Going Deeper with Embedded FPGA Platform for Convolutional Neural Network

Jiantao Qiu[1,2], Jie Wang[1], Song Yao[1,2], Kaiyuan Guo[1,2], Boxun Li[1,2],Erjin Zhou[1], Jincheng Yu[1,2], Tianqi Tang[1,2], Ningyi Xu[3], Sen Song[2,4], Yu Wang[1,2], and Huazhong Yang[1,2]
[1]Department of Electronic Engineering, Tsinghua University
[1]Tsinghua National Laboratory for Information Science and Technology
[2]Center for Brain-Inspired Computing Research, Tsinghua University
[3]Hardware Computing Group, Microsoft Research Asia [4]School of Medicine, Tsinghua University
{songyao, yu-wang}@mail.tsinghua.edu.cn

ABSTRACT

In recent years, Convolutional Neural Network (CNN) based methods have achieved great success in a large number of applications and have been among the most powerful and widely used techniques in computer vision. However, CNN-based methods are computational-intensive and resource-consuming, and thus are hard to be integrated into embedded systems such as smart phones, smart glasses, and robots. FPGA is one of the most promising platforms for accelerating CNN, but the limited bandwidth and on-chip memory size limit the performance of FPGA accelerator for CNN.

In this paper, we go deeper with the embedded FPGA platform on accelerating CNNs and propose a CNN accelerator design on embedded FPGA for Image-Net large-scale image classification. We first present an in-depth analysis of state-of-the-art CNN models and show that Convolutional layers are computational-centric and Fully-Connected layers are memory-centric. Then the dynamic-precision data quantization method and a convolver design that is efficient for all layer types in CNN are proposed to improve the bandwidth and resource utilization. Results show that only 0.4% accuracy loss is introduced by our data quantization flow for the very deep VGG16 model when 8/4-bit quantization is used. A data arrangement method is proposed to further ensure a high utilization of the external memory bandwidth. Finally, a state-of-the-art CNN, VGG16-SVD, is implemented on an embedded FPGA platform as a case study. VGG16-SVD is the largest and most accurate network that has been implemented on FPGA end-to-end so far. The system on Xilinx Zynq ZC706 board achieves a frame rate at 4.45 fps with the top-5 accuracy of 86.66% using 16-bit quantization. The average performance of Convolutional layers and the full CNN is 187.8 GOP/s and 137.0 GOP/s under 150MHz working frequency, which outperforms previous approaches significantly.

This work was supported by 973 project 2013CB329000, National Natural Science Foundation of China (No. 61373026, 61261160501), the Importation and Development of High-Caliber Talents Project of Beijing Municipal Institutions, Microsoft, Xilinx University Program, and Tsinghua University Initiative Scientific Research Program.

FPGA'16, February 21-23, 2016, Monterey, CA, USA

DOI: http://dx.doi.org/10.1145/2847263.2847265

Keywords

Embedded FPGA; Convolutional Neural Network (CNN); Dynamic-precision data quantization; Bandwidth utilization

1. INTRODUCTION

Image classification is a basic problem in computer vision (CV). In recent years, Convolutional Neural Network (CNN) has led to great advances in image classification accuracy. In Image-Net Large-Scale Vision Recognition Challenge (ILSVRC) 2012 [1], Krizhevsky et al. showed that CNN had great power by achieving the top-5 accuracy of 84.7% in classification task [2], which was significantly higher than other traditional image classification methods. In the following years, the accuracy has been improved to 88.8% [3], 93.3% [4], and 96.4% [5] in ILSVRC 2013, 2014, and 2015.

While achieving state-of-the-art performance, CNN-based methods demand much more computations and memory resources compared with traditional methods. In this manner, most CNN-based methods have to depend on large servers. However, there has been a non-negligible market for embedded systems which demands capabilities of high-accuracy and real-time object recognition, such as auto-piloted car and robots. But for embedded systems, the limited battery and resources are serious problems.

To address this problem, many researchers have proposed various CNN acceleration techniques from either computing or memory access aspects [6, 7, 8, 9, 10, 11, 12, 13]. However, most of previous techniques only considered small CNN models such as the 5-layer LeNet for simple tasks such as MNIST handwritten digits recognition [14]. State-of-the-art CNN models for large-scale image classification have extremely high complexity, and thus can only be stored in external memory. In this manner, memory bandwidth becomes a serious problem for accelerating CNNs especially for embedded systems. Besides, previous research focused on accelerating Convolutional (CONV) layers, while the Fully-Connected (FC) layers were not well studied. Consequently, we need to go deeper with the embedded FPGA platform to address these problems.

In this paper, we make a deep investigation on how to deploy full CNNs to accelerators on embedded FPGA platform. A CNN accelerator for Image-Net large-scale classification is proposed, which can execute the very deep VGG16-SVD model at a speed of 4.45 fps. Specifically, this paper makes the following contributions.

- We present an in-depth analysis of state-of-the-art CNN models for large-scale image classification. We show that state-of-the-art CNN models are extremely complex (for example, the VGG16 model has 138 million weights and needs over 30 GOPs), CONV layers are computational-centric, and FC layers are memory-centric.
- For the first time, we present an automatic flow for dynamic-precision data quantization and explore various data quan-

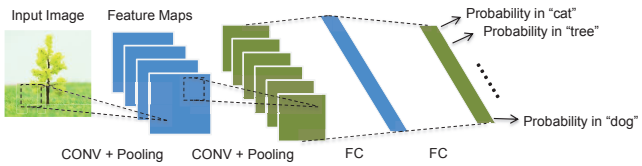


Figure 1: A typical CNN structure from the feature map perspective.

tization configurations. Results show that only a 0.4% accuracy loss is introduced with VGG16 model under 8/4 bit dynamic-precision quantization. Specific hardware is also designed to support dynamic-precision data quantization.

- We show that the performance of FC layers is mainly limited by the memory bandwidth on embedded FPGA platform, which is different from CONV layers. In this manner, we apply SVD to the weight matrix of the first FC layer, which reduces 85.8% memory footprint of this layer, design the convolvers that can compute FC layers to reduce resource consumption, and propose a data arrangement scheme to accelerate FC layers.
- We propose a CNN accelerator design on an embedded FPGA platform for Image-Net large-scale classification. On the Xilinx Zynq platform, our system achieves the performance at 187.8 GOP/s and 137.0 GOP/s for CONV layers and full CNN under 150 MHz frequency respectively. With VGG16-SVD network, our implementation achieves a top-5 accuracy of 86.66% at a 4.45 fps speed.

The rest of paper is organized as follows. In Section 2, the background of CNN is presented. In Section 3, the related work is introduced and discussed. We analyze the complexity distribution of state-of-the-art CNN models in Section 4. In Section 5, the dynamic-precision data quantization flow is proposed. The proposed image classification system design and implementation details are introduced in Section 6. The memory system and data arrangement method for FC layers are introduced in Section 7. The performance of the proposed system is evaluated and discussed in Section 8. We finally conclude this paper in Section 9.

2. BACKGROUND

Deep CNN achieves the state-of-the-art performance on a wide range of vision-related tasks. To help understand the CNN-based image classification algorithms analyzed in this paper, in this section, we introduce the basics of CNN. An introduction to the Image-Net dataset and state-of-the-art CNN models is also presented.

2.1 Primer on CNN

A typical CNN consists of a number of *layers* that run *in sequence*. The parameters of a CNN model are called "*weights*". The first layer of a CNN reads an input image and outputs a series of *feature maps*. The following layers read the feature maps generated by previous layers and output new feature maps. Finally a classifier outputs the probability of each category that the input image might belong to. CONV layer and FC layer are two essential types of layer in CNN. After CONV layers, there are usually pooling layers. A typical CNN example is shown in Figure 1.

In this paper, for a CNN layer, f_j^{in} denotes its j-th input feature map, f_i^{out} denotes the i-th output feature map, and b_i denotes the bias term to the i-th output map. For CONV layers, n_{in} and n_{out} represent the number of input and output feature maps respectively. For FC layers, n_{in} and n_{out} are the length of the input and output feature vector.

CONV layer takes a series of feature maps as input and convolves with convolutional kernels to obtain the output feature maps. A nonlinear layer, which applies nonlinear activation function to each element in the output feature maps is often attached to CONV layers. The CONV layer can be expressed with Equation 1:

$$f_i^{out} = \sum_{j=1}^{n_{in}} f_j^{in} \otimes g_{i,j} + b_i \qquad (1 \le i \le n_{out}), \tag{1}$$

where $g_{i,j}$ is the convolutional kernel applied to j-th input feature map and i-th output feature map.

FC layer applies a linear transformation on the input feature vector:

$$f^{out} = W f^{in} + b, \tag{2}$$

where W is an $n_{out} \times n_{in}$ transformation matrix and b is the bias term. It should be noted, for the FC layer, the input is not a combination of several 2-D feature maps but just a feature vector. Consequently, in Equation 2, the parameter n_{in} and n_{out} actually corresponds to the lengths of the input and output feature vector.

Pooling layer, which outputs the maximum or average value of each subarea in each feature maps, is often attached to the CONV layer. Max-pooling can be expressed as Equation 3:

$$f_{i,j}^{out} = \max_{p \times p} \begin{pmatrix} f_{m,n}^{in} & \cdots & f_{m,n+p-1}^{in} \\ \vdots & & \vdots \\ f_{m+p-1,n}^{in} & \cdots & f_{m+p-1,n+p-1}^{in} \end{pmatrix}, \tag{3}$$

where p is the pooling kernel size. This non-linear "down sampling" not only reduces the feature map size and the computation for later layers, but also provides a form of translation invariance.

CNN can be used to classify images in a forward inference process. But before using the CNN for any task, one should first *train* the CNN on a dataset. Recent research [15] showed that, a CNN model pre-trained on a large dataset for a given task can be used for other tasks and achieved high accuracy with minor adjustment in network weights. This minor adjustment is called "*fine-tune*". The training of the CNN is mostly implemented on large servers. For embedded FPGA platform, we only focus on accelerating the inference process of a CNN.

Table 1: # of layers in VGG models.

Model	CONV Group 1	CONV Group 2	CONV Group 3	CONV Group 4	CONV Group 5	FC	Total
VGG11	1	1	2	2	2	3	11
VGG16	2	2	3	3	3	3	16
VGG19	2	2	4	4	4	3	19

2.2 Image-Net Dataset

Image-Net [1] dataset is regarded as the standard benchmark to evaluate the performance of image classification and object detection algorithms. So far Image-Net dataset has collected more than 14 million images within more than 21 thousand categories. Image-Net releases a subset with 1.2 million images in 1000 categories for the ILSVRC classification task, which has significantly promoted the development of CV techniques. In this paper, all the CNN models are trained with ILSVRC 2014 training dataset and evaluated with ILSVRC 2014 validation set.

2.3 State-of-the-Art CNN Models

In ILSVRC 2012, the SuperVision team won the first place in image classification task using AlexNet by achieving 84.7% top-5 accuracy [2]. CaffeNet is a replication of AlexNet with minor changes. Both of AlexNet and CaffeNet consist of 5 CONV layers and 3 FC layers.

The Zeiler-and-Fergus (ZF) network achieved 88.8% top-5 accuracy and won the first place in image classification task of ILSVRC 2013 [3]. The ZF network also has 5 CONV layers and 3 FC layers.

The VGG model achieved a top-5 accuracy of 92.6% and won the second place in image classification task of ILSVRC 2014 [16]. VGG model consists of 5 CONV layer groups and 3 FC layers. According to the exact number of layers, there are several versions

of the VGG model including VGG11, VGG16, and VGG19, as listed in Table 1.

3. RELATED WORK

To accelerate CNN, a set of techniques from both software and hardware perspectives have been studied. From software perspective, the target is compressing CNN models in order to reduce the memory footprint and the number of operations while minimizing accuracy loss. From the hardware perspective, specific architecture and modules are designed to reuse data, enhance "locality" of data, and accelerate convolution operations. To deploy CNN models on embedded systems, the bit widths of operators and weights are often reduced compared to that on CPU or GPU platform.

3.1 Model Compression

Network pruning and decomposition were widely used to compress CNN models. In early work, network pruning proved to be a valid way to reduce the network complexity and over-fitting [17, 18, 19]. In [20], Han et al. pruned less influential connections in neural networks, and achieved 9× and 13× compression for CaffeNet and VGG16 model without accuracy loss. The Singular Value Decomposition (SVD) [21] is frequently used to reduce memory footprint. In [22], Denton et al. used SVD and filters clustering to speedup the first two FC layers of CNNs. Zhang et al. [23] proposed a method that was tested on a deeper model, which used low rank decomposition on network parameters and took nonlinear units into consideration. Jaderberg et al. [24] used rank-1 filters to approximate the original ones.

3.2 Data Quantization

Implementing fixed-point arithmetic units on ASIC and FPGA is much more efficient compared with floating-point ones. Consequently, most of previous CNN accelerators used fixed-point numbers instead of floating-point numbers [7, 25, 26, 6]. Shorter fixed-point representation of weights and data can also significantly reduce memory footprint and computation resources. For example, Chen et al. showed that the area and power of a 16-bit multiplier is 0.164× and 0.136× compared with that of 32-bit multiplier under 65nm fabrication technology [7].

Most of previous work adopted the 16-bit quantization strategy [27, 25, 7, 8]. In [7], Chen et al. showed that using 16-bit numbers instead of 32-bit ones only introduced 0.26% more error rate on MNIST dataset. In [8], 16-bit numbers were used in the inference process while 32-bit numbers were used in training process, and results on MNIST dataset showed that there was only 0.01% accuracy reduction.

To accelerate large CNN models on the embedded FPGA platform, data quantization is rather important and a shorter representation that introducing negligible accuracy loss is always expected. However, though previous work used data quantization, there is no comprehensive analysis of different quantization strategies.

3.3 CNN Accelerator

Previous CNN accelerator designs can be generally classified into two groups: the first group focuses on the computing engine and the second group aims to optimize the memory system.

CNNs are extremely computational-intensive, and thus powerful computing engines are necessary to accelerate them. Chakaradhar et al. in [11] proposed a dynamically configurable architecture for CNN. They added dedicated switches between the computing modules to enable design space exploration for dynamic configuration across different CNN layers. An associate compiler was also proposed to fully exploit the parallelism among the CNN workloads.

The weights in CONV layers of CNN are used for multiple times in computation, and thus the overall performance can be significantly degraded by frequent memory access. In [7], Chen et al. used the tiling strategy and dedicated buffers for data reuse to reduce the total communication traffic. In their further study [8], a multi-chip supercomputer was proposed which offered sufficient memory capacity to store all the weights in the CNN on chip. In [10], all the weights of one CNN layer were also stored in on-chip memory. In this manner, the data traffic between on-chip and off-chip memory could be minimized.

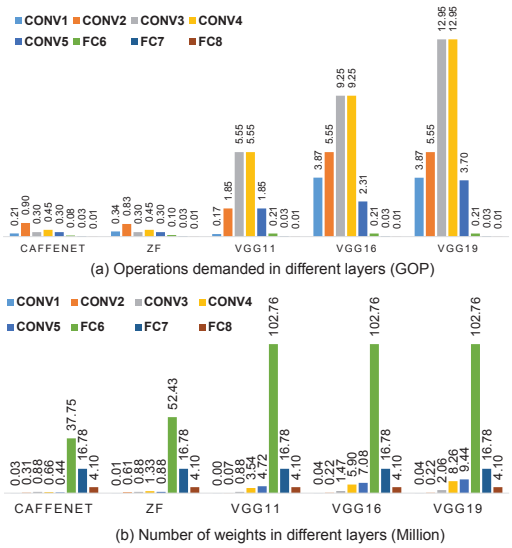


Figure 2: The complexity distribution of state-of-the-art CNN models: (a) Distribution of operations by theoretical estimation; (b) Distribution of weight number.

3.4 Motivation

State-of-the-art CNN models for large-scale visual recognition are much larger and deeper than early small CNN models. In this case, CNN accelerators such as ShiDianNao [10] which store weights on chip are hard to support those large CNN models. Consequently, state-of-the-art CNN models can only be stored in external memory and the bandwidth problem needs to be considered.

Most of previous studies focused on only accelerating the CONV layers of CNN. For example, in [6], the accelerator design was only applied to several CONV layers rather than the full CNN. In [26] and [11], authors only used models with few CONV layers without any FC layer. In this manner, those accelerators were hard to be used for accelerating full CNNs.

A full CNN model consists of both CONV layers and FC layers, and thus an efficient CNN accelerator for real-life applications need to consider both of them. For CONV layers and FC layers, the encountered problems are rather different. CONV layers are computation-centric: they contain few parameters but need a great deal of operations; FC layers are memory-centric: they usually contain hundreds of million weights, and each weight is used for only once. Consequently, loading weights from the external memory significantly degrades the performance of FC layers. In other words, the bandwidth limits the performance of FC layers. Considering this, we go deeper with the embedded FPGA platform on alleviating the bandwidth problem.

4. COMPLEXITY ANALYSIS OF CNN

Time complexity of a layer in CNN can be evaluated by the number of multiplication operations in the inference process. In a CONV layer, each convolutional kernel is a $k \times k$ filter applied to a $r \times c$ dimension input feature map. The number of kernels equals to $n_{in} \times n_{out}$. Consequently, according to Equation 1, the complexity of this CONV layer is

$$C_{CONV}^{Time} = O(n_{in} \cdot n_{out} \cdot k^2 \cdot r \cdot c). \quad (4)$$

Table 2: The Memory footprint, Computation Complexities, and Performance of the VGG16 model and its SVD version.

Network	FC6	# of total weights	# of operations	Top-5 accuracy
VGG16	25088×4096	138.36M	30.94G	88.00%
VGG16-SVD	25088×500 + 500×4096	50.18M	30.76G	87.96%

For pooling layers and FC layers, the time complexities are

$$C_{Pooling}^{Time} = O(n_{in} \cdot r \cdot c), \tag{5}$$

$$C_{FC}^{Time} = O(n_{in} \cdot n_{out}). \tag{6}$$

For pooling layers, n_{out} equals to n_{in} since each input feature map is pooled to a corresponding output feature map, and thus the complexity is linear to either input or output feature map number.

Space complexity refers to the memory footprint. For a CONV layer, there are $n_{in} \times n_{out}$ convolution kernels, and each kernel has k^2 weights. Consequently, the space complexity for a CONV layer is

$$C_{CONV}^{Space} = O(n_{in} \cdot n_{out} \cdot k^2). \tag{7}$$

FC layer actually applies a multiplication to the input feature vector, and thus the complexity for FC layer is measure by the size for the parameter matrix, which is shown in Equation 8:

$$C_{FC}^{Space} = O(n_{in} \cdot n_{out}) \tag{8}$$

No space is needed for pooling layers since it has no weight.

The distribution of demanded operations and weight numbers in the inference process of state-of-the-art CNN models are shown in Figure 2. The measured operations consist of multiplications, adds, and non-linear functions.

As shown in Figure 2 (a), **the operations of CONV layers compose most of the total operations of CNN models**, and thus the time complexity of CONV layers is much higher than that of FC layers. Consequently, for CONV layers, more attention should be paid to accelerate convolution operations.

For space complexity, the situation is quite different. As shown in Figure 2 (b), **FC layers contribute to most of the weights**. Since each weight in FC layers is used only once in one inference process, leaves no chance for reuse, the limited bandwidth can significantly degrade the performance since loading those weights may take quite long time.

Since FC layers contribute to most of memory footprint, it is necessary to reduce weights of FC layers while maintaining comparable accuracy. In this paper, SVD is adopted for accelerating FC layers. Considering an FC layer $f^{out} = Wf^{in} + b$, the weight matrix W can be decomposed as $W \approx U_d S_d V_d = W_1 W_2$, in which S_d is a diagonal matrix. By choosing the first d singular values in SVD, i.e. the rank of matrix U_d, S_d, and V_d, both time and space complexity can be reduced to $O(d \cdot n_{in} + d \cdot n_{out})$ from $O(n_{in} \cdot n_{out})$. Since accuracy loss may be minute even when d is much smaller than n_{in} and n_{out}, considerable reduction of time consumption and memory footprint can be achieved.

The effectiveness of SVD is proved by the results in Table 2. By applying SVD to the parameter matrix of the FC6 layer and choosing first 500 singular values, the number of weights in FC6 layers is reduced to 14.6 million from 103 million, which achieves a compression rate at 7.04×. However, the number of operations does not decrease much since the FC layer contributes little to total operations. The SVD only introduces 0.04% accuracy loss.

5. DATA QUANTIZATION

Using short fixed-point numbers instead of long floating-point numbers is efficient for implementations on the FPGA and can significantly reduce memory footprint and bandwidth requirements. A shorter bit width is always wanted, but it may lead to a severe accuracy loss. Though fixed-point numbers have been widely used in CNN accelerator designs, there is no comprehensive investigation on different quantization strategies and the trade-off between the bit length of fixed-point numbers and the accuracy. In this section, we propose a dynamic-precision data quantization flow and compare it with widely used static-precision quantization strategies.

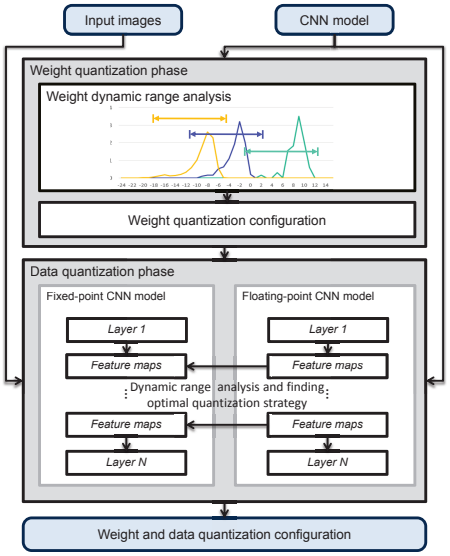


Figure 3: The dynamic-precision data quantization flow.

5.1 Quantization Flow

For a fixed-point number, its value can be expressed as

$$n = \sum_{i=0}^{bw-1} B_i \cdot 2^{-f_l} \cdot 2^i, \tag{9}$$

where bw is the bit width and f_l is the fractional length which can be negative. To convert floating-point numbers into fixed-point ones while achieving the highest accuracy, we propose a dynamic-precision data quantization strategy and an automatic workflow, as shown in Figure 3. Unlike previous static-precision quantization strategies, in the proposed data quantization flow, f_l **is dynamic for different layers and feature map sets while static in one layer to minimize the truncation error of each layer**. The proposed quantization flow mainly consists of two phases: the weight quantization phase and the data quantization phase.

The weight quantization phase aims to find the optimal f_l for weights in one layer, as shown in Equation 10:

$$f_l = \underset{f_l}{\operatorname{argmin}} \sum |W_{float} - W(bw, f_l)|, \tag{10}$$

where W is a weight and $W(bw, f_l)$ represents the fixed-point format of W under the given bw and f_l. In this phase, the dynamic ranges of weights in each layer is analyzed first. After that, the f_l is initialized to avoid data overflow. Furthermore, we search for the optimal f_l in the adjacent domains of the initial f_l.

The data quantization phase aims to find the optimal f_l for a set of feature maps between two layers. In this phase, the intermediate data of the fixed-point CNN model and the floating-point CNN model are compared layer by layer using a greedy algorithm to reduce the accuracy loss. For each layer, the optimization target is shown in Equation 11:

$$f_l = \underset{f_l}{\operatorname{argmin}} \sum |x_{float}^{+} - x^{+}(bw, f_l)|. \tag{11}$$

In Equation 11, x^+ represents the result of a layer when we denote the computation of a layer as $x^+ = A \cdot x$. It should be noted, for either CONV layer or FC layer, the direct result x^+ has longer bit width than the given standard. Consequently, truncation is needed when optimizing f_l selection. Finally, the entire data quantization configuration is generated.

Table 3: Exploration of different data quantization strategies with state-of-the-art CNNs.

Network	CaffeNet			VGG16							VGG16-SVD		
Experiment	**Exp 1**	**Exp 2**	**Exp 3**	**Exp 4**	**Exp 5**	**Exp 6**	**Exp 7**	**Exp 8**	**Exp 9**	**Exp 10**	**Exp 11**	**Exp 12**	**Exp 13**
Data Bits	Single-float	16	8	Single-float	16	16	8	8	8	8	Single-float	16	8
Weight Bits	Single-float	16	8	Single-float	16	8	8	8	8	8 or 4	Single-float	16	8 or 4
Data Precision	N/A	Dynamic	Dynamic	N/A	2^{-2}	2^{-2}	Not available	2^{-5} or 2^{-1}	Dynamic	Dynamic	N/A	Dynamic	Dynamic
Weight Precision	N/A	Dynamic	Dynamic	N/A	2^{-15}	2^{-7}	Not available	2^{-7}	Dynamic	Dynamic	N/A	Dynamic	Dynamic
Top 1 Accuracy	53.90%	53.90%	53.02%	68.10%	68.02%	62.26%	Not available	28.24%	66.58%	66.96%	68.02%	64.64%	64.14%
Top 5 Accuracy	77.70%	77.12%	76.64%	88.00%	87.94%	85.18%	Not available	49.66%	87.38%	87.60%	87.96%	86.66%	86.30%

[1] The weight bits "8 or 4" in Exp10 and Exp13 means 8 bits for CONV layers and 4 bits for FC layers.
[2] The data precision "2^{-5} or 2^{-1}" in Exp8 means 2^{-5} for feature maps between CONV layers and 2^{-1} for feature maps between FC layers.

5.2 Analysis of Different Strategies

We explore different data quantization strategies with CaffeNet, VGG16, and VGG16-SVD networks and the results are shown in Table 3. All results are obtained under Caffe framework [28].

- For CaffeNet, as shown in Exp 1, the top-5 accuracy is 77.70% when 32-bit floating-point numbers are used. When employing static-precision 16-bit quantization and 8/4-bit dynamic-precision quantization, the top-5 accuracy results are 77.12% and 76.64% respectively.
- VGG16 network with static-precision quantization strategies are tested in Exp 4 to Exp 8. As shown in Exp 4, single-float VGG16 network 88.00% top-5 accuracy. When using the 16-bit quantization configuration, only 0.06% accuracy loss is introduced. However, when employing 8-bit static-precision quantization, no configuration is available since the feature maps between FC layers are quantized to 0. As shown in Exp 8, at least two precisions are needed when using 8-bit quantization and the accuracy degrades greatly in this case.
- Results of VGG16 network with dynamic-precision quantization are shown in Exp 9 and Exp 10. When 8-bit dynamic-precision quantization is used for both data and weights, the top-5 accuracy is 87.38%. Using 8/4-bit dynamic-precision quantization for weights in CONV layers and FC layers respectively even achieves higher accuracy. As shown in Exp 10, in this case, the top-5 accuracy is 87.60%.
- The results of VGG16-SVD network are shown in Exp 11 to Exp 13. Compared with the floating-point VGG16 model, floating-point VGG16-SVD only introduces 0.04% accuracy loss. However, when 16-bit dynamic-precision quantization is adopted, the top-5 accuracy is down to 86.66%. With 8/4-bit dynamic-precision quantization, the top-5 accuracy further drops to 86.30%.

The results show that dynamic-precision quantization is much more favorable compared with static-precision quantization. With dynamic-precision quantization, we can use much shorter representations of operations while still achieving comparable accuracy. For example, compared with 16-bit quantization, 8/4-bit quantization halves the storage space for intermediate data and reduce three-fourths memory footprint of CNN models. Besides, the utilization of bandwidth can also be significantly increased.

6. SYSTEM DESIGN

In this section, we introduce the design of our CNN accelerator. First, the overall architecture is presented. After that, the designs of major modules are introduced. Finally, the implementation details are presented.

6.1 Overall Architecture

In this work, we propose a CPU+FPGA heterogeneous architecture to accelerate CNNs. Figure 4 (a) shows an overview of the proposed system architecture. The whole system can be divided into two parts: the Programmable Logic (PL) and the Processing System (PS).

PL is the FPGA chip, on which we place the *Computing Complex*, *On-chip Buffers*, *Controller*, and *DMAs*. The Computing Complex consists of Processing Elements (PEs) which take charge of the majority of computation tasks in CNN, including CONV layers, Pooling layers, and FC layers. On-chip buffers, including input buffer and output buffer, prepare data to be used by PEs and store the results. Controller fetches instructions from the external memory and decodes them to orchestrate all the modules except DMAs on the PL. DMAs are working for transferring data and instructions between the external memory on the PS side and On-chip Buffers on the PL side.

PS consists of general-purpose processors and the external memory. All the CNN model parameters, data, and instructions are stored in the external memory. Processors run bare-metal programs and help to orchestrate the whole inference phase by configuring the DMAs. We also realize Softmax function on CPU considering that its FPGA implementation will bring inevitable design overhead with little performance improvement since this function is called only in the last layer of the whole CNN.

The complete inference process of an image with the proposed CNN accelerator consists of three steps that are executed in sequence: data preparation, data processing, and result output.

Data Preparation. In this phase, all the data needed in the computation including image data, model data, and control data are stored in the external memory. Control data includes the Buffer Descriptors (BD) used by DMAs and instructions used by Controller. So far the image data is not obtained from the camera.

Data Processing. When all the data are prepared, CPU host starts to configure DMAs with the BDs that are pre-stored in the external memory. The configured DMA loads data and instructions to the controller, triggers a computation process on PL. Each time a DMA interrupt is asserted, CPU host adds up the self-maintained pointer address for each DMA's BD list and configures them with new BDs. This phase works until the last BD has been transferred.

Result Output. After receiving the interrupt of the last BD from DMA, the processor host applies Softmax function to the final results from PEs, and output the results to UART port.

6.2 PE Architecture

Figure 4 (b) shows the architecture of the PE and other modules involved. A PE consists of five parts, including the *Convolver Complex*, the *Adder Tree*, the *Non-Linearity* module, the *Max-Pooling* module, and the *Bias Shift*.

- For **Convolver Complex**, we employ the classical line buffer design [29] as shown in Figure 4 (c). When Input Data goes through the buffer in row-major layout, the line buffer releases a window selection function on the input image. Thus the selected window followed by multipliers and an adder tree will compute the convolution result, one data per cycle. Since the bottleneck of FC layers appears at the bandwidth, we use this module to compute matrix-vector multiplication for FC layers even the efficiency is not good. To realize this function, we set the delay of each line of the line buffer the same as the kernel size by using a MUX at the end of each line. In the proposed implementation, the kernel size is 3. When Input Data goes through the buffer, we get a totally new vector every 9 cycles in the selected window and do a vector inner product. Thus a convolver can do a matrix multiplied by a vector of size 9.

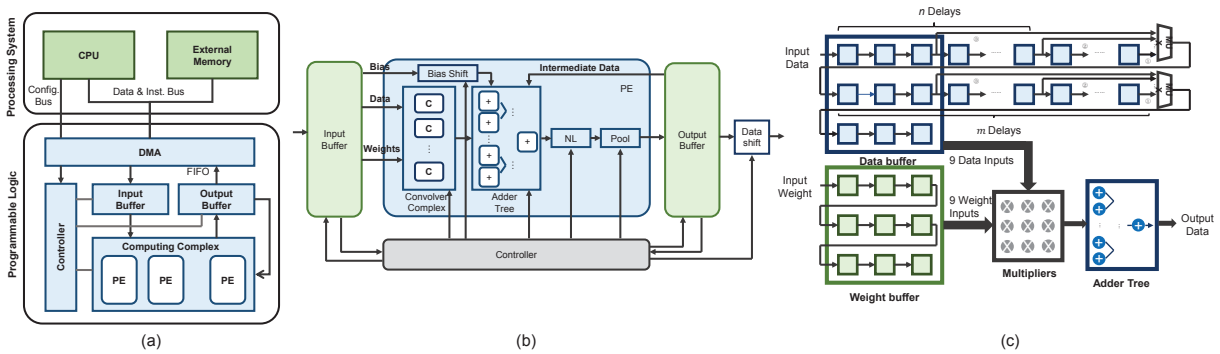


Figure 4: The design of our image classification system: (a) the overall architecture; (b) the processing element; (c) the convolver in the processing element.

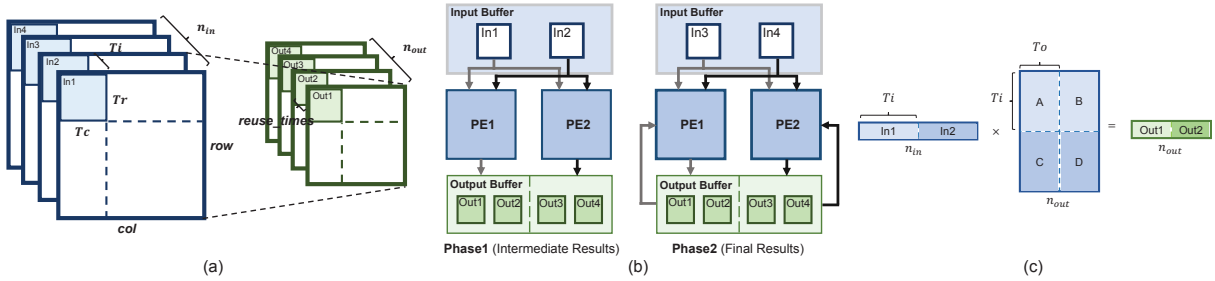


Figure 5: Workload schedule for CONV layers and FC layers: (a) Tiling and reuse of feature maps in CONV layers; (b) two phases in the execution of CONV layers; (c) workload schedule in FC layers.

- **Adder Tree** (AD) sums all the results from convolvers. It can add the intermediate data from Output Buffer or bias data from Input Buffer if needed.
- **Non-Linearity** (NL) module applies non-linear activation function to the input data stream.
- **Max-Pooling** module utilizes the line buffers to apply the specific 2×2 window to the input data stream, and outputs the maximum among them.
- **Bias Shift** module and **Data Shift** module are designed to support dynamic quantization. Input bias will be shifted by Bias Shift according to the layer's quantization result. For a 16-bit implementation, the bias is extended to 32-bit to be added with convolution result. The output data will be shifted by Data Shift and cut back to the original width.

The size of convolutional kernel usually has only several options such as 3×3, 5×5, and 7×7. All the convolutional kernels in the VGG16 model are in 3×3 dimension, and thus in the Convolver Complex, the 2D convolvers are designed for convolution operation only over a 3×3 window.

6.3 Implementation Details

6.3.1 Workloads Schedule

Parallelism. Chakradhar et al. pointed out that there are mainly three types of parallelism in CNN workloads: operator-level (fine-grained) parallelism, intra-output parallelism (multiple input features are combined to create a single output), and inter-output parallelism (multiple independent features are computed simultaneously) [11]. In our implementation, all the three types of parallelism are considered. The operator-level parallelism is realized with 2D convolvers. The intra-output parallelism is realized with multiple convolvers working simultaneously in each PE. The inter-output parallelism is realized by placing multiple PEs.

Tiling and Reuse. Due to limited on-chip memory, tiling is necessary for CNNs. For tiling in CONV layers, we tile each input image by the factor Tr (Tc) in row (column). And we tile the input (output) feature maps n_{in} (n_{out}) by the factor Ti (To). For FC layers, we tile each matrix into tiles of $Ti \times To$. For reuse, the times

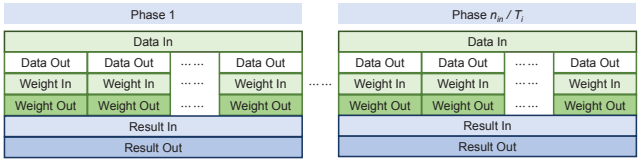


Figure 6: Timing graph. There are totally n_{in}/Ti phases to generate the $reuse\_times \times PE\_num$ tiles in the output layer. In each phase, the next group of data is loaded and the data preloaded in the last phase are output and reused for $reuse\_times$ times. Meanwhile, accompanied weights are loaded and output for $reuse\_times$ times with no reuse. The output buffer works on collecting data in the entire phase, while outputting intermediate data and final data to PEs or the external memory.

of each input tiled block (vector) to be reused is $reuse\_times$. We show how this workload schedule mechanism applies to CONV layers in Figure 5 (a) (b) and FC layers in Figure 5 (c).

6.3.2 Controller System

In each computation phase, the Controller decodes a 16-bit instruction to generate control signals for on-chip buffers and PEs. One instruction is composed with the following signals.

- **Pool Bypass** and **NL Bypass** are used to bypass the Pool and NL module if needed.
- **Zero Switch** is used to select either zero or bias data into added to the result of adder tree, since usually more than one phase is needed to calculate the final result and the bias should be added only once.
- **Result Shift** and **Bias Shift** describe the number of bits and direction for data shifting, for dynamic data quantization.
- **Write En** is used to switch the data from the Output Buffer either to the external memory or to the PEs to be reused.
- **PE En** offers us the flexibility to set several PEs as idle if needed. This can help save energy when computation capacity meet the demand.

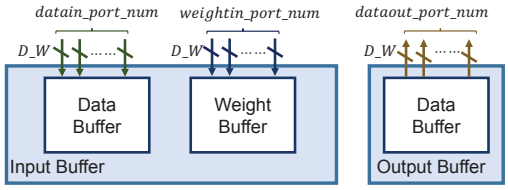


Figure 7: Buffer structure. Image data and weights are stored separately inside Input Buffer, and bias are stored in Data Buffer. The total bandwidth of each buffer is defined by corresponding port numbers multiplied by data width (D_W).

- **Phase Type** helps the Controller to distinguish these phases and send out the corresponding signals.helps the Controller to distinguish these phases and send out the corresponding signals. Several phases need to be specifically taken care of. For example, for the last phase in the last layer an the last output image, no more weights or data should be loaded in, and the input buffers should be configured differently compared to previous phases.
- **Pic Num** and **Tile Size/Layer Type** help the Controller to configure the Input Buffer and Output Buffer.

A compiler is developed on *Matlab* to automatically generate instructions. The compiler takes the fixed-point CNN model as the input and generates instructions as output. Table 4 shows the generated the instructions with the example in Figure 5 (a).

- Instruction 1 commands Input Buffer to load all the needed data, which is distinguished by the Phase Type signal. PE En enables two PEs working in parallel. As $Ti = 2$, Pic Num is set as 2. Tile Size is set as the defined Tr. Layer Type defines the layer type as CONV layer. All the other signals are useless in this phase.
- Instruction 2 starts calculating the four tiled blocks in the output layer. Since they are all intermediate results, Pool and NL modules are bypassed. Bias will be added in this phase only once. And Bias Shift specifies the shift configuration for bias data. Output Buffer will only collect the intermediate data and not write to anywhere.
- In instruction 3, Write En is set as "PE" to command Output Buffer to send the intermediate results back to the PEs. Bias is no longer added, and thus Zero Switch is set to zero. Since all the data generated in this phase is the final results, Pool and NL Bypass are disabled to let data from AD enter these two modules in sequence.
- In the last instruction, supposing this CONV layer is the last layer, then no module is working in PE. Write EN is set as "DDR" to command the Output Buffer to write results back to the external memory. Result Shift is set to shift the results data as we want. This phase is distinguished by Controller by setting Phase Type as last.

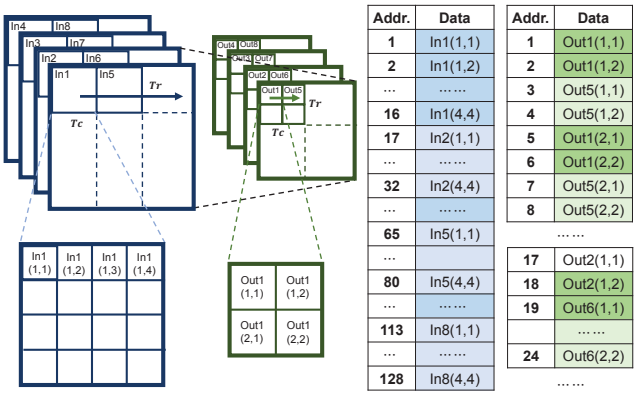
7. MEMORY SYSTEM

In this section, we introduce the memory system design which aims to feed the PEs with data efficiently. First the designs of buffers are introduced. After that, the data arrangement mechanisms for CONV and FC layers are presented.

7.1 Buffer Design

As shown in Figure 4 (a), there are two on-chip buffers on the PL side, the Input Buffer and the Output Buffer. The Input Buffer stores the bias, image data, and weights. The Output Buffer saves the results generated from PE and offers intermediate results to the PEs at proper time. For simplicity of illustration, we define three parameters as shown in Figure 7

- $datain\_port\_num$**.** The maximum amount of data that can be transferred by DMA each cycle.
- $weightin\_port\_num$**.** The maximum amount of weights that can be transferred by DMA each cycle.



Addr.	Data
1	In1(1,1)
2	In1(1,2)
...	.....
16	In1(4,4)
17	In2(1,1)
...	.....
32	In2(4,4)
...	.....
65	In5(1,1)
...	
80	In5(4,4)
...	.....
113	In8(1,1)
...	.....
128	In8(4,4)

Addr.	Data
1	Out1(1,1)
2	Out1(1,2)
3	Out5(1,1)
4	Out5(1,2)
5	Out1(2,1)
6	Out1(2,2)
7	Out5(2,1)
8	Out5(2,2)
.....	
17	Out2(1,1)
18	Out2(1,2)
19	Out6(1,1)
	.....
24	Out6(2,2)
.....	

Figure 8: Storage pattern for one CONV layer with max-pooling when the parameter group <Ti, To, $reuse\_times$, PE_num> is set to <2, 4, 2, 2>.

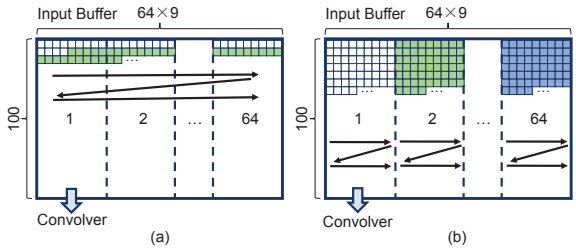


Figure 9: Data arrangement in external memory: (a) Linear arrangement; (b) DMA-oriented arrangement.

- $dataout\_port\_num$**.** The maximum amount of results that can be transferred by DMA each cycle.

In CONV layers, the total amount of weights needed in each phase is far less than that of image data, while in FC layers, the amount of weights is far more than the amount of data in input vectors. Therefore, we save the weights of FC layers in data buffer whose capability is larger than weight buffer, and save the input data vector in the weight buffer.

7.2 Data Arrangement for CONV layers

In order to reduce the unnecessary access latency of external memory, we optimize the storage pattern of data in the memory space. The principle is to maximize the burst length of each DMA transaction. Figure 8 shows a brief example of how we organize the input and output data in one CONV layer with max-pooling. We store the tiles which are at the same relative locations in each picture continuously. Therefore, in each phase, we can load all the input tiles for computation continuously. The output feature maps will be the input feature maps of the next layer, therefore, the same storage pattern applies as well.

There is a slight difference between CONV layers with Pooling and other layers. After a 2×2 pooling, the result is only a quarter of a tile. In Figure 8, *Out(2, 1)*, instead of *Out(1,2)*, will be calculated after *Out(1,1)*. This means adjacent result tiles are not stored continuously in external memory. If we write each result tile as soon as it is generated, the burst length will be only $Tr/2$. This will significantly degrade the utilization of the external memory. To solve this problem, we increase the memory budget on chip. We buffer *Out(1,1)* to *Out(4,1)* before generating *Out(1,2)*, then write *Out(1,1)* and *Out(1,2)* together. This strategy increases the burst length to $Tr \times Tc/2$.

7.3 Data Arrangement for FC Layers

The speed of computing FC layers is mainly restricted by the bandwidth. In this manner, using specific hardware to accelerate

Table 4: Instructions for One CONV layer generated by the compiler.

Index	Pool Bypass	NL Bypass	Zero Switch	Result Shift	Bias Shift	Write En	PE En	Phase Type	Pic Num	Tile Size	Layer Type
1	X	X	X	X	X	No	2	First	2	Tr	CONV
2	Yes	Yes	Bias	X	BS	No	2	Cal	2	Tr	CONV
3	No	No	Zero	X	X	PE	2	Cal	2	Tr	CONV
4	X	X	X	RS	X	DDR	2	Last	2	Tr	CONV

Table 5: Parameter configuration and resource utilization.

Param.	tile_size	convolver_num	PE_num	reuse_times
Config.	28	64	2	16
Param.	datain_port_num	weightin_port_num	dataout_port_num	
Config.	8	4	2	

Resource	FF	LUT	DSP	BRAM
Utilization	127653	182616	780	486
Percent(%)	29.2	83.5	89.2	86.7

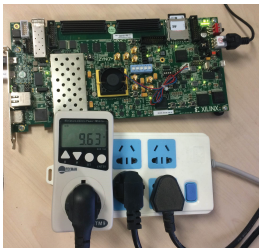


Figure 10: Testing platform. We use Xilinx Zynq ZC706 for on-board testing. A power meter is used for power analysis.

FC layers is not effective. Considering this, the proposed system uses the Convolver Complex in one of the PEs to do the computation for FC layers. In this case, we need to fully utilize the bandwidth of the external memory with the current PL structure.

In our system, we assign a buffer of length 900, the same as $Tr \times Tr$ to each of the 64 Compute Complex in one PE. The buffers are filled one by one when computing CONV layers. To reduce extra data routing logic for filling buffers while keep a long burst length when fetching data for computing FC layers, we arrange the weight matrix in the external memory. We first divide the whole matrix with blocks of 64×9 columns and 100 rows such that one block can be processed in a phase. In each block, the data is arranged as shown in Figure 9 (b). Without data arrangement for FC layers, as shown in Figure 9 (a), we need **64×100** DMA transactions to load one block while the burst length is just 9. By arranging the data following Figure 9 (b), we need just one DMA transaction to load the whole block and the long burst length ensures a high utilization of the bandwidth of external memory.

8. SYSTEM EVALUATION

In this section, the performance of the implemented system is evaluated. First, we analyze the performance of our system architecture under given design constraints. After that, the performance of the proposed system is presented and compared with other platforms. Finally we compare our system with previous FPGA-based CNN accelerators.

We use 16-bit dynamic-precision quantization and Xilinx Zynq ZC706 for the implementation. Xilinx Zynq platform consists of a Xilinx Kintex-7 FPGA, dual ARM Cortex-A9 Processor, and 1 GB DDR3 memory. It offers a bandwidth of up to 4.2GB/s. All the synthesis results are obtained from Xilinx Vivado 2014.4. We first synthesize each module in Vivado to figure out the resource utilization. Then we choose the optimal parameter group to maximize the throughput with the resource and bandwidth constraints. The parameters and resource utilization are shown in Table 5. We can see that our parameter configuration helps to maximize the resource utilization. Figure 10 shows the hardware platform.

The CPU platform is Intel Xeon E5-2690 CPU@2.90GHz. The GPU platform is Nvidia K40 GPU (2880 CUDA cores with 12GB GDDR5 384-bit memory), and the mGPU platform is the Nvidia TK1 Mobile GPU development kit (192 CUDA cores). For experiments on CPU, GPU, and mGPU, the operating system is Ubuntu 14.04 and the deep learning software framework is Caffe [28].

8.1 Theoretical Estimation

For CONV layer, the number of phases needed in one CONV layer when tiling is adopted can be calculated by the following formula:

$$N_{phase}^{CONV} = \lceil \frac{n_{in}}{Ti} \rceil \times \lceil \frac{n_{out}}{To} \rceil \times \lceil \frac{row}{Tr} \rceil^2,$$

where $To = reuse\_times \times PE\_num$ and $Tc = Tr$. The time of computation and loading data in each phase are:

$$t_{compute\_data}^{CONV} \approx Tr^2 \times reuse\_times,$$

and

$$t_{load\_data}^{CONV} = \frac{Tr^2 \times Ti}{datain\_port\_num}.$$

CONV layers are usually computation-intensive. Consequently, in order to keep ping-pong mechanism working, typically t_{load} is smaller than $t_{compute}$, and thus there should be:

$$datain\_port\_num^{CONV} \geq \frac{Ti}{reuse\_times}.$$

In each phase, data will be reused for $reuse\_times$ times, each accompanied with a new group of weights (9 weights for each kernel as for the model we use). Therefore, for weights, we have:

$$t_{compute\_weight}^{CONV} \approx Tr^2$$
$$t_{load\_weight}^{CONV} = \frac{9 \times Ti \times PE\_num}{weightin\_port\_num}$$
$$weightin\_port\_num^{CONV} \geq \frac{9 \times Ti \times PE\_num}{Tr^2}.$$

According to the workloads schedule shown in Figure 6, the constraint for $dataout\_port\_num$ is:

$$dataout\_port\_num^{CONV} \geq PE\_num.$$

In order to minimize the bandwidth consumption, we consider to choose $weightin\_port\_num$ and $datain\_port\_num$ as fewer as possible. Under the above constraints, we can estimate the computation time for one CONV layer:

$$t_{CONV} = N_{Phase} \times t_{compute}$$
$$= \lceil \frac{n_{in}}{Ti} \rceil \times \lceil \frac{n_{out}}{To} \rceil \times \lceil \frac{row}{Tr} \rceil^2 \times Tr^2 \times reuse\_times.$$

Considering that $Ti = convolver\_num$, $Tr = tile\_size$ and $To = reuse\_times \times PE\_num$ in CONV layers, we further get:

$$t_{CONV} = \lceil \frac{n_{in}}{convolver\_num} \rceil \times \lceil \frac{n_{out}}{reuse\_times \times PE\_num} \rceil$$
$$\times \lceil \frac{row}{tile\_size} \rceil^2 \times tile\_size^2 \times resue\_times.$$
$$\approx \frac{n_{in} \times n_{out} \times row^2}{convolver\_num \times PE\_num}$$

Table 6: Performance of different platforms with VGG16-SVD network.

Platform	Embedded FPGA				CPU	GPU	mGPU	CPU	GPU	mGPU
Layer (Group)	Theoretical Computation Time (ms)	Real Computation Time (ms)	Total Operations (GOP)	Real Performance (GOP/s)	Real Computation Time (ms)			Real Performance (GOP/s)		
CONV1	21.41	31.29	3.87	123.76	83.43	2.45	59.45	46.42	1578.8	65.15
CONV2	16.06	23.58	5.55	235.29	68.99	3.31	79.73	80.44	1675.5	69.60
CONV3	26.76	39.29	9.25	235.38	76.08	4.25	89.35	151.57	2177.1	103.51
CONV4	26.76	36.30	9.25	254.81	62.53	3.31	107.49	147.91	2791.6	86.04
CONV5	32.11	32.95	2.31	70.16	12.36	2.30	63.75	186.99	1003.5	36.27
CONV Total	**123.10**	**163.42**	**30.69**	**187.80**	**312.36**	**15.45**	**399.77**	**98.26**	**1986.0**	**76.77**
FC6-1	10.45	20.17	0.025	1.24	1.69	0.445	29.35	14.87	56.404	0.86
FC6-2	1.71	3.75	0.0041	1.09	0.26	0.031	5.26	15.65	132.26	0.78
FC7	13.98	30.02	0.034	1.12	1.86	0.19	14.74	18.04	177.78	2.28
FC8	3.413	7.244	0.0082	1.13	0.46	0.96	4.58	17.75	8.56	1.79
FC Total	**29.55**	**61.18**	**0.073**	**1.20**	**4.28**	**1.79**	**53.93**	**17.17**	**40.98**	**1.36**
Total	**152.65**	**224.60**	**30.76**	**136.97**	**316.64**	**17.25**	**453.70**	**97.16**	**1783.9**	**67.81**

For FC layers, the number of phases and the time of different tasks can be estimated with the following equations:

$$N_{phase}^{FC} = \lceil \frac{n_{in}}{Ti} \rceil \times \lceil \frac{n_{out}}{To \times PE\_num} \rceil$$

$$t_{load\_data}^{FC} = \frac{PE\_num \times Ti \times To}{datain\_port\_num}$$

$$t_{load\_weight}^{FC} = \frac{Ti}{weightin\_port\_num}$$

$$t_{compute\_data}^{FC} = t_{compute\_weight}^{FC} = \frac{Ti \times To}{convolver\_num}.$$

Typically, for FC layers, $t_{compute}^{FC}$ is much smaller than t_{load}^{FC}, and thus the total cycles needed by one FC layer can be estimated as:

$$\begin{aligned} t_{FC} &= N_{Phase} \times t_{load} \\ &= \lceil \frac{n_{in}}{Ti} \rceil \times \lceil \frac{n_{out}}{To \times PE\_num} \rceil \times \frac{PE\_num \times Ti \times To}{datain\_port\_num} \\ &\approx \frac{n_{in} \times n_{out}}{datain\_port\_num}. \end{aligned}$$

In summary, under the given constraints, the runtime of a CONV layer and an FC layer can be estimated through Equation 13 and Equation 12:

$$t_{FC} = \frac{n_{in} \times n_{out}}{datain\_port\_num}, \tag{12}$$

$$t_{CONV} = \frac{n_{in} \cdot n_{out} \cdot row^2}{convolver\_num^2 \times PE\_num}. \tag{13}$$

As shown in Equation 13, CONV layers are bounded both by bandwidth and computation resources. For FC layers, as shown in Equation 12, it is bandwidth-bounded only. Consequently, higher bandwidth can help reduce the runtime of FC layers.

8.2 Performance Analysis

Though the performance of FC layers on FPGA is limited by the bandwidth, it is still higher than ARM processors. Consequently, in our implementation, the FC layer workloads are placed on FPGA.

The performance of our system, CPU, GPU, and mGPU is shown in Table 6. The VGG16-SVD network needs 30.764 GOPs including multiplications, adds, and non-linear functions. Our system achieves an average performance of 187.80 GOP/s for CONV layers and 136.97 GOP/s for the whole network. The frame rate of our system is 4.45 fps, which is 1.4× and 2.0× faster than the CPU and mGPU platform (the power of CPU and mGPU are 135W and 9W respectively). The overall performance of GPU is 13.0× higher than our implementation, but it consumes 26.0× more power compared with embedded FPGA (250W versus 9.63W).

The performance of our system on FC layers is much lower than that of CONV layers even though data arrangement method is adopted due to the limited bandwidth. Consequently, though the number of operations needed by FC layers is only 0.0024× of CONV layers, the runtime of FC layers is 0.374× of CONV layer. The mGPU platform suffers from the same problem due to the limited bandwidth.

Compared with theoretical estimation, there is around 47% performance degradation for on-board test, as shown in the 2nd column and the 3rd column of Table 6. One possible reason is the DDR access latency. The other possible reason is that different DMAs are working asynchronously, since different DMA transactions may affect each other and reduce the total efficiency of bandwidth usage.

Table 7: Comparison with other FPGA accelerators.

	[11]	[30]	[6]	Ours
Year	2010	2014	2015	2015
Platform	Virtex5 SX240t	Zynq XC7Z045	Virtex7 VX485t	Zynq XC7Z045
Clock(MHz)	120	150	100	150
Bandwidth (GB/s)	–	4.2	12.8	4.2
Quantization Strategy	48-bit fixed	16-bit fixed	32-bit float	16-bit fixed
Power (W)	14	8	18.61	9.63
Problem Complexity (GOP)	0.52	0.552	1.33	30.76
Performance (GOP/s)	16	23.18	61.62	187.80 (CONV) 136.97 (Overall)
Resource Efficiency (GOP/s/Slices)	4.30×10^{-4}	–	8.12×10^{-4}	3.58×10^{-3} (CONV) 2.61×10^{-3} (Overall)
Power Efficiency (GOP/s/W)	1.14	2.90	3.31	19.50 (CONV) 14.22 (Overall)

Table 8: Projected frame rates on Zynq/VC707 board using 16-bit and 8/4-bit quantization with VGG16-SVD network.

Platform	Total Resources			16-bit Quantization		8-bit Quantization	
	LUT	FF	Bandwidth	# of PE	**FPS**	# of PE	**FPS**
Zynq	218600	437200	4.2GBps	2	**4.45**	4	**8.9**
VC707	303600	607200	4.2GBps	3	**5.88**	6	**11.76**

8.3 Design Comparison

As shown in Table 7, we compare our CNN accelerator with previous work. In [30], the design was verified on 3 models, including a single CONV layer, a model consisting of 2 CONV layers for face recognition, and a model for street parsing without structure details. Since the first model lacks generality and structure details of the third model were not provided, results of a 2-layer CNN model is adopted for comparison. We also transform the unit $GFLOP$ in [6] into GOP for comparison.

In summary, our accelerator achieved the highest performance, resource efficiency, and power efficiency compared with previous designs. It should be noted, all the performance results of previous designs were obtained from CONV layers only. If we only consider CONV layers, the average performance of our system is 187.80

GOP/s, which is several times higher than previous designs. The performance of our system with the full VGG16-SVD network is 136.97 GOP/s.

8.4 Discussion

At present, our implementation uses 16-bit fixed-point numbers and Zynq board. The projected results with different quantization strategies and platforms are shown in Table 8. Theoretically, when using the 8-bit quantization, $2\times$ PEs can be placed on the FPGA and thus the performance on CONV layers doubles. Besides, $2\times$ weights can be loaded to the system with the same bandwidth compared with the 16-bit quantization, and thus the performance on FC layers also doubles. Further more, when deploying to the VC707 board, one more PE can be placed, and thus the processing capability on CONV layers is expected to be $1.5\times$ higher than that of Zynq platform. For VGG16-SVD network on VC707 with 8-bit dynamic-precision quantization, it is expected to achieve a frame rate at 11.76 fps.

9. CONCLUSION

The limited bandwidth is one of the bottlenecks of accelerating deep CNN models on embedded systems. In this paper, we make an in-depth investigation of the memory footprint and bandwidth problem in order to accelerate state-of-the-art CNN models for Image-Net classification on the embedded FPGA platform. We show that CONV layers are computation-centric and FC layers are memory-centric. A dynamic-precision data quantization flow is proposed to reduce memory footprint and bandwidth requirements while maintaining comparable accuracy. Convolver that can be used for both CONV layers and FC layers is designed to save the resource. A data arrangement scheme for FC layers is also proposed to ensure high bandwidth utilization. Our implementation on Xilinx Zynq with very deep VGG16-SVD model for Image-Net classification achieves a frame rate at 4.45 fps with 86.66% top-5 accuracy with 16-bit quantization. The average performances of CONV layers and the full CNN are 187.8 GOP/s and 137.0 GOP/s under 150MHz working frequency.

10. REFERENCES

[1] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," pp. 211–252, 2015.

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

[3] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," in *ECCV*, 2014, pp. 818–833.

[4] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," *arXiv preprint arXiv:1409.4842*, 2014.

[5] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *arXiv preprint arXiv:1512.03385*, 2015.

[6] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of ISFPGA*. ACM, 2015, pp. 161–170.

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

[8] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun *et al.*, "Dadiannao: A machine-learning supercomputer," in *MICRO*. IEEE, 2014, pp. 609–622.

[9] D. Liu, T. Chen, S. Liu, J. Zhou, S. Zhou, O. Teman, X. Feng, X. Zhou, and Y. Chen, "Pudiannao: A polyvalent machine learning accelerator," in *ASPLOS*. ACM, 2015, pp. 369–381.

[10] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: shifting vision processing closer to the sensor," in *ISCA*. ACM, 2015, pp. 92–104.

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

[12] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun, "Neuflow: A runtime reconfigurable dataflow processor for vision," in *CVPRW*. IEEE, 2011, pp. 109–116.

[13] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun, "Cnp: An fpga-based processor for convolutional networks," in *FPL*. IEEE, 2009, pp. 32–37.

[14] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[15] A. S. Razavian, H. Azizpour, J. Sullivan, and S. Carlsson, "Cnn features off-the-shelf: an astounding baseline for recognition," in *CVPRW*. IEEE, 2014, pp. 512–519.

[16] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[17] S. J. Hanson and L. Y. Pratt, "Comparing biases for minimal network construction with back-propagation," in *NIPS*, 1989, pp. 177–185.

[18] Y. LeCun, J. S. Denker, S. A. Solla, R. E. Howard, and L. D. Jackel, "Optimal brain damage," in *NIPS*, vol. 89, 1989.

[19] B. Hassibi and D. G. Stork, *Second order derivatives for network pruning: Optimal brain surgeon*. Morgan Kaufmann, 1993.

[20] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," *arXiv preprint arXiv:1506.02626*, 2015.

[21] G. H. Golub and C. F. Van Loan, "Matrix computations. 1996," *Johns Hopkins University, Press, Baltimore, MD, USA*, pp. 374–426, 1996.

[22] E. L. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus, "Exploiting linear structure within convolutional networks for efficient evaluation," in *NIPS*, 2014, pp. 1269–1277.

[23] X. Zhang, J. Zou, X. Ming, K. He, and J. Sun, "Efficient and accurate approximations of nonlinear convolutional networks," *arXiv preprint arXiv:1411.4229*, 2014.

[24] M. Jaderberg, A. Vedaldi, and A. Zisserman, "Speeding up convolutional neural networks with low rank expansions," *arXiv preprint arXiv:1405.3866*, 2014.

[25] C. Farabet, Y. LeCun, K. Kavukcuoglu, E. Culurciello, B. Martini, P. Akselrod, and S. Talay, "Large-scale fpga-based convolutional networks," *Machine Learning on Very Large Data Sets*, vol. 1, 2011.

[26] M. Sankaradas, V. Jakkula, S. Cadambi, S. Chakradhar, I. Durdanovic, E. Cosatto, and H. Graf, "A massively parallel coprocessor for convolutional neural networks," in *ASAP*, July 2009, pp. 53–60.

[27] D. Larkin, A. Kinane, and N. O'Connor, "Towards hardware acceleration of neuroevolution for multimedia processing applications on mobile devices," in *Neural Information Processing*. Springer, 2006, pp. 1178–1188.

[28] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[29] B. Bosi, G. Bois, and Y. Savaria, "Reconfigurable pipelined 2-d convolvers for fast digital signal processing," *VLSI*, vol. 7, no. 3, pp. 299–308, 1999.

[30] V. Gokhale, J. Jin, A. Dundar, B. Martini, and E. Culurciello, "A 240 g-ops/s mobile coprocessor for deep neural networks," in *CVPRW*. IEEE, 2014, pp. 696–701.