A High Performance FPGA-based Accelerator for Large-Scale Convolutional Neural Networks

Huimin Li, Xitian Fan, Li Jiao, Wei Cao, Xuegong Zhou, Lingli Wang*
State Key Laboratory of ASIC and System
Fudan University
Shanghai, China
*caow@fudan.edu.cn

***Abstract*—In recent years, convolutional neural networks (CNNs) based machine learning algorithms have been widely applied in computer vision applications. However, for large-scale CNNs, the computation-intensive, memory-intensive and resource-consuming features have brought many challenges to CNN implementations. This work proposes an end-to-end FPGA-based CNN accelerator with all the layers mapped on one chip so that different layers can work concurrently in a pipelined structure to increase the throughput. A methodology which can find the optimized parallelism strategy for each layer is proposed to achieve high throughput and high resource utilization. In addition, a batch-based computing method is implemented and applied on fully connected layers (FC layers) to increase the memory bandwidth utilization due to the memory-intensive feature. Further, by applying two different computing patterns on FC layers, the required on-chip buffers can be reduced significantly. As a case study, a state-of-the-art large-scale CNN, AlexNet, is implemented on Xilinx VC709. It can achieve a peak performance of 565.94 GOP/s and 391 FPS under 156MHz clock frequency which outperforms previous approaches.**

Keywords—convolutional neural networks; pipeline; AlexNet; parallelism; memory bandwidth

I. INTRODUCTION

In recent years, there has been a growing interest in the study of convolutional neural networks (CNNs), inspired by the nervous system in human brain. Owing to the high accuracy and good performance, CNNs have been widely adopted in many applications such as image classification [1, 2], face recognition [3, 4], digital video surveillance [5] and speech recognition [6], etc.

As CNN models become larger and deeper, numerous operations and data accesses are demanded for CNN-based implementations. Hence, general purpose processors are not so efficient while a number of CNN implementations based on GPU [1], FPGA [7, 8, 9, 10, 11, 12, 13, 14, 15] even ASIC [16, 17, 18, 19] have been proposed. Among these platforms, FPGA is especially suitable as the hardware accelerator owing to its flexibility and efficiency. However, there are still many challenges. Since CNNs are known for their computation complexity and frequent data access, previous work mainly focuses on two aspects: maximizing the parallelism for computation and reducing required memory bandwidth. In the existing implementations [7, 8, 9, 10, 17], the same parallelism strategy is used to implement different layers of a CNN model. Consequently, the single parallelism strategy can't meet the requirements for different layers, which reduces the overall performance. Moreover, only one layer is working at one time instead of all the layers working concurrently in a pipelined style. Therefore, there is still a large space to improve the overall performance. With the development of FPGA technology, the on-chip resources have been increased significantly. It is possible to map all the layers onto one chip and adopt different parallelism strategies for different layers, which can achieve high performance and high resource utilization. In addition, the limited memory bandwidth is a bottleneck, especially for fully connected layers (FC layers). To overcome this problem, weights are stored in the on-chip memory to reduce the data accesses in [17, 18]. However, for large-scale CNNs, it is infeasible because too many on-chip buffers are required. Thus, we optimize the data access of FC layers in this work.

The key contributions of this work are as follows:

- An end-to-end high performance CNN accelerator is proposed with all the layers working concurrently in a pipelined style.
- A methodology is proposed on how to determine the optimized parallelism strategy for each layer to achieve high throughput and high resource utilization.
- As the convolutional layers (CONV layers) are computation-intensive while FC layers are memory-intensive [9], different computing methods are employed. In FC layers, a batch-based computing method is used to reduce the required memory bandwidth for weights. Further, two computing patterns are applied on FC layers alternately, which can reduce the required on-chip buffers in FC layers significantly.
- A state-of-the-art large-scale CNN, AlexNet [1], is implemented on VC709 FPGA board with the proposed architecture and methodology. It can achieve a peak performance with 391 FPS and 565.94 GOP/s at a 156MHz clock which outperforms previous approaches.

The rest of paper is organized as follows. Section II describes the related work. In Section III, a brief introduction to CNNs and a CNN model AlexNet are given. Section IV explores the design space and proposes a methodology to find the optimized parallelism strategy along with an optimization method for data access. Section V describes our implementation details. A

comparison is made between our design and previous implementations in Section VI. Section VII concludes the paper.

II. Related Work

To design CNN accelerators, researchers mainly focus on two aspects. One is to reduce the required memory bandwidth. The other is to explore the parallelism space and increase the throughput.

During data processing, numerous input data and weights need to be read, which demands a high required bandwidth. References [17] and [18] have stored the weights on chip to reduce the required bandwidth. However, as CNN models become larger, it is infeasible to store all the weights on chip. If all the weights are stored off chip, then a high memory bandwidth will be required. By delicate data reuse or data precision reduction, the bandwidth utilization can be increased in [7, 9, 10, 16, 11, 12]. Moreover, reference [9] has shown that FC layers are memory-intensive, so Singular Value Decomposition (SVD) is applied on the weight matrix. In our design, on-chip buffers are used to store the intermediate data between layers while the weights are stored off chip. In addition, we apply a batch-based computing method on FC layers to reduce the required bandwidth, which is based on the batch processing in [1]. Thus, multiple input images can be processed in parallel as a batch with the same weights.

In order to determine the proper parallelism strategy and optimize the computation, much work has been done such as [7, 8, 10, 12, 13]. References [12] and [13] use parallelism in feature maps and convolution operations. This work adopts 3 types of parallelism as proposed in [8]: intra-output parallelism, inter-output parallelism and parallelism within a convolution operation. However, [8, 12, 13] haven't optimized the memory bandwidth. References [7] and [10] have optimized both the bandwidth and the computation, which achieve a good balance. Even so, due to the on-chip resource limitation, different layers can't work concurrently in these approaches. Moreover, a single parallelism strategy is adopted for different layers. As the on-chip resources have been increased significantly, this work employs a pipelined structure, so that different layers can work simultaneously on one FPGA chip to increase the throughput remarkably. Furthermore, different parallelism strategies are used in different layers to meet their requirements.

III. Background

In this section, a typical CNN structure is briefly introduced. Then the state-of-the-art CNN model AlexNet and some statistical information are presented for the architecture design.

A. Primer on the CNN Structure

A typical CNN model is composed of several layers. In each layer, there are numerous neurons connected with the neurons in the adjacent layers. When an input is given, it passes through layers to generate a feature vector. After that, a classifier is applied on the generated feature vector to produce the classification result. There are mainly three types of layers in a CNN model during the feed-forward computation, CONV layers, pooling layers (POOL layers) and FC layers.

CONV layers are the core building blocks in the CNN model. A CONV layer accepts several input feature maps. For each input map, a convolution kernel is applied to generate a temporary output map. All the temporary maps and a bias are added to generate a final output map. Next, a nonlinear activation function will be applied on every neuron of the output map. In AlexNet, Rectified Linear Unit (ReLU) is adopted as the activation function as in (1). Other output maps can be generated in the same way. The operation in a CONV layer is expressed in (2). Here, IN^i and OUT^j represent the *i*-th input feature map and the *j*-th output feature map respectively; W^{ij} is the convolution kernel applied on the *i*-th input map to generate the *j*-th output map; N_{in} is the number of input maps in this layer; $bias^j$ is the bias for the *j*-th output map.

$$ReLU(x) = max(x, 0) \tag{1}$$

$$OUT^j = \sum_{i=1}^{N_{in}} IN^i \otimes W^{ij} + bias^j \tag{2}$$

Pooling layers usually follow the CONV layers. Pooling is a form of non-linear sub-sampling which can bring translation invariance and avoid over-fitting. After a pooling operation, the size of the input map is reduced because the neighboring neurons in a sliding window are merged into one single neuron. The commonly used pooling methods include max pooling and average pooling. In AlexNet model, max pooling is adopted.

FC layers are usually in the posterior of a CNN model. In an FC layer, each neuron is connected with all the neurons in the previous layer, so the number of weights in FC layers is very large. Equation (3) shows the operation in an FC layer. Here, IN^i and OUT^j are the *i*-th neuron of the input vector and the *j*-th neuron of the output vector respectively; w^{ij} is the weight; N_{in} is the number of the input neurons. For each neuron of the output vector, there is a corresponding bias $bias^j$.

$$OUT^j = \sum_{i=1}^{N_{in}} IN^i \times w^{ij} + bias^j \tag{3}$$

B. State-of-the-art CNN Model AlexNet

Nowadays, several large-scale state-of-the-art CNN models have been proposed with high classification accuracy. For instance, AlexNet is proposed in the 2012 Image-Net Large-Scale Vision Recognition Challenge (ILSVRC) [20] with top-1 and top-5 classification error rates of 37.5% and 17% on ImageNet dataset [1]. Fig.1 shows the AlexNet model which includes 5 CONV layers and 3 FC layers. Three max pooling layers follow the first, the second and the fifth CONV layers. ReLU function is applied on every layer except the last one. During the computation, it has been divided into two groups. The configurations of AlexNet are in Table I, where N_{in} and N_{out} are the numbers of input maps and output maps; $Size_{in}$ and $Size_{out}$ are the sizes of the input map and output map; $Size_{kernel}$ is the size for the convolution kernel; $stride$ is the shifting stride of the kernel during convolution operations.

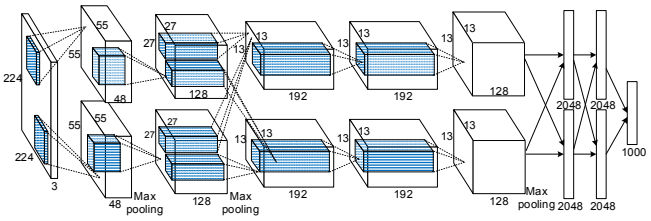


Fig.1. An overview of the large-scale CNN model AlexNet.

TABLE I. CONFIGURATIONS OF DIFFERENT LAYERS IN ALEXNET

Layer	N_{in}	N_{out}	$Size_{in}$	$Size_{out}$	$Size_{kernel}$	Stride
CONV1	3	96	227x227	55x55	11x11	4
POOL1	96	96	55x55	27x27	3x3	2
CONV2	48	256	27x27	27x27	5x5	1
POOL2	256	256	27x27	13x13	3x3	2
CONV3	256	384	13x13	13x13	3x3	1
CONV4	192	384	13x13	13x13	3x3	1
CONV5	192	256	13x13	13x13	3x3	1
POOL5	256	256	13x13	6x6	3x3	2
FC6	9216	4096	1x1	1x1	--	--
FC7	4096	4096	1x1	1x1	--	--
FC8	4096	1000	1x1	1x1	--	--

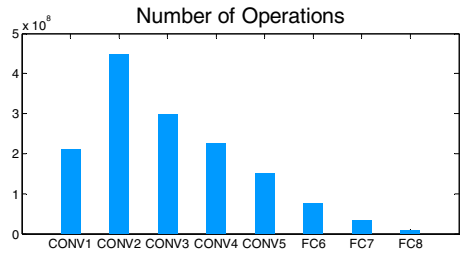


Fig.2. Number of operations for different layers in AlexNet.

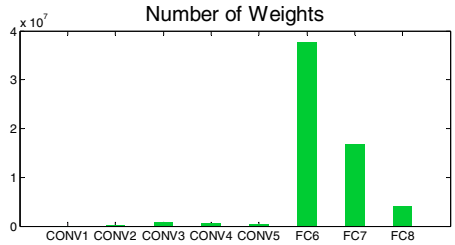


Fig.3. Number of weights for different layers in AlexNet.

Reference [9] has shown that in the VGG model [21], CONV layers are computation-intensive while FC layers are memory-intensive. Thus, the operation number and weight number for each layer in the AlexNet model are collected as shown in Fig.2 and Fig.3, which reflects the computation-intensive feature of CONV layers and the memory-intensive feature of FC layers. On this account, different methods are used to implement CONV layers and FC layers. For CONV layers, a higher parallel degree and more resources are used than FC layers. For FC layers, we focus on reducing the required memory bandwidth. The pooling layers are usually included in the large CONV layers in the architecture design.

IV. EXPLORATION AND OPTIMIZATION

In this section, the overall architecture of our accelerator is presented firstly. Secondly, the basic computation modules for different types of layers are introduced. Thirdly, we explore the design space and discuss how to find the optimized parallelism strategy for each layer. Finally, an optimization method for FC layers is presented to reduce the required bandwidth.

A. Accelerator Architecture Overview

As described in the previous section, in the existing FPGA-based CNN implementations, there is still a large space to improve the overall performance. To further improve the performance and increase the throughput, a pipelined structure can be applied on the CNN model with different parallelism strategies for different layers. Fig.4 is the overall architecture of our accelerator, taken AlexNet as an example.

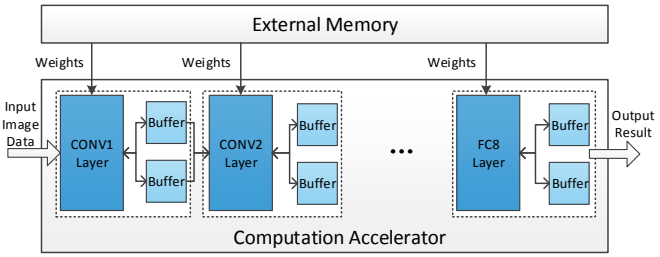


Fig.4. An overview of the accelerator architecture.

In the architecture, the 5 CONV layers and 3 FC layers run in the 8 modules concurrently in a pipelined style. Consequently, numerous input data and weights are required, which leads to a high data access workload. To solve this problem, intermediate results between layers are stored in on-chip buffers. In this way, the data access workload can be reduced significantly. Weights are stored in the external memory because the number of weights is too large. Furthermore, on-chip buffers also facilitate data reuse. There are 2 ping-pong buffers for each stage, where the former layer may write to or read from one buffer while the next layer reads data from the other buffer. Dual-port memory blocks are used as the ping-pong buffers. Hence, ping-pong buffers can avoid the overlap and improve the performance.

B. Basic Computation Modules

To implement a **CONV layer**, various structures have been developed as listed in [17], where the input data propagate in a systolic style in most of the structures. This work adopts a 1-dimensional processing element (1-D PE) structure in Fig.5, where 3×3 kernel size and stride 1 are used as an example.

Fig.5 shows the 1-D PE for applying 3×3 convolution operations on a 5×5 input map. The input data flow into the PE row by row as shown by the arrows in the input map. A 1-D PE is mainly composed of three accumulators and three multipliers along with three multiplexers to choose the required weights. The computation process is as follows. First, the input data pass through three shift registers. When the first input x_{11} arrives at the input of the rightmost multiplier, the 3 multipliers start calculating. After 3 clock cycles, the rightmost multiplier generates a temporary result of the first convolution window with value $x_{11} \times w_{11} + x_{12} \times w_{12} + x_{13} \times w_{13}$, while the other two multipliers generate the temporary results of y_{12} and y_{13}. When the first three rows finish calculating, the final results of y_{11}, y_{12} and y_{13} are generated. Other results in the output map are generated in the same way. As there are overlaps between sliding windows, all the rows should be read for 3 times except the 2 rows in the top and the bottom of the input map. If a higher speed up ratio is required, the 1-D PE can be extended into a 2-D PE, as illustrated in Fig.6. A 2-D PE is composed of multiple 1-D PEs. During computing, the input data is broadcast to these 1-D PEs. Thus, each row of the input map can be calculated in multiple 1-D PEs concurrently by applying different weights. In this way, each row needs to be read for only once. Hence, the speed up ratio can be increased by three times compared to 1-D PE in this case.

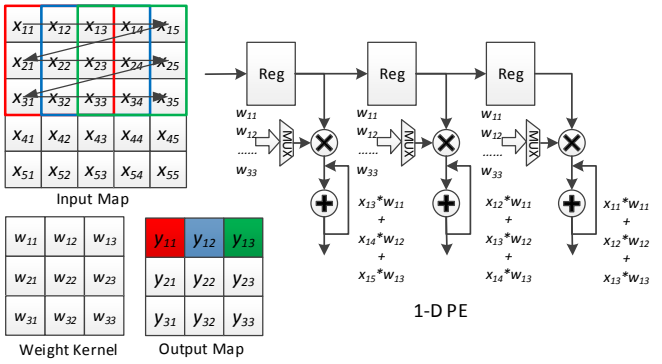


Fig.5. The 1-dimensional Processing Element (1-D PE) to implement convolution operations with a 3 × 3 weight kernel and a 5 × 5 input map.

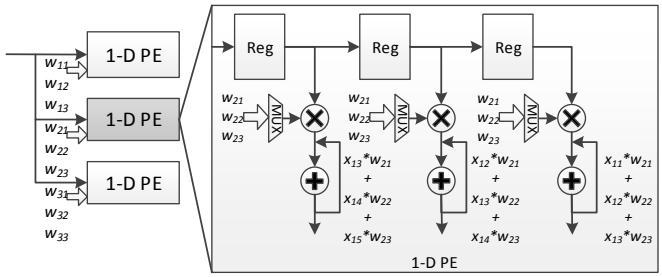


Fig.6. Architecture of the 2-dimensional Processing Element (2-D PE).

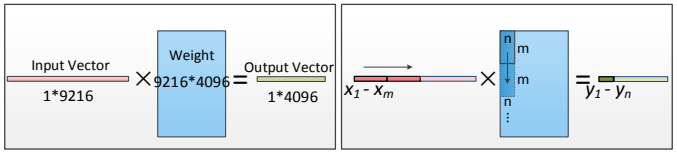
This structure is generally applicable for any convolution operations. The numbers of the used multipliers, accumulators and multiplexers change with the kernel size and shifting stride. Given the kernel size $Size_{kernel}$ and the corresponding shifting stride $stride$, the numbers of multipliers, accumulators and multiplexers required for 1-D and 2-D PE are in (4) and (5).

$$Num_{1-D} = Size_{kernel}/stride \tag{4}$$

$$Num_{2-D} = {Num_{1-D}}^2 \tag{5}$$

For a **pooling layer**, the architecture is similar with the CONV layer, only the convolution operations should be replaced by the max or average operations.

FC layer can be regarded as matrix multiplication. The FC6 layer in AlexNet is taken as an example to illustrate the computing pattern, where there are a 9216-dimensional input feature vector and a 4096-dimensional output feature vector. The weights are expressed as a 9216 × 4096 matrix in Fig.7 (a). To implement such a large-scale matrix multiplication with limited hardware resources, we divide it into multiple small-scale matrix multiplications as shown in Fig.7 (b). Firstly, x_1 to x_m are taken as the input vector to generate the temporary results of y_1 to y_n by applying $m \times n$ weights. Then we take x_{m+1} to x_{2m} with another $m \times n$ weights as the input to update the



(a) Basic operation in an FC layer (b) multiple small-scale matrix multiplications

Fig.7. Matrix multiplication in an FC layer, taken the FC6 layer in AlexNet model as an example.

temporary results of y_1 to y_n. After the 9216 input data and the first n rows in the weight matrix have been traversed, the final results of y_1 to y_n can be generated. The other output results can be generated in the same way.

C. Parallelism Space Exploration

As CNN models become larger and deeper, various parallelism strategies have emerged with different performance and resource utilization. Thus, it is necessary to explore the parallelism space. This work adopts 3 types of parallelism to implement a CNN layer as proposed in [8]: parallelism within a convolution operation, parallelism among multiple input maps and parallelism among multiple output maps. The first type of parallelism is exhibited in the basic computation modules as in 1-D PE and 2-D PE, where the parallel degrees are Num_{1-D} and Num_{2-D} respectively. Thus, we mainly discuss the other two types. To determine the parallelism strategy for a layer means to find the input map parallelism and output map parallelism. There are many factors to consider, such as on-chip resource limitation and the data dependency between layers. In addition, to maximize the throughput in a pipelined structure, the time cost on each pipeline stage should be approximately the same.

In this design, 1-D PE is adopted as the basic computation module in CONV layers, so the parallelism inside a convolution operation is $Size_{kernel}/stride$. Moreover, the input map should be read for $Size_{kernel}/stride$ times to apply one convolution kernel. $Para_{in}$ and $Para_{out}$ are used to denote the input map parallelism and the output map parallelism. Given a clock frequency f, the product of $Para_{in}^i$ and $Para_{out}^i$ for layer i can be expressed in (6), which is the product of the computation time for a single input map, the number of input maps N_{in}^i and the number of output maps N_{out}^i in the i-th layer. In (6), t_s represents the time used in each pipeline stage; $Size_{in}^i$ is the size of the input map. For an FC layer, N_{in} and N_{out} can be regarded as the input number and output number of neurons respectively; $Size_{in}$, $Size_{kernel}$ and $stride$ equal to 1. Our target is to determine the proper $Para_{in}$ and $Para_{out}$ for each layer.

$$Para_{in}^i \times Para_{out}^i = \frac{\frac{1}{f} \times \frac{Size_{kernel}^i}{stride^i} \times \left(Size_{in}^i\right)^2}{t_s} \times N_{in}^i \times N_{out}^i \tag{6}$$

Since DSP blocks in FPGA are used to implement multiplication operations, the total number of multipliers should be subject to the DSP block resources on chip as in (7). N_{layer} is the number of layers in a CNN model; DSP_{total} is the total number of DSPs on an FPGA chip. In addition, as on-chip Block RAMs (BRAMs) are used to store intermediate results between layers, the on-chip memory resources should be considered. Firstly, the BRAMs used for one layer should be enough to store the output maps of this layer as described in (8),where N_{BRAM}^i is the number of BRAMs used in the i-th layer; $C_{oneBRAM}$ is the storage capacity of one BRAM. Secondly, N_{BRAM}^i should be larger than the input parallelism of the next layer and the output parallelism of this layer as in (9) and (10). Thus, multiple data can be written into or read from the BRAMs simultaneously. Thirdly, the BRAMs used in all layers should be subject to the on-chip BRAM resources $BRAM_{total}$ as in (11). Besides (7) to (11), there are some other constraints we need to pay attention to in practice. One is that $Para_{in}^i$ and $Para_{out}^i$ should be a

divisor of N_{in}^{i} and N_{out}^{i} respectively. Moreover, N_{BRAM}^{i} should be a multiple of $Para_{in}^{i+1}$ and $Para_{out}^{i}$. These two constraints can reduce the control complexity significantly.

$$\sum_{i=1}^{N_{layer}} Para_{in}^{i} \times Para_{out}^{i} \times \frac{Size_{kernel}^{i}}{stride^{i}} \leq DSP_{total} \quad (7)$$

$$N_{BRAM}^{i} \geq \frac{N_{out}^{i} \times (Size_{out}^{i})^{2}}{C_{oneBRAM}} \quad (8)$$

$$N_{BRAM}^{i} \geq Para_{in}^{i+1} \quad (9)$$

$$N_{BRAM}^{i} \geq Para_{out}^{i} \quad (10)$$

$$\sum_{i=1}^{N_{layer}} N_{BRAM}^{i} \leq BRAM_{total} \quad (11)$$

Based on the above analysis, a methodology to determine the optimized parallelism strategy for each layer can be proposed with the following 3 steps. Firstly, given a value for t_s, the product of $Para_{in}^{i} \times Para_{out}^{i}$ for each layer can be calculated by (6). If (7) is not satisfied, t_s should be increased until (7) is satisfied. Secondly, equation (8) is used to calculate the minimum number of BRAMs for each layer. Thirdly, equations (9), (10), (11) and the product of $Para_{in}^{i} \times Para_{out}^{i}$ are used to determine the $Para_{in}^{i}$ and $Para_{out}^{i}$ for layer i. In this step, N_{BRAM}^{i}, $Para_{in}^{i}$ and $Para_{out}^{i}$ should be adjusted to make $Para_{in}^{i}$ and $Para_{out}^{i}$ divisors of N_{in}^{i} and N_{out}^{i}. In addition, $Para_{in}^{i+1}$ and $Para_{out}^{i}$ should be the divisors of N_{BRAM}^{i}. In this way, the control complexity can be reduced significantly.

D. Optimization for Memory Bandwidth

During computing, 8 layers read weights from the off-chip memory simultaneously, where high memory bandwidth is still required. Fig.3 has shown that FC layers contribute to most of the data accesses. Therefore, a batch-based computing method for FC layers is employed. Fig.8 illustrates the scenario in which multiple input feature vectors are computed as a batch in FC layers. In Fig.8, N is the number of input feature vectors, which can be regarded as the batch size. As a result, the operation amount can be increased by N times without increasing the number of data accesses for weights. Hence, the required memory bandwidth can be reduced.

In Fig.9, the input parallelism and output parallelism are represented by m and n respectively. With $m \times n$ DSP blocks, $m \times n$ multiplications can operate concurrently, where $m \times n$ weights are required. Thus, it takes N clock cycles to operate $N \times m \times n$ multiplications. Moreover, during the N clock cycles, the next $m \times n$ weights need to be ready for the next $N \times m \times n$ multiplications. Therefore, N should be no less than the number of required clock cycles to read $m \times n$ weights, as in (12). In this way, the FC layers can compute continuously without waiting for weights. In (12), N^{i} denotes the batch size for the i-th FC layer; $Bitwidth_{para}$ is the bit width of the weight; $Bandwidth^{i}$ is the required memory bandwidth for the i-th layer. The batch size N should be subject to the memory bandwidth provided by hardware which is denoted by $Bandwidth_{total}$, as shown in (13). For CONV layers and FC layers, the required memory bandwidth can be calculated by the equations in (14) and (15). Batch-based computing reduces data access, but sacrifices the latency for the first output result.

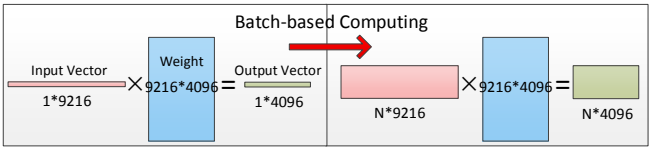


Fig.8. Batch-based computing method in the FC layer.

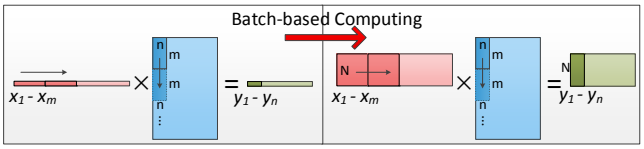


Fig.9. One computing pattern for the FC layer; the weight window in the weight matrix shifts vertically.

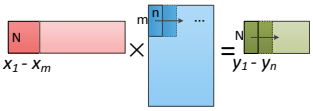


Fig.10. Another computing pattern in the FC layer; the weight window in the weight matrix shifts horizontally

$$N^{i} \geq \frac{m^{i} \times n^{i} \times Bitwidth_{para}}{Bandwidth^{i}/f} \quad (12)$$

$$\sum_{i=1}^{N_{layer}} Bandwidth^{i} \leq Bandwidth_{total} \quad (13)$$

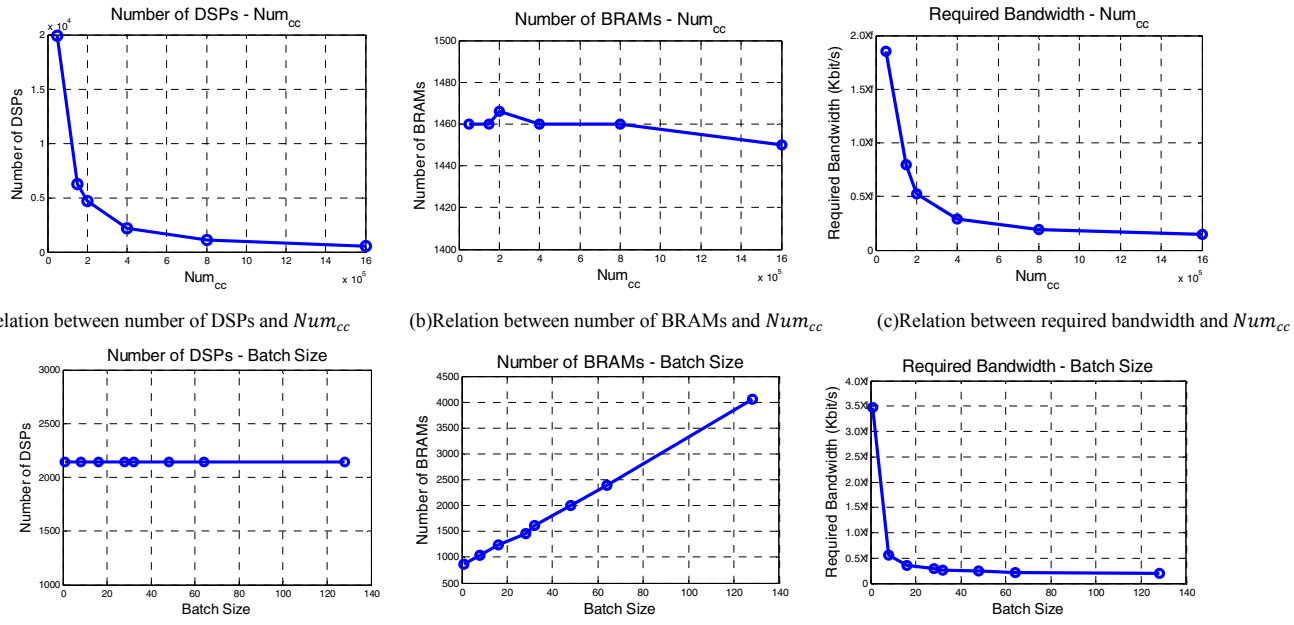
$$\frac{Bandwidth_{Conv}^{i}}{f} = \frac{Para_{in}^{i} \times Para_{out}^{i} \times (Size_{kernel}^{i})^{2} \times Bitwidth_{para}}{(Size_{in}^{i})^{2} \times Size_{kernel}^{i}/stride^{i}} \quad (14)$$

$$\frac{Bandwidth_{FC}^{i}}{f} = \frac{Para_{in}^{i} \times Para_{out}^{i} \times Bitwidth_{para}}{N} \quad (15)$$

The batch-based computing method can bring another issue because the required on-chip buffers for FC layers are expanded by N times. To address this issue, the computing pattern of FC layers is rearranged to reduce the buffer requirement. Besides the computing pattern in Fig.9, another pattern is employed as in Fig.10. In this pattern, the temporary results of all the output neurons are generated first by using x_1 to x_m only as the input. Next, the temporary results of all the output neurons are updated by taking x_{m+1} to x_{2m} as the input. The window with size $m \times n$ in the weight matrix shifts horizontally which distinguishes the vertically shifting pattern in Fig.9. In this work, these two patterns are both adopted to implement FC layers by applying them alternately on FC layers. For the first FC layer, the vertically shifting pattern is used to generate y_1 to y_n. Thus, the second FC layer can start with y_1 to y_n as the input by applying the horizontally shifting pattern. Since the second FC layer starts before the whole input vector is generated from the first FC layer, a few BRAMs are enough to store the $N \times n$ neurons. However, in the second FC layer, a number of BRAMs are still required to store the temporary results of all the output neurons. For the third FC layer, the vertically shifting pattern is applied as in the first FC layer. In this way, the required buffers can be reduced remarkably.

E. Implementation of the Large-Scale AlexNet

As a case study, the proposed architecture and methodology are used to implement the CNN model, AlexNet, where 16-bit fixed point data are adopted for weights, input data and the intermediate results between layers. In addition, 18Kb BRAM is used as the basic memory block for analysis.



(a)Relation between number of DSPs and Num_{cc} (b)Relation between number of BRAMs and Num_{cc} (c)Relation between required bandwidth and Num_{cc}

(d)Relation between number of DSPs and batch size (e)Relation between number of BRAMs and batch size (f)Relation between required bandwidth and batch size

Fig.11. An analysis on the relations between number of DSPs, number of BRAMs , batch size, required bandwidth and Num_{cc} based on AlexNet model. In (a-c), batch size N is the controlling variable with value 28 while Num_{cc} changes; in (d-f), Num_{cc} is the controlling variable with value 400,000 while N changes

Based on the AlexNet model, the relations between the number of DSPs, BRAMs, the required memory bandwidth and the number of clock cycles in one pipeline stage are evaluated as in Fig.11 (a) - (c). Here, Num_{cc} is the number of clock cycles in a pipeline stage, which equals to the value of $t_s \times f$. t_s is the time cost on each pipeline stage and f is the clock frequency. The batch size N is set as the controlling variable with a value 28 while the value of Num_{cc} changes. Furthermore, the relations between the number of used DSPs, the number of used BRAMs, the required bandwidth and the batch size N are also evaluated, which is described in Fig.11 (d) - (f). In this evaluation, Num_{cc} is set as the controlling variable with value 400,000 while the batch size N changes.

From Fig.11 (b), we can find that when the batch size is fixed, there is no obvious relevancy between the number of required BRAMs and the number of clock cycles in a pipeline stage. However, if Num_{cc} becomes smaller, which means higher parallel degree is required, more DSPs are employed along with a higher required memory bandwidth to support the data accesses for weights as shown in Fig.11 (a, c). The number of DSPs is not affected by the batch size, while the number of BRAMs increases linearly with the growth of the batch size as in Fig.11 (d, e). In addition, in Fig.11 (f), the bandwidth can be reduced significantly by applying the batch-based computing method. Nonetheless, when N is larger than 30, there is no more obvious improvement for data access while the number of used BRAMs still keeps increasing. Hence, the batch size is not necessary to be larger than 30 in AlexNet model implementation.

Based on the above analysis, the large-scale CNN model AlexNet is implemented with batch size 28. In this design, 400,000 is adopted as the value of Num_{cc}. If a smaller Num_{cc} is adopted, a higher throughput and resource utilization can be achieved. However, a larger memory bandwidth will be demanded. For this reason, 400,000 is adopted to balance the data access workload and the throughput along with the resource utilization. Moreover, 1-D PE is employed to implement the convolution operations in these 5 CONV layers. The input parallelism, the output parallelism and the number of used BRAMs for each layer are listed in Table II. In layer CONV2, $132\ (4 + 64 \times 2)$ BRAMs are used because extra 4 BRAMs are required to store the intermediate results before pooling. In layer CONV5, extra 64 BRAMs are needed for the same reason as in CONV2. However, no more BRAMs are needed to store the results before pooling in CONV1 because the input parallelism $Para_{in}$ equals to the number of input maps in this layer. By applying these two computing patterns presented in Section IV-D, the numbers of BRAMs used in layer FC6 and layer FC8 can be reduced remarkably. In this way, the total number of required BRAMs for the 3 FC layers are 244. If one computing pattern is applied only, 504 BRAMs will be needed. Hence, the number of BRAMs is reduced by 55.6% in FC layers. Moreover, by applying the proposed batch-based computing method with batch size 28, the total required memory bandwidth to read weights is reduced to $288 \times f$ bits per second, while if the batch-based method is not used, $3472 \times f$ bits per second are required.

TABLE II. CONFIGURATION FOR ALEXNET HARDWARE IMPLEMENTATION

Layer	$Para_{in}$	$Para_{out}$	*Number Of BRAMs*	*Size of 1-D PE*	*Number of DSPs*	*Required bandwidth (bit/s)*
CONV1	3	48	$192(96 \times 2)$	3	432	$16 \times f$
CONV2	32	4	$132(4 + 64 \times 2)$	5	640	$16 \times f$
CONV3	32	4	$128(64 \times 2)$	3	384	$48 \times f$
CONV4	32	3	$192(96 \times 2)$	3	288	$32 \times f$
CONV5	32	2	576 $(64 + 256 \times 2)$	3	192	$32 \times f$
FC6	16	8	$16(8 \times 2)$	--	128	$64 \times f$
FC7	8	8	$224(112 \times 2)$	--	64	$64 \times f$
FC8	8	2	$4\ (2 \times 2)$	--	16	$16 \times f$

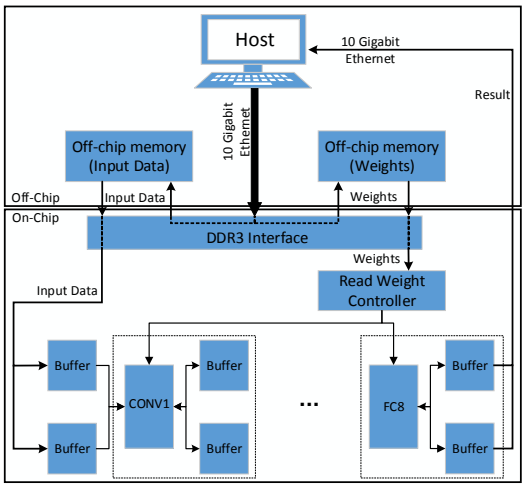


Fig.12. System Architecture.

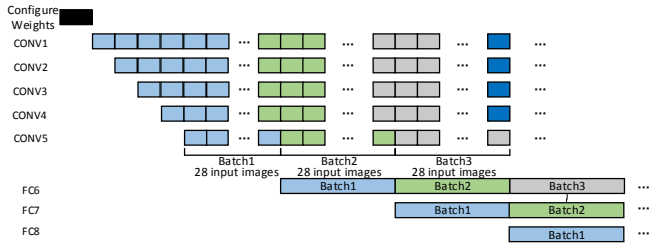


Fig.13. Execution time of each layer working in a pipelined style.

V. Implementation Details

A. System Architecture

As illustrated in Fig.12, the whole system fits in one FPGA chip along with two 4GB DDR3 external memory blocks with bandwidth 14.9 GB/s to store the input data and the weights. The host sends the weights first to the DDR3 interface through 10 Gigabit Ethernet. Then the weights are stored in one of the DDR3 blocks. Next, the host sends the input image data which is stored in another DDR3 block. This DDR3 acts as a large buffer, so that the processing speed of the accelerator can match the data sending speed. In the accelerator, 8 layers are implemented by 8 modules where 2 ping-pong buffers are utilized to store the intermediate data. During computing, these 8 layers request for weights to the "Read Weight Controller". Then, the controller makes an arbitration within the 8 request signals and read weights from the external memory. Here, 156MHz clock is used for the accelerator and the 10 Gigabit Ethernet. Given the Num_{cc} with value 400,000, which has been mentioned in Section IV-E, the time cost in each pipeline stage t_s is 2.56ms. In addition, as references [9, 22, 23] have shown that low precision quantization is sufficient for CNN implementations,16-bit fixed point data are adopted for the input data, weights and the intermediate data. In Table II, it is shown that the required memory bandwidth for weights is $288 \times f$ bit/s, which is 45 Gb/s with frequency 156MHz, and can be supported by the DDR3. The proposed accelerator can receive the image data from the host and send back the final 1000-dimensional feature vector to the host which then ranks the top 5 classification results and display the results on a website as in Fig.14. ImageNet-1K is the test set in this work.

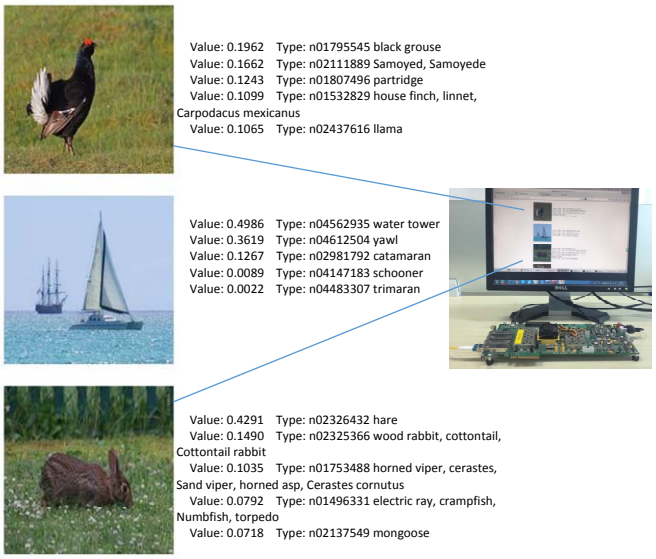


Fig.14 Implementation platform and the result display. Xilinx VC709 is used to implement the accelerator; the host sends data and receives results with the 10 gigabit Ethernet; the classification results are displayed on a website.

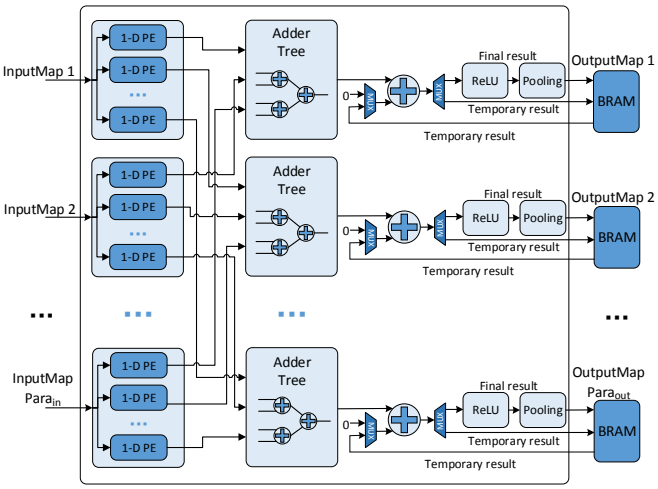


Fig.15 Architecture for a typical layer.

The execution time for each pipeline stage is described in Fig.13. Firstly, the host sends the weights to the DDR3 memory. Then the 8 layers work in the pipelined style to maximize the throughput. The batch-based computing method is employed in FC layers with batch size 28. Hence, 28 images can be executed together as a batch in FC layers.

B. Architecture for a Typical Layer

The architecture for a typical CNN layer is in Fig.15 where the input parallelism and output parallelism are $Para_{in}$ and $Para_{out}$ respectively. Therefore, $Para_{in}$ input maps are read concurrently. For each input map, it is broadcast to $Para_{out}$ 1-D PEs to generate $Para_{out}$ output maps concurrently. Then, the output data of the 1-D PEs for the same output map are added together by an adder tree. Before all the input maps are traversed, temporary results are generated from the adder tree and stored in the BRAMs. Then, the temporary results are read from the BRAMs to add the newly generated temporary results until all the input maps are traversed. Then, the pooling function and the

activation function ReLU are applied on the "final result" as in Fig.15 After that, the generated output maps will be stored in the BRAMs. For some layers, pooling and ReLU are not used, so that they can be removed from the architecture. To implement an FC layer, the 1-D PE in Fig.15 should be replaced by one single DSP block.

VI. Experimental Results

In this section, we quantitatively make a comparison between our design and some previous implementations. This design is implemented on Xilinx VC709 with Vivado (v2015.1). The resource utilization after routing is shown in Table III.

Since references [7, 10, 15] have also implemented an accelerator based on the large-scale CNN model AlexNet, a comparison is made between our work and theirs. In [10, 15], the complete AlexNet model is implemented while [7] has only implemented the 5 CONV layers. Therefore, the processing time per image of work [7] is listed in Table IV by using the time for processing the 5 CONV layers only. Furthermore, we compare our work with Caffe tool [24], which is running on Intel Xeon E5-2637 CPU (3.5GHz) and GTX TITAN X GPU. The results are included in Table IV. In our design, by using the pipelined structure among layers, the time of processing one image is 2.56 *ms* under the peak performance. Thus, the corresponding frame per second (FPS) is 391, which is 7.82x higher than work [10], 2.92x higher than work [15] and 78x higher than Caffe tool running on CPU. Although the throughput of Caffe tool running on GPU is 2x higher than our design, it consumes 8.3x more power. Reference [10] has just measured the power consumption of P395-D8 board after programming AlexNet configuration as 19.1W without measuring the power during classification, so the power for performing classification is larger than 19.1W. In this design, the power consumption is measured on FPGA board during the classification, where the peak value is 30.2W. Specifically, the energy for processing one image is compared by taking the product of the processing time per image and the power consumption. To process one image, our design consumes 4.2x lower energy than GPU, 329.9x lower energy than CPU, 2.4x lower energy than work [15] and 5x lower energy than the most recently work in [10].

Furthermore, we compare our work with some previous work on CNN implementations as illustrated in Table V. Our work achieves a high performance with 565.94 GOP/s, which significantly outperforms the previous work. In addition, this implementation can achieve the highest power efficiency with value 22.15 GOP/s/W in comparison with the previous designs.

VII. Conclusions

This work proposes an end-to-end CNN accelerator with all the layers working concurrently in a pipelined structure, which can improve the performance remarkably. In addition, a methodology is proposed to find the optimized parallelism strategy for each layer. In FC layers, a batch-based computing method is adopted to reduce the required memory bandwidth. Further, two computing patterns are applied on FC layers which can reduce the on-chip buffer requirement significantly. As a case study, a large-scale CNN model AlexNet is implemented on Xilinx VC709; it achieves 565.94 GOP/s and 391 FPS which outperforms the previous work.

Acknowledgements

This work is supported by National Natural Science Foundation of China 61131001. We also appreciate careful works and the constructive suggestions of the anonymous reviewers.

TABLE III. Resource Utilization

	Resource Utilization on Xilinx VC709			
	FF	***LUT***	***DSP48***	***BRAM(18Kb)***
Available	866400	433200	3600	2940
Utilization	262703	273805	2144	1913
Percent(%)	30.32	63.21	59.56	65.07

TABLE IV. Comparison with Other Implementations Based-on AlexNet

	Platform	***Processing time/ image(ms)***	***FPS***	***Power (W)***	***Energy/image (J)***
Caffe CPU	Intel Xeon E5-2637	195	5	130	25.4
Caffe GPU	GTX TITAN X	1.3	769	250	0.325
FPGA 2015 [7]	Virtex-7 VX485T	21.61 (5 CONV layers)	--	18.61	>0.402
FPGA 2016 [10]	Stratix-V GSD8	20.1	50	>19.1	>0.384
Microsoft Catapult [15]	Catapult Server+ Stratix V D5	7.46	134	25	0.187
This Work	Virtex-7 VC709	2.56	391	30.2	0.077

TABLE V. Comparison with Previous CNN Implementations

	ACM2010[8]	***FPGA2015[7]***	***FPGA2016[10]***	***FPGA2016[9]***	***This Work***
Platform	Virtex-5 SX240T	Virtex-7 VX485T	Stratix-V GSD8	Zynq XC7Z045	Virtex-7 VC709
Clock (MHz)	120	100	120	150	156
Precision	48-bit fixed	32-bit float	(8-16)-bit fixed	16-bit fixed	16-bit fixed
CNN Size (GOP)	0.52	1.33	30.9	30.76	1.45
Performance (GOP/s)	16	61.62	117.8	136.97	565.94
Power (W)	14	18.61	25.8	9.63	30.2
Power Efficiency (GOP/s/W)	1.14	3.31	4.57	14.22	22.15

REFERENCES

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in Neural Information Processing Systems, 2012, 25 (2), pp. 1097-1105.

[2] L. Yan, "Gradient-based learning applied to document recognition," in Proceedings of the IEEE, 2010.

[3] Y. Sun and X. Wang, "Deep Learning Face Representation from Predicting 10,000 Classes," in Computer Vision and Pattern Recognition (CVPR), 2014, pp. 1891-1898.

[4] Y. Sun, Yi, X. Wang, and X. Tang, "Deep Learning Face Representation by Joint Identification-Verification," in Neural Information Processing Systems, 2014, pp. 1988-1996.

[5] S. Ji and W. Xu, "3D Convolutional Neural Networks for Automatic Human Action Recognition," in Pattern Analysis & Machine Intelligence, 2013, 35(1), pp. 221-31.

[6] O. Abdel-Hamid, "Convolutional Neural Networks for Speech Recognition," in Audio Speech & Language Processing, 2014.

[7] C. Zhang and P. Li, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in ACM International Symposium on Field-Programmable Gate Arrays, 2015, pp. 161-170.

[8] S. Chakradhar and M. Sankaradas, "A dynamically configurable coprocessor for convolutional neural networks," in ACM Sigarch Computer Architecture News, 2010, 38(3), pp. 247-257.

[9] J. Qiu and J. Wang, "Going deeper with embedded FPGA platform for convolutional neural network," in ACM International Symposium on Field-Programmable Gate Arrays, 2016.

[10] N. Suda and V. Chandra, "Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks," in ACM International Symposium on Field-Programmable Gate Arrays, 2016.

[11] M. Peemen, "Memory-centric accelerator design for Convolutional Neural Networks," in 31st International Conference on Computer Design (ICCD), 2013, pp. 13-19.

[12] M. Sankaradas and V. Jakkula, "A Massively Parallel Coprocessor for Convolutional Neural Networks," in International Conference on Application-specific Systems, 2009, pp. 53-60.

[13] S. Cadambi, A. Majumdar, and M. Becchi, "A programmable parallel accelerator for learning and classication," in 19th International Conference on Parallel Architectures and Compilation Techniques, 2010, pp. 273-284.

[14] C. Farabet and C. Poulet, "CNP: An FPGA-based processor for Convolutional Networks," in International Conference on Field Programmable Logic & Applications, 2009, pp. 32 - 37.

[15] Microsoft Research

[16] T. Chen and Z. Du, "DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning," in ACM Sigplan Notices, 2014, 49 (4), pp. 269-284.

[17] Z. Du and R. Fasthuber, "ShiDianNao: Shifting vision processing closer to the sensor," in ACM International Symposium Computer Architecture (ISCA), 2015.

[18] Y. Chen and Y. Luo "DaDianNao: A Machine-Learning Supercomputer," in ACM International Symposium on Microarchitecture, 2014, pp. 609-622.

[19] D. Liu and T. Chen, "PuDianNao: A Polyvalent Machine Learning Accelerator," In ACM Sigplan Notices, 2015, 50 (4), pp. 369-381.

[20] O. Russakovsky and J. Deng, "ImageNet Large Scale Visual Recognition Challenge," in International Journal of Computer Vision, 2015, 115 (3), pp. 211-252.

[21] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," in Eprint Arxiv, 2014.

[22] M. Courbariaux and Y. Bengio, "Low Precision Storage for Deep Learning," in International Conference on Learning Representations (ICLR), 2015.

[23] S. Gupta and A. Agrawal, "Deep Learning with Limited Numerical Precision," arXiv preprint arXiv: 1502. 02551, 2015.

[24] Y. Jia, "Caffe: Convolutional Architecture for Fast Feature Embedding," in Eprint Arxiv, 2014, pp. 675-678.