm{Bram}_{\text{num}} \\ \mathrm{ExeTime}_i \approx \mathrm{ExeTime}_j, \ \forall i, j \in (0, \alpha] \end{cases} \tag{15}$$

where

$$\mathrm{FM}_i = \begin{cases} 16b \times N_i \times R_i \times C_i \times S_i^2, & i \in A \\ 16b \times N_i \times R'_i \times C'_i \times S_i^2, & i \in B \\ 16b \times T_{n_i} \times \text{batch}, & i \in (\beta, \alpha]. \end{cases} \tag{16}$$

3) Roofline Model Oriented Objective Function: For both paralleling strategies, after determining their respective constraint conditions, then we need to establish the corresponding objective functions with roofline model. More specifically, we need to get a variable set (paralleling scheme) with maximum throughput while achieving a higher value of CTC Ratio, as shown in (17) and (18). In (19), f denotes the hardware clock frequency, and we assume that the interlayers pipeline has been filled up, and layer x is the longest pipeline stage. Please note that the constraint conditions have ensured that the execution time of the longest pipeline stage is approximately equal to the average length of all stages. In (20), WGT_c and WGT_f denote the total amount of off-chip memory access for weight in CONV layers and FC layers, respectively, and IDX, IN, and OUT denote the total amount of off-chip memory access for the index, input, and output feature maps during one iteration, respectively. Here, one iteration corresponds to the inference process of batch inputs. The effect of network pruning in FC layers has been taken into account when calculating OP_f, WGT_f, and IDX in (22), (24), and (25), which will be described later

$$\text{Objective 1} = \text{Max } \{\text{Overall Throughput}\} \tag{17}$$

$$\text{Objective 2} = \text{Max } \{\text{CTC Ratio}\} \tag{18}$$

where

$$\begin{aligned} &\text{Overall Throughput} \\ &= \frac{\text{total number of operations}}{\text{execution time of the longest stage}} \\ &= \frac{\mathrm{OP}_c + \mathrm{OP}_f}{\left\lceil \frac{M_x \times N_x \times R_x \times C_x \times K_{1_x} \times K_{2_x}}{T_{m_x} \times T_{n_x} \times T_{r_x} \times T_{c_x} \times S_x^2} \right\rceil \times \text{batch} \times f} \end{aligned} \tag{19}$$

$$\begin{aligned} \text{CTC Ratio} &= \frac{\text{total number of operations}}{\text{total amount of off}-\text{chip memory access}} \\ &= \frac{\mathrm{OP}_c + \mathrm{OP}_f}{\mathrm{WGT}_c + \mathrm{WGT}_f + \mathrm{IDX} + \mathrm{IN} + \mathrm{OUT}} \end{aligned} \tag{20}$$

$$\mathrm{OP}_c = \text{batch} \times \sum_1^\beta M_i \times N_i \times R_i \times C_i \times K_{1_i} \times K_{2_i} \times \frac{1}{S_i^2} \tag{21}$$

$$\mathrm{OP}_f = \text{batch} \times \sum_{\beta+1}^\alpha M_i \times N_i \times 0.1 \tag{22}$$

$$\mathrm{WGT}_c = 16b \times \text{batch} \times \sum_1^\beta \left(M_i \times N_i \times K_{1_i} \times K_{2_i}\right) \tag{23}$$

$$\mathrm{WGT}_f = 16b \times \sum_{\beta+1}^\alpha (M_i \times N_i \times 0.1) \tag{24}$$

$$\mathrm{IDX} = 14b \times \sum_{\beta+1}^\alpha (M_i \times N_i \times 0.1) \tag{25}$$

$$\mathrm{IN} = 16b \times \text{batch} \times N_1 \times R_1 \times C_1 \times S_1^2 \tag{26}$$

$$\mathrm{OUT} = 16b \times \text{batch} \times M_\alpha. \tag{27}$$

Until now, we have finished the establishment of the optimization model for different paralleling strategies. The difference in the optimization model of two strategies lies in the different constraints that the objective functions need to satisfy. For this optimization model with multivariables, we can solve it by heuristic search like the genetic algorithm, and then implement the accelerator with the solved paralleling scheme.

C. Redundancy Elimination for FC Layers

Traditionally, the FC layers account for more than 90% of total weights in the CNNs. Since there is a lot of redundancy in these weights, a state-of-the-art pruning method proposed in 2015 [29], [30] has been widely adopted in FC layers processing, which can cut nearly 90% of the weights without loss of accuracy [4]. The pruned FC layers will be sparsity. Therefore, in our design, we enhance the PEs of FC layers to support sparse networks to eliminate the data communication as well as computation redundancy. Furthermore, in order to simplify the control complexities of computation logic and buffer accesses, we modify the pruning rule with a balanced manner, as shown in Fig. 7, which will eliminate as the equal number of the connected weights as possible for each output neuron. For those imbalanced output neurons, they can be balanced by filling zeros into their weights, as shown in Fig. 7, where the red dotted arrow in the right side denotes the appended zero. By this way, on the one hand, different output neurons can be calculated in a synchronous way with

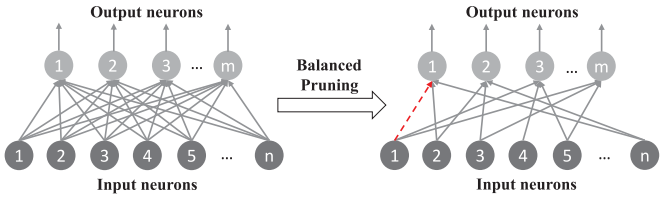


Fig. 7. Example of the balanced pruning in FC layers.

	Compressed Weights:				Location Information:			
Output Neuron 1:	0	w_{12}	...	w_{1n}	1	2	...	n
Output Neuron 2:	w_{22}	w_{23}	...	w_{2n}	2	3	...	n
Output Neuron 3:	w_{31}	w_{34}	...	w_{35}	1	4	...	5
...			...				...	
Output Neuron m:	w_{m1}	w_{m3}	...	w_{m5}	1	3	...	5

Fig. 8. Example of weight compression.

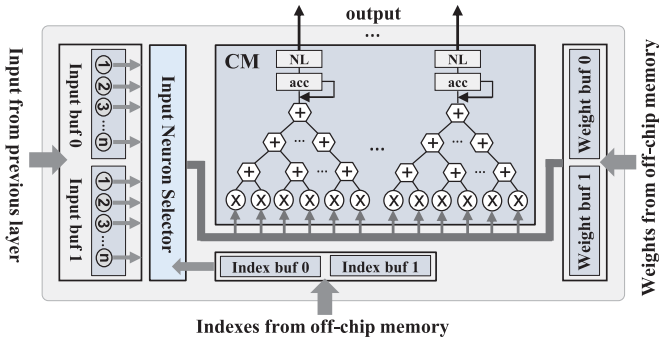


Fig. 9. Architecture of the PE for FC layer.

static control process since the data access becomes regular, and on the other hand, the total amount of weight data is easier to count compared to the traditional pruning so that it is beneficial for building constraint conditions described in Section III-B. After that, we need to compress the pruned weights of each output neuron and save the location information of each of these compressed weights by recording their indexes in the original (un-pruned) FC layer. Fig. 8 depicts the compressed weights and the related location information of the case shown in Fig. 7. In experimenting, we find that the accuracy losses caused by balanced pruning can be compensated effectively by increasing the number of iterations during training.

The architecture of the PE for FC layer is shown in Fig. 9. The main components of the PE include three ping-pong buffers for input data, compressed weights, and indexes (location information) of each buffered weights, respectively, a CM for inner-production operation. The input neurons and the weights are represented by fixed-16 data type, the indexes are represented by the unsigned integer. The bit wide of each index is determined by the number of input neurons of the corresponding layer, and generally, it will not exceed 14 bits. As we will see soon, the PE will skip the computation of pruned weights, and the computation amount of the FC layers will be greatly decreased, therefore, these layers traditionally will not

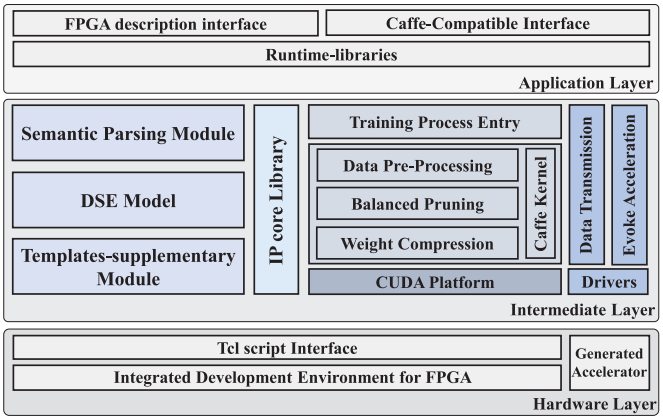


Fig. 10. Overall architecture of the proposed framework.

be the performance bottleneck of the interlayers pipeline even if they are implemented with tiny tiling sizes. The sizes of the weight and index buffers are set to the same value as the number of remaining weights of each output neuron multiplying with the tiling size of the output neurons.

During each iteration, the PE first accesses the compressed weights from weight buffer and send them to CM, and then the input neuron selector will access the index buffer to get the location information of the accessed weights. After that, the selector selects the related neurons from input buffer according to the location information, and then send them to the CM. Finally, CM inner-products the input data to the weights according to the order they entered. The inner production result within CM will be accumulated, and then either be stored or output directly after nonlinear translation, depending on whether it is the final result of the output neuron. The weights and indexes used for computing the current output neuron will continue to be used until the batch size computation rounds are completed. Meanwhile the weights and indexes, which are used for the next output neuron will be transferred to the corresponding ping-pong buffers, respectively. By doing so, the computation and data communication are overlapped, and only the remaining weights with the related input neurons are calculated, the redundancy from computation, data communication, and storage of the FC layers can be eliminated effectively.

IV. Programming Framework

To ease the programming burden and gain more performance portability, we propose a hierarchical framework which integrates the design methodology and optimizing strategies for our proposed accelerator between top-level programming interface and bottom-level IDE of FPGA design. Application designers can use high-level interfaces to describe the structure of their CNN model, and our framework will generate the corresponding FPGA accelerator with the data/control flow automatically. This process is transparent to the designer, and the only difference from the traditional software deep learning framework is that users need to give the hardware information and wait for the hardware synthesis process.

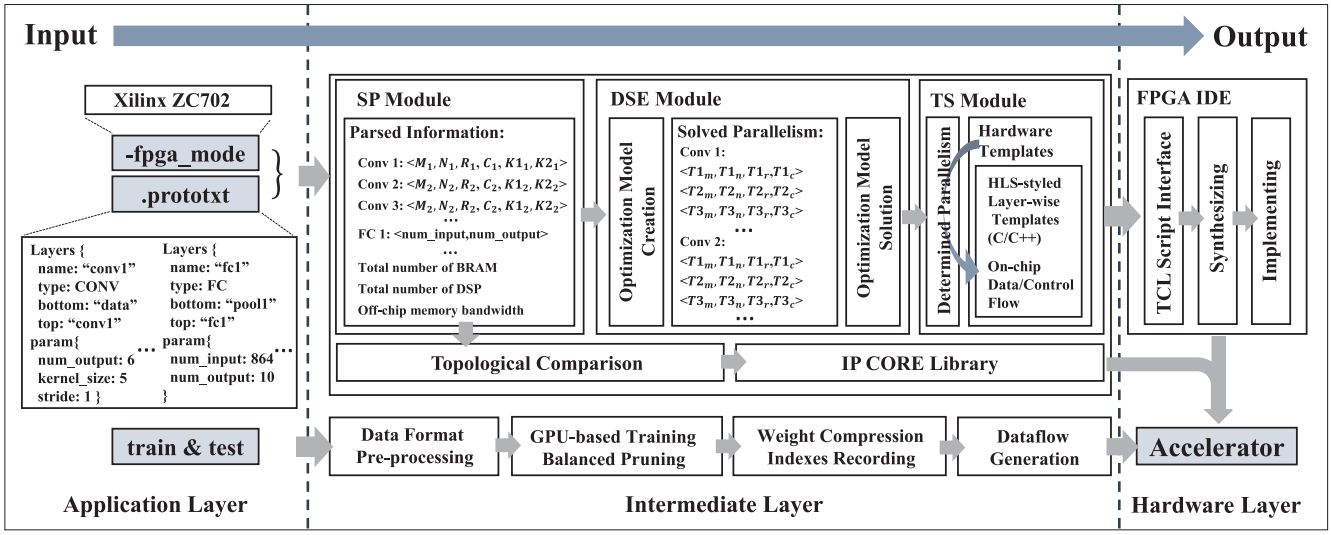


Fig. 11. Work flow of the proposed framework.

A. Hierarchical Architecture

The overall architecture of our proposed framework is illustrated in Fig. 10, which consists of an application layer, an intermediate layer, and a hardware implementation layer. In particular, each layer has specific features as below.

The application layer is running on the general purpose processor to provide the user programming interface and run-time environment. To eliminate the programming wall, we propose a high-level programming model compatible to Caffe [31], in which the detailed information of the CNN model including the number and order of the layers as well as the structural parameters of each layer can be described in the *.prototxt* file. Besides, we also add another interface to user for describing the hardware information of the integrated FPGA platform.

The intermediate layer is mainly responsible for translating the high-level described application to the corresponding hardware description, and managing the acceleration process after the accelerator is implemented. This layer consists of four parts. The first part on the left is used to generate the hardware description for the accelerator. It consists three components including a semantic parsing (SP) module, a DSE module, and a templates-supplementary (TS) module. The second part on the left is the IP core lib, in which we placed the synthesized bitstreams of commonly used CNN model including LeNet-5, AlexNet, Caffenet, Cifar-10, and VGGNet for different FPGA devices with our optimizing methodology in order to make it easier for users to quickly implement the accelerator. The third part on the left is responsible for preparing trained data. It integrates the training part of the Caffe, and we have modified the kernel functions of which to support balanced pruning with related weight compression. The last part on the right side consists of the data transmission and acceleration evoking interfaces, which is mainly used to evoke and manage the programmed FPGA device to perform acceleration.

The hardware implementation layer is mainly composed of the hardware design tools and the related script files to generate the accelerator. It sends the hardware description received from the intermediate layer to the underlying FPGA IDE, to complete the synthesizing and implementation of the accelerator.

B. Runtime Work Flow

The complete work flow of our proposed framework is illustrated in Fig. 11, from the user input to the final accelerator implementation. The application layer first receives the user inputs, and then passes it to the intermediate layer. In intermediate layer, the SP module parses the user input, and then converts the parsed information to the unified format which can be recognized by the DSE module. The information mainly includes the structural parameters of each layer of the targeted CNN model, and the hardware resources such as the total capacity of BRAM, the total number of DSP and BRAM modules, and the off-chip memory bandwidth of the given FPGA platform. Here, the off-chip memory bandwidth of mainstream FPGA devices we measured by the method proposed in [8]. Before DSE, the structural information will be used to retrieve the IP core lib. If the corresponding bitstream is found, the accelerator will be implemented directly. Otherwise, the DSE module will determine the corresponding paralleling strategy based on the size of the CNN model and the BRAM capacity of the FPGA platform, then brings them into the corresponding optimization model to solve the paralleling scheme of the accelerator. After that, the solved tiling sizes, batch size, and so on, will be used as paralleling parameters to fill in and supplement the hardware templates in the TS module. Each type of layer corresponds to a certain type of hardware template written in the HLS-based C++ code, in which all design elements related to parallelism including the number and organization of the deployed DSP module, as well as the number and partition manner of the deployed BRAM module, are carefully abstracted and can be initialized according to the substituted paralleling parameters. Different types of layers will be instantiated to multiple entities from different templates, while different layers of the same layer type

will be instantiated to multiple entities from the same template, and all instantiated entities will be synthesized to the PEs, respectively, in the FPGA IDE within hardware layer. Also, some other modules like the on-chip data/control flow related to each PE, the on-chip interconnection [32], and the data transfer module will be integrated with the synthesized PEs to generate the complete hardware project. Finally, the generated hardware project will be synthesized again and implemented as the Bitstream to return to the designer. All these processes are automated through a set of carefully crafted TCL script calls. Meanwhile, there is another execution path in the intermediate layer that will call the GPU/CUDA to perform balanced pruning-based training and weight compression. After FPGA is programmed, designers can evoke the accelerator to do inference process through the same software command interface as in the original Caffe framework.

V. Evaluation

To evaluate our proposal, we build four end-to-end accelerators for the modern CNNs with different network sizes on off-the-shelf FPGA platforms and compare them with CPU, GPU, and other FPGA accelerators regarding the performance and energy efficiency.

A. Experimental Setup

1) FPGA Implementation: We use Xilinx ZC702 and Digilent NetFPGA SUME development board as the target FPGA platforms, and implement accelerators for LeNet-5 and AlexNet on ZC702, and AlexNet and VGG16-C on NetFPGA SUME, respectively. ZC702 board is an embedded platform which has a Zynq-7020 FPGA chip, with 1-GB DDR3 off-chip memory and 4.2-GB/s off-chip memory bandwidth. NetFPGA SUME is a high performance FPGA platform which consists of a Xilinx Virtex-7 690t FPGA chip, and 4-GB DDR3 memory with 14.9-GB/s off-chip memory bandwidth. The clock frequencies of the accelerators implemented on ZC702 and NetFPGA SUME are 200 and 150 MHz, respectively. Each hardware PE is synthesized and generated with Xilinx Vivado HLS 2016.1, and the complete project is implemented using Xilinx Vivado 2016.1 IDE. Both these two tools are seamlessly integrated into our framework.

Since the on-chip memory capacity of ZC702 (612.5 KB) and NetFPGA (6.46 MB) is relatively small with respect to the data amount of the overall intermediate output of AlexNet (2×607.6 KB) and VGG16-C (2×17.39 MB) with fixed-16 data type, respectively, the layer fusion-based paralleling strategy is adopted for these implementations. Furthermore, the VGG16-C is implemented with the partitioned layer fusion, in which CONV 1 to CONV 7 are divided into one partition and CONV 8 to CONV 13 are divided into another partition, and accordingly, the input feature maps of CONV 8 need to be buffered on-chip totally. According to the solution results of the corresponding optimization model, the paralleling schemes for different accelerator implementations are shown in Tables I–IV, respectively, and the resource utilization of each implementation after placement and routing is shown in Table V.

TABLE I
Determined Parallelism for LeNet-5 on ZC702

$Layer$	T_m	T_n	T_r	T_c	$batch$
$CONV1$	2	1	12	3	1
$CONV2$	2	2	8	4	1
$FC3$	1	3	/	/	1
$FC4$	1	1	/	/	1
$FC5$	1	1	/	/	1

TABLE II
Determined Parallelism for AlexNet on ZC702

$Layer$	T_m	T_n	T_r	T_c	R'	C'	$batch$
$CONV1$	11	1	3	1	46	46	1
$CONV2$	12	2	3	1	18	18	1
$CONV3$	4	3	4	1	8	8	1
$CONV4$	3	4	3	1	6	6	1
$CONV5$	3	4	2	1	4	4	1
$FC6$	1	2	/	/	/	/	1
$FC7$	1	1	/	/	/	/	1
$FC8$	1	1	/	/	/	/	1

TABLE III
Determined Parallelism for AlexNet on NetFPGA

$Layer$	T_m	T_n	T_r	T_c	$batch$
$CONV1$	3	3	1	5	2
$CONV2$	14	12	1	7	2
$CONV3$	4	15	1	13	2
$CONV4$	3	15	1	13	2
$CONV5$	2	15	1	13	2
$FC6$	1	2	/	/	2
$FC7$	1	1	/	/	2
$FC8$	1	1	/	/	2

TABLE IV
Determined Parallelism for VGG16-C on NetFPGA

$Layer$	T_m	T_n	T_r	T_c	R'	C'	$batch$
$CONV1$	1	3	8	1	44	44	1
$CONV2$	6	12	7	1	42	42	1
$CONV3$	6	6	7	1	20	20	1
$CONV4$	7	12	6	1	18	18	1
$CONV5$	7	9	4	1	8	8	1
$CONV6$	12	14	3	1	6	6	1
$CONV7$	4	7	2	1	4	4	1
$CONV8$	7	9	4	1	18	18	1
$CONV9$	1	3	4	1	16	16	1
$CONV10$	2	4	7	1	14	14	1
$CONV11$	7	9	2	1	6	6	1
$CONV12$	7	9	2	1	4	4	1
$CONV13$	2	7	1	1	2	2	1
$FC14$	1	4	/	/	/	/	1
$FC15$	1	1	/	/	/	/	1
$FC16$	1	1	/	/	/	/	1

2) CPU Baseline: We use Caffe deep learning framework with Intel Core i7-4790K processor as the CPU baseline (CPU-Caffe), which works on the clock frequency at 4.00 GHz, including 16-GB DDR3 main memory, four physical cores, and eight threads.

TABLE V
FPGA RESOURCE UTILIZATION

Resource	DSP	BRAM	LUT	FF
Available on Zynq 7020	220	280	53200	106400
LeNet-5 on Zynq 7020	205	242	38136	42618
AlexNet on Zynq 7020	218	268	49823	61203
Available on Virtex-7 690t	3600	2940	433200	866400
AlexNet on Virtex-7 690t	2980	2192	305118	281841
VGG16 on Virtex-7 690t	2688	2365	320203	309227

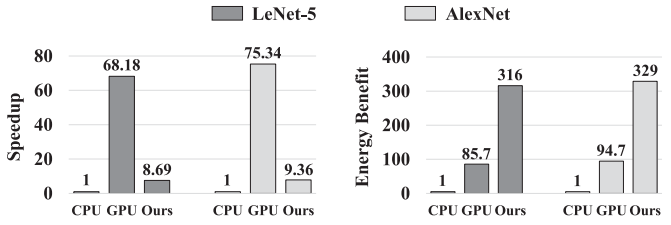


Fig. 12. Comparison with CPU/GPU implementations for LeNet-5 and AlexNet on ZC702.

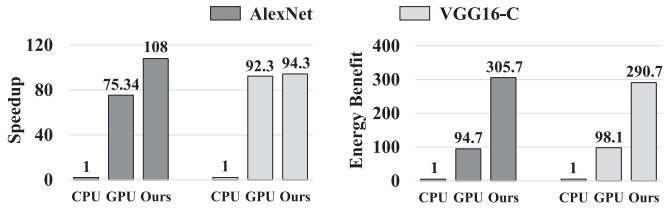


Fig. 13. Comparison with CPU/GPU implementations for AlexNet and VGG16-C on NetFPGA.

3) GPU Baseline: We use the state-of-the-art NVIDIA Tesla K40C GPU card, 12-GB GDDR5, working at 745–875 MHz frequency, integrated with 15 SMs, 2880 hardware threads. The GPU baseline is also running on Caffe (GPU-Caffe), with cuDNN 5.1 acceleration.

4) Accelerator Baseline: We also compare our accelerators against some representative FPGA-based accelerators [7]–[10], [12], [14], [26] with traditional accelerator architecture.

B. Experimental Results

We first compare the speedup and energy benefit for our accelerators against CPU-Caffe and GPU-Caffe. The comparison results of our implementations on ZC702 and NetFPGA are shown in Figs. 12 and 13, respectively. All result numbers are normalized to CPU-Caffe. For implementations on ZC702, our accelerators are 8.69× and 9.36× faster than CPU-Caffe for LeNet-5 and AlexNet, respectively, and comparing to GPU-Caffe, on average, we only achieve 0.125× speedup. This is mainly due to that ZC702 is an embedded platform, on which hardware resources are limited. In energy benefit, on average, our implementations achieve 322.2× and 3.58× more efficient than CPU-Caffe and GPU-Caffe, respectively. For implementations on NetFPGA, comparing to CPU-Caffe, our accelerators achieve 108× and 94.3× speedup for AlexNet and VGG16-C, respectively, and comparing to GPU-Caffe, we also achieve 1.43× and 1.02× for two networks, respectively. In energy benefit, comparing to CPU-Caffe, our implementations achieves 305.7× and 290.7× more efficient for AlexNet and VGG16-C, respectively, and comparing to GPU-Caffe, our implementations achieves 3.23× and 2.96× improvement for the two networks, respectively.

We further compare our designs with some previous representative FPGA-based CNN accelerators as illustrated in Table VI. For implementations on ZC702, we achieve 76.48 GOPS with energy efficiency 35.57 GOP/s/W, and 80.35 GOPS with energy efficiency 36.36 GOP/s/W for LeNet-5 and AlexNet, respectively. For implementations on NetFPGA, we achieve 910.2 GOPS with energy efficiency 29.94 GOP/s/W, and 829.84 GOPS with energy efficiency 26.6 GOP/s/W for AlexNet and VGG16-C, respectively. Compared with traditional accelerator architecture on the same hardware platform, our implementations on average can achieve 2.46× and 2.08× improvement for performance and energy efficiency, respectively. And compared with other multilayers implementation [26], we also achieve 2.27× speedup after the equivalent conversion of 16-bit fixed data type. There are three main reasons for the advantages of our implementations. First, each PE in our accelerator is only responsible for processing one single layer and is optimized independently, thus, avoiding the reduction of computational efficiency caused from PE multiplexing. Second, multiple layers are deployed independently and processing concurrently, therefore, the overall exploitable data-level parallelism is improved, which is also reflected in that more DSP resources in our implementations are employed. Finally, and most importantly, the increased data-level parallelism does not aggravate the off-chip memory traffic to lead to the bottleneck of memory access, which is benefited from the following aspects. First, the computation-intensive CONV layers and memory-intensive FC layers are calculated at the same time, instead of calculating one type of layer at a time, therefore, the utilization of off-chip bandwidth is more balanced. Second, the employed network pruning and semibatch processing techniques alleviate the off-chip memory access pressure of the FC layers significantly. Third, our proposed optimizing methodology based on roofline model oriented optimization model makes a good tradeoff and equipoise between on-chip computation and off-chip memory access, thus, avoiding the emergence of bottlenecks.

VI. RELATED STUDIES

In this part, we discuss some other related design and optimizing methodologies on hardware acceleration.

Recently, the optimization focus of hardware acceleration has been shifted from simple computation units to more comprehensive consideration and new techniques. Zhang *et al.* [8] first introduced roofline model into FPGA-based CNN accelerator designs to identify the bottleneck of CONV layers, and [9] extended this method into both CONV layers and FC layers by unifying the computation patterns of two types of layers. Xiao *et al.* [14] and Lu *et al.* [33] introduced *winograd* algorithm into CNN accelerator design to decrease the number of addition/multiplication needed by the convolution operation,

TABLE VI
COMPARISON WITH PREVIOUS FPGA-BASED CNN IMPLEMENTATIONS

	ISCA 2010[7]	*FPGA 2015*[8]	*FPL 2016*[26]	*ICCAD 2016*[9]	*FPGA 2016*[10]	*FCCM 2017*[12]	*DAC 2017*[14]	*This Work*			
Platform	Virtex-5 SX240T	Virtex-7 VX485T	Virtex-7 VX485T	Virtex-7 VX690T	Zynq Z-7045	Stratix-5 GSMD5	Zynq Z-7045	Zynq Z-7020		Virtex-7 VX690T	
Clock (MHz)	200	100	100	150	120	150	100	200	200	150	150
CNN Size (GOP)	0.52	1.33	1.33	30.76	30.76	30.76	30.76	5.3×10^{-4}	1.34	1.34	30.76
Precision	48-bit fixed	32-bit float	32-bit float	16-bit fixed	16-bit fixed	16-bit fixed	16-bit fixed	16-bit fixed	16-bit fixed	16-bit fixed	16-bit fixed
Performance (GOP/s)	16	61.62	80.11	354.00	136.97	364.36	229.5	76.48	80.35	910.2	829.84
Power (W)	14	18.61	-	26.00	9.63	25	9.4	2.15	2.21	30.4	31.2
Efficiency (GOP/s/W)	1.14	3.31	-	13.62	14.22	14.57	24.42	35.57	36.36	29.94	26.6

and therefore improves the utilization efficiency of DSP blocks.

From the programming perspective, Sharma *et al.* [16] and Wang *et al.* [34]–[36] proposed high-level design flow to simplify the acceleration process. Liu *et al.* [5] and Wang *et al.* [37], [38] improved the computational efficiency and parallelism of hardware acceleration through ISA customization and task-level out-of-order scheduling, respectively. Some other studies including [39] and [40] also tried to optimize the accelerator design from high-level FPGA design tools.

VII. CONCLUSION

In this paper, we propose a comprehensive design and optimizing methodology for FPGA-based CNN accelerators with all the network layers implemented on-chip. A roofline model oriented optimization model-based DSE is proposed to find the optimal paralleling scheme of each layer for the given CNN model and FPGA platform. Besides, a balanced pruning-based method is applied on the FC layers, which can reduce the redundancy significantly. Furthermore, in order to gain more performance portability, we propose a ubiquitous design framework between top-level modern deep learning programming interface and bottom level hardware IDE. As a case study, we implement three widely used CNN models including LeNet-5, AlexNet, and VGG16, on modern commercial FPGA platforms, and experimental results show that our methodology can achieve significant progress in both performance and energy efficiency over previous implementations.

ACKNOWLEDGMENT

The authors would like to thank the reviewers and editors for their valuable suggestions for this paper.

REFERENCES

[1] T. Chen *et al.*, "DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 269–284, 2014.

[2] Y. Chen *et al.*, "DaDianNao: A machine-learning supercomputer," in *Proc. MICRO*, Cambridge, U.K., 2015, pp. 609–622.

[3] Z. Du *et al.*, "ShiDianNao: Shifting vision processing closer to the sensor," in *Proc. ISCA*, Portland, OR, USA, 2015, pp. 92–104.

[4] S. Zhang *et al.*, "Cambricon-X: An accelerator for sparse neural networks," in *Proc. MICRO*, Taipei, Taiwan, 2016, pp. 393–405.

[5] S. Liu *et al.*, "Cambricon: An instruction set architecture for neural networks," in *Proc. ACM/IEEE Int. Symp. Comput. Archit.*, Seoul, South Korea, 2016, pp. 393–405.

[6] C. Wang *et al.*, "DLAU: A scalable deep learning accelerator unit on FPGA," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 36, no. 3, pp. 513–517, Mar. 2017.

[7] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, "A dynamically configurable coprocessor for convolutional neural networks," in *Proc. ISCA*, Saint-Malo, France, 2010, pp. 247–257.

[8] C. Zhang *et al.*, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *Proc. ACM/SIGDA FPGA*, Monterey, CA, USA, 2015, pp. 161–170.

[9] C. Zhang, Z. Fang, P. Pan, P. Pan, and J. Cong, "Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks," in *Proc. ICCAD*, Austin, TX, USA, 2016, p. 12.

[10] J. Qiu *et al.*, "Going deeper with embedded FPGA platform for convolutional neural network," in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays*, Monterey, CA, USA, 2016, pp. 26–35.

[11] N. Suda *et al.*, "Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks," in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays*, Monterey, CA, USA, 2016, pp. 16–25.

[12] Y. Guan *et al.*, "FP-DNN: An automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates," in *Proc. IEEE FCCM*, Napa, CA, USA, 2017, pp. 152–159.

[13] F. Sun *et al.*, "UniCNN: A pipelined accelerator towards uniformed computing for CNNs," *Int. J. Parallel Program.*, no. 3, pp. 1–12, 2017.

[14] Q. Xiao, Y. Liang, L. Lu, S. Yan, and Y. W. Tai, "Exploring heterogeneous algorithms for accelerating deep convolutional neural networks on FPGAs," in *Proc. DAC*, Austin, TX, USA, 2017, p. 62.

[15] L. Gong, C. Wang, X. Li, H. Chen, and X. Zhou, "A power-efficient and high performance FPGA accelerator for convolutional neural networks: Work-in-progress," in *Proc. Int. Conf. Hardw. Softw. Codesign Syst. Synth.*, Seoul, South Korea, 2017, pp. 1–2.

[16] H. Sharma *et al.*, "From high-level deep neural models to FPGAs," in *Proc. MICRO*, Taipei, Taiwan, 2016, pp. 1–12.

[17] C. Wang *et al.*, "SOLAR: Services-oriented deep learning architectures," *IEEE Trans. Services Comput.*, to be published.

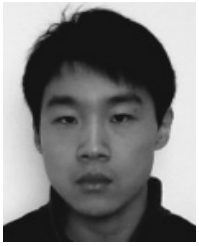
[18] Y. Lu, C. Wang, L. Gong, and X. Zhou, "SparseNN: A performance-efficient accelerator for large-scale sparse neural networks," *Int. J. Parallel Program.*, no. 11, pp. 1–12, 2017.

[19] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. NIPS*, 2012, pp. 1097–1105.

[20] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014.

[21] J. Mao, X. Chen, K. W. Nixon, C. Krieger, and Y. Chen, "MoDNN: Local distributed mobile computing system for deep neural network," in *Proc. DATE*, Lausanne, Switzerland, 2017, pp. 1400–1405.

[22] W. Choi *et al.*, "On-chip communication network for efficient training of deep convolutional networks on heterogeneous manycore systems," *IEEE Trans. Comput.*, vol. 67, no. 5, pp. 672–686, May 2018.
[23] C. Wang *et al.*, "Service-oriented architecture on FPGA-based MPSoC," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 10, pp. 2993–3006, Oct. 2017.
[24] F. N. Iandola *et al.*, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and 0.5MB model size," in *Proc. ICLR*, 2016, pp. 1–13.
[25] M. Courbariaux, Y. Bengio, and J.-P. David, "BinaryConnect: Training deep neural networks with binary weights during propagations," in *Proc. NIPS*, Montreal, QC, Canada, 2015, pp. 3123–3131.
[26] Y. Shen, M. Ferdman, and P. Milder, "Overcoming resource underutilization in spatial CNN accelerators," in *Proc. Int. Conf. Field Program. Logic Appl.*, Lausanne, Switzerland, 2016, pp. 1–4.
[27] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, 2009.
[28] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer CNN accelerators," in *Proc. MICRO*, Taipei, Taiwan, 2016, pp. 1–12.
[29] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding," *Fiber*, vol. 56, no. 4, pp. 3–7, 2015.
[30] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," in *Proc. 28th Int. Conf. Neural Inf. Process. Syst.*, Montreal, QC, Canada, 2015, pp. 1135–1143.
[31] Y. Jia *et al.*, "Caffe: Convolutional architecture for fast feature embedding," in *Proc. ACM Int. Conf. Multimedia*, Orlando, FL, USA, 2014, pp. 675–678.
[32] C. Wang, X. Li, J. Zhang, X. Zhou, and A. Wang, "A star network approach in heterogeneous multiprocessors system on chip," *J. Supercomput.*, vol. 62, no. 3, pp. 1404–1424, 2012.
[33] L. Lu, Y. Liang, Q. Xiao, and S. Yan, "Evaluating fast algorithms for convolutional neural networks on FPGAs," in *Proc. IEEE Int. Symp. Field Program. Custom Comput. Mach.*, Napa, CA, USA, 2017, pp. 101–108.
[34] C. Wang, X. Li, and X. Zhou, "SODA: Software defined FPGA based accelerators for big data," in *Proc. Design Autom. Test Europe Conf. Exhibit.*, Grenoble, France, 2015, pp. 884–887.
[35] C. Wang, X. Li, X. Zhou, N. Nedjah, and A. Wang, "Codem: Software/hardware codesign for embedded multicore systems supporting hardware services," *Int. J. Electron.*, vol. 102, no. 1, pp. 32–47, 2015.
[36] Y. Wang, J. Xu, Y. Han, H. Li, and X. Li, "DeepBurning: Automatic generation of FPGA-based learning accelerators for the neural network family," in *Proc. 53rd ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, Austin, TX, USA, 2016, pp. 1–6.
[37] C. Wang, J. Zhang, X. Li, A. Wang, and X. Zhou, "Hardware implementation on FPGA for task-level parallel dataflow execution engine," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 8, pp. 2303–2315, Aug. 2016.
[38] C. Wang *et al.*, "Architecture support for task out-of-order execution in MPSoCs," *IEEE Trans. Comput.*, vol. 64, no. 5, pp. 1296–1310, May 2015.
[39] W. Zuo *et al.*, "Improving high level synthesis optimization opportunity through polyhedral transformations," in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays*, Monterey, CA, USA, 2013, pp. 9–18.
[40] J. Liu, J. Wickerson, S. Bayliss, and G. A. Constantinides, "Polyhedral-based dynamic loop pipelining for high-level synthesis," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, to be published.



Lei Gong (S'18) received the B.S. degree from Nanchang Hangkong University, Nanchang, China, in 2014. He is currently pursuing the Ph.D. degree in computer science with the University of Science and Technology of China, Hefei, China.

His current research interests include deep learning accelerators and field-programmable gate array-based acceleration.

Chao Wang (M'11–SM'17) received the B.S. and Ph.D. degrees in computer science from the University of Science and Technology of China, Hefei, China, in 2006 and 2011, respectively.

He is an Assistant Professor with the Embedded System Laboratory, Suzhou Institute of University of Science and Technology of China, Suzhou, China. He has authored or co-authored over 20 publications and patents.

Dr. Wang is the Co-Chair of the International Symposium on Applied Reconfigurable Computing 2017, the International Conference on High Performance and Embedded Architectures and Compilers 2015, and the International Symposium on Parallel and Distributed Processing with Applications 2014. He serves as an Editorial Board Member of *Microprocessors and Microsystems: Embedded Hardware Design*, the *Institution of Engineering and Technology - Computers and Digital Techniques*, and the *International Journal of Business Process Integration and Management*.

Xi Li received the B.S. degree from the Chengdu University of Information Technology, Chengdu, China, in 1985, and the Ph.D. degree from the University of Science and Technology of China, Hefei, China, in 2003.

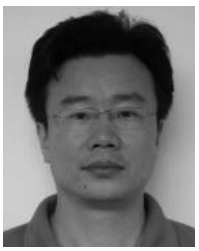
He is a Professor with the School of Computer Science and the Vice Dean with the School of Software Engineering, University of Science and Technology of China, Hefei, China, where he also directs the Research Programs of Embedded System Laboratory, examining various aspects of embedded system with the focus on performance, availability, flexibility, and energy efficiency. He has lead several national key projects of China, several national 863 projects, and NSFC projects.

Prof. Li is a member of ACM, and a Senior Member of CCF.

Huaping Chen received the B.S., M.S., and Ph.D. degrees from the University of Science and Technology of China, Hefei, China, in 1987, 1990, and 1997, respectively.

He is a Professor and the Dean with the School of Software Engineering, University of Science and Technology of China. He has lead several national key projects of China, several national 863 projects, and NSFC projects. His current research interests include high-performance computing, network computing, and intelligent computing.

Prof. Chen is a member of ACM, and a Senior Member of CCF.



Xuehai Zhou received the B.S., M.S., and Ph.D. degrees from the University of Science and Technology of China, Hefei, China, in 1987, 1990, and 1997, respectively.

He is a Professor with the School of Computer Science and the School of Software Engineering, University of Science and Technology of China. His current research interest includes various aspects of multicore and distributing systems.

Mr. Zhou serves as a General Secretary of Steering Committee of Computer College Fundamental Lessons, and the Technical Committee of Open Systems, CCF.