Quantization-based Optimization of CNN Inference on a multi-FPGA system
量子化モデルによる複数 FPGA 上での推論の最適化

37-196530

Hongyi Pan (潘　虹芸)

Supervisor: Prof. Tomohiro Kudoh (工藤知宏)

Department of Electrical Engineering and Information Systems

University of Tokyo

This dissertation is submitted for the degree of

Master of Engineering

January 28, 2020

Acknowledgements

I am very grateful to have worked with many wonderful people throughout my M.Eng study and research.

First and foremost, this work was partly supported by JST CREST Grant Number JPMJCR19K1, Japan. And I sincerely thank Professor Amano and students from Amano Lab from Keio University for their guidance and assistance.

I would like to express my sincere gratitude to my advisor Professor Tomohiro Kudoh. He gave me the chance to study at the Univ. of Tokyo and work with good people. And I admire, appreciate, and am inspired by Kudoh’s enthusiasm, dedication, and work ethic in his teaching and research. Inspired by him, I always keep in mind to work hard and never forgetting to chase the latest technology trends. Kudoh has not only introduced me to the wonderful field of engineering but has also helped me vastly improved my research ability and communication skills. In addition, he is very concerned about my living in Japan. I really thank him for his help.

I also would like to thank our collaborators from AIST, Akram Ben Ahmed, and Tsutomu Ikegami. I am always amazed at their intelligence and their advice really helped a lot to my research.

Besides, I would like to thank the following members of the Computer Network Laboratory (CNL): k.satou, i.horikoshi, mazhaoyu, j.lu, s.kikuchi, g.koujitani, k.sugiura, s.sugimura, k.tominaga, y.nakazawa, r.nakajima, y.hayashi, a.hirai, dingyepeng, weiyi, for their two-year company in my master’ s life. I am always amazed by their incred-

ible creativity. Music, mahjong, traveling...I will always remember these wonderful memories as well as how gorgeous they are.

Last but not least, I would like to express my thanks to my parents for their unconditional support. Without them, I would not achieve anything.

Abstract

With specifically designed hardware, FPGA is a promising candidate for neural network inference acceleration. However, the gap between neural network model size and FPGA on-chip resources is huge. To increase the on-chip memory, We use a multi-FPGA system. The total amount of BRAM is proportional to the number of boards, and the communication delay between FPGAs is negligible. However, even for multiple FPGAs, insufficient resources and communication delays with hosts are still problems. In this paper, we use the quantization method based on LQ-Nets proposed by the Microsoft group to reduce the required storage space and communication latency. At the same time, The tradeoff between the accuracy and resource can be achieved by changing the bit width. Besides, We proposed methods for accelerating convolution layers based on LQ-Nets. The synthesis results of the first two layers of Alexnet indicate that the BRAM usage has decreased and the performance has improved.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

Deep neural networks (DNNs) have gained prominence recently by producing state-of-art results in pattern recognition, speech synthesis, customer preference elicitation, and other machine learning tasks [6]. However, the largest CNN model for a 224×224 image classification requires up to 39 billion floating-point operations (FLOP) and more than 500MB model parameters [29]. CNN is extremely powerful but CNN-based methods are computational-intensive and resource-consuming and are hard to be realized on embedded systems. In most applications, CNNs are first trained off-line in CPU and GPU machine clusters with strong computing power and then deployed for inference tasks in data centers or an embedded environment, serving a large set of end-users and applications. For example, a pre-trained neural network model that recognizes specific thinks like dogs and cats can be deployed on thousands of servers, making inferences for billions of image recognition tasks on websites and apps.

Recently, FPGA cluster-based accelerators for machine learning inference has become a research hotspot. FPGAs (Field-programmable gate arrays) are integrated circuits that can be configured by the end-user to implement digital circuits. With a neural network-oriented hardware design, FPGAs are possible to achieve higher energy efficiency compared with CPU and GPU. But due to a large number of weights and intermediate features, most of the previous research focus on using off-chip memory which would significantly reduce efficiency by causing communication delay and overhead.

1.1 Targeting Application Scenario

In recent years, mobile smart devices have become increasingly important as a tool for entertainment, learning, news, businesses, and social networking for smarter living [3] [30]. Corresponding to the more and more powerful mobile applications, storage and computing capacity of edge devices remain limited because they are designed to be portable and are only equipped with limited hardware resource. Cloud computing is a well-accepted choice for offloading heavy computational tasks from mobile devices. However, when it comes to scenarios in which an immediate response time is critical to users, such as augmented reality, automatic drive and mobile multiplayer gaming systems, cloud computing will lose its power because of the latency.

To address the above-mentioned problem, cloudlet-based offloading has been proposed, where mobile devices offload computational process to a computing infrastructure (i.e., cloudlet) that is in relatively close proximity to the users [27]. In order to solve the Wi-Fi access problem, the researchers further put forward mobile edge computing system that enables mobile users to access IT and cloud computing services in close proximity within the range of radio access networks [21] [28] [24]. Compared with cloud system, mobile edge computing system can make the communication distance between the user and the server greatly shortened by placing the server in the base station close to the user, which can greatly reduce the communication delay as well and the burden of bandwidth. Recent work has experimentally quantified the benefits of edge computing. For instance, by placing VM-based cloudlets at the network edge to accelerate the computational engine, one can achieve lower response time by up to 4.9x compared with cloud offloading [15].

On the other side, Field Programmable Gate Array (FPGA) has been proven to be an appealing solution to accelerate compute-intensive workloads. Because a large majority of the electrical functionality inside the device can be reconfigured, FPGA is possible to achieve high efficient pipeline and parallelism design and obtain higher energy efficiency and better performance compared with CPU and GPU. Besides, because of the FPGA's controllable design and countable number of clocks, the predictable time delay can be achieved.

Motivated by the advantages of muti-access edge computing and FPGA-based accelerator, we are seeking a solution to combine these two technologies to achieve efficient responsiveness and meet the performance requirements of mobile applications. As a case study, we chose to implement ImageNet-based CNN inference for image recognition on FPGA-based server.

1.2 Contributions

- Implement Alexent inference on multi-FPGA system.
- Accelerate convolution layers based on LQ-Nets quantization method
- Increase the system throughput by adding HLS pragmas.

Chapter 2

Background

Before discussing our solution for CNN inference acceleration, let me first introduce the basic concepts of neural networks and data quantization.

2.1 Primer on Neural Network

2.1.1 General Structure

Deep Neural Network is an imitation of the information-processing paradigm in biological nervous systems. The human brain has an average of around 86 billion neurons and each neuron receives stimulus from its surrounding neurons, and when the stimulus reaches a specific threshold, it will generate an output [9]. Inspired by this, the structure of the neuron that is the basic element of CNN is shown in Fig 2.1, the neuron sums up the products of all pairs of inputs and synaptic weights and offsets with a bias to make the model more general. Activation function is introduced at the neuron output to generalize or adapt with variety of data and to differentiate between the output. There are numerous activation functions. One typical activation function is the *sigmoid* function that maps the weighted sum values from (-inf, +inf) to (0, 1)

(Figure 2.3a). Another example is the *Rectified Linear Unit (ReLU)* that clamps all negative values to 0 and retains all positive values (Figure 2.3b).

The robustness and functionality of the model can be enhanced by increasing the number of neurons and layers. The layers between the input layer and the output layer are called hidden layers since their states are usually not directly observable. When the number of layers reaches a certain level, the network can be referred as a deep neural network (Figure 2.2).

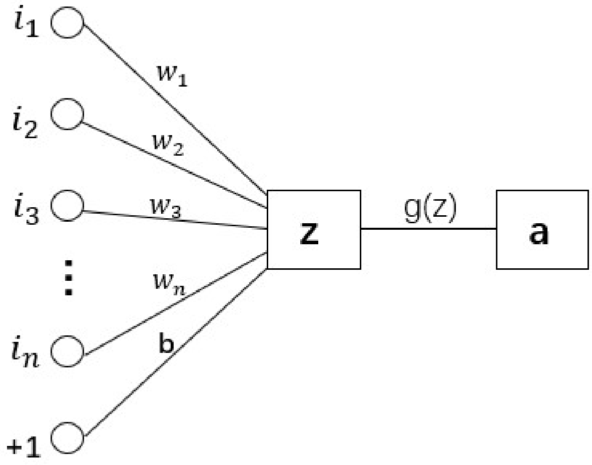


Fig. 2.1 A neural in neural network. i, w, b, g(z), a respectively refers to input vector, weight, bias, activation function, output.

2.1.2 Neural Network training and Inference

Training a neural network is the process of finding a set of parameters (weights and bias) that minimize the model's approximation error on the training dataset. The approximation error can be calculated by a loss function, which is typically determined based on the task [19].

Trained neural networks can apply what they have learned to applications, such as speech recognition, computer vision and medical imaging, and so on. This process of using trained model to predict and classify new data is referred as inference of NN.

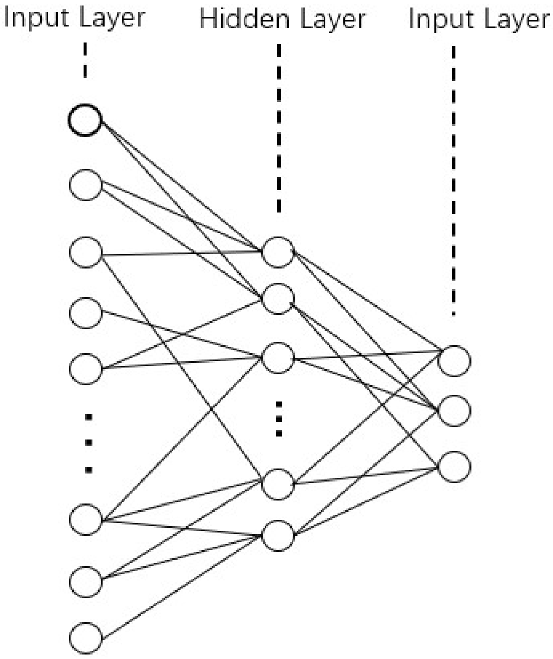
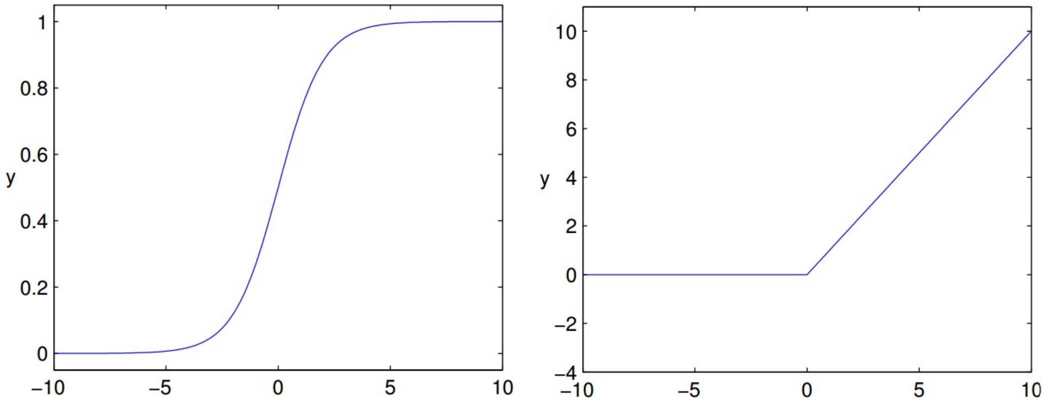


Fig. 2.2 Sketch map of a simple deep neural network



(a) Sigmoid function, y = $\frac{1}{1+e^{-x}}$ (b) ReLU function: y = (x>0) ? x:0

Fig. 2.4 Example activation functions

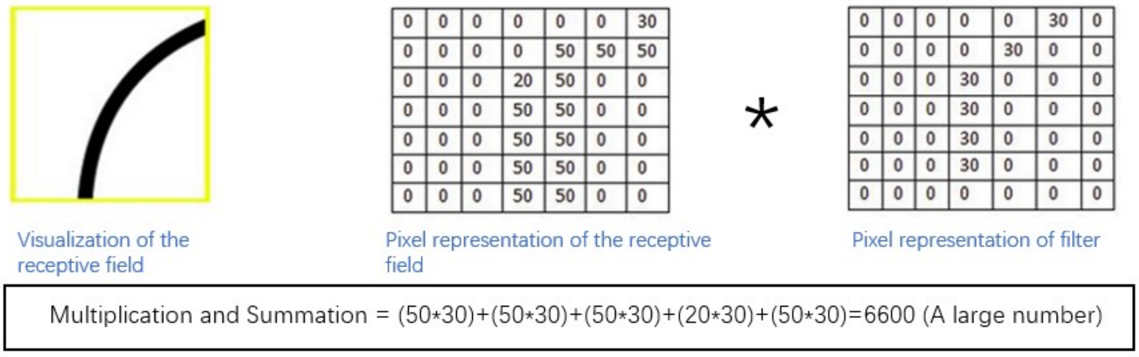
In this paper, the training process is not discussed and we only focus on the inference of NN.

2.1.3 Major layers

Convolution Layers

Convolution layer is composed of several convolution kernels, and the parameters of each kernel are optimized by the back propagation algorithm. The convolution operation is shown in the figure 2.5.

The filter acts like a sliding window that moves from top to bottom according to the input image. When there are lines in the input image that are very similar to that in the filter, the calculation result of this area will be large, while the calculation result of other parts will be small. When all the calculations are done, we will get an activation map that has high values in certain patterns and low values in other areas. When convolution layer goes deeper, the features to be detected become more complex. Therefore, when training a convolution layer of CNNs, a series of filters are actually trained to achieve image classification or detection.



0	0	0	0	0	0	30
0	0	0	0	50	50	50
0	0	0	20	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0

Pixel representation of the receptive field

0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

Pixel representation of filter

Multiplication and Summation = (50*30)+(50*30)+(50*30)+(20*30)+(50*30)=6600 (A large number)

Fig. 2.5 The process of the convolution operation between input and filter

Pooling Layers

Pooling payer is another common component in CNN. It was first used in Lenet [17], called the subsample. The pooling name is adopted after Alexnet [16]. The pooling layer imitates the human visual system to reduce the dimension of the data and represents the image with higher level features. The purpose of pooling is to reduce the information redundancy, keep good scale invariance and rotation invariance and to prevent overfitting.

Maxpooling is the most common and most used pooling operation. As is shown in Fig. 2.6, it reduces the input dimensionality by extracting the maximum value from a set of neighbouring inputs.

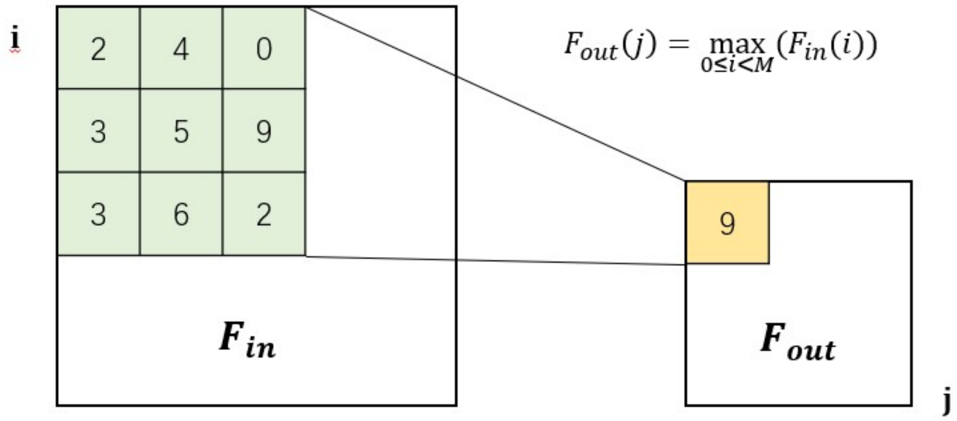


Fig. 2.6 Illustration of a maxpooling layer.

BatchNormalization Layer

Batch normalization (also known as batch norm) is a method used to make neural networks faster and more stable through normalization of the input layer by re-centering and re-scaling. It was proposed by Sergey Ioffe and Christian Szegedy in 2015 [14]. The computation can be formulated as follows

$$x_{out} = \gamma(\frac{x - \mu_c(x)}{\sigma_c(x)}) + \beta \tag{2.1}$$

The terms μ_c and σ_c are mean and variance at each channel of feature map, respectively. The γ and β are trainable parameters.

In the inference stage, μ_c and σ_c of each layer of each mini-batch training data are retained in the model and the statistics of the whole sample are used to normalize the test data.

Fully Connected(FC) Layers

Fully Connected (FC) Layers plays the role of ”classifier” in the neural network. If the operations of convolution layer, pooling layer are to map the original data to the hidden feature space, the full connection layer is to map the distributed feature representation learned to the sample tag space. In practice, the full connection layer can be realized by the convolution operation. As is shown in Fig.2.7, in a fully-connected layer, every neuron is connected to all the neurons in its previous layer.

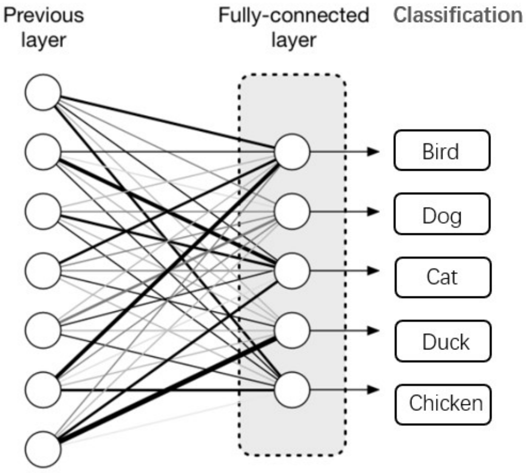


Fig. 2.7 Illustration of a fully connected layer.

2.1.4 Alexnet for The ImageNet Dataset

ImageNet is a large image dataset organized primarily by the Standford Vision Lab [2]. The dataset contains more than 14 miliion images that have been hand-annotated belonging to aroung 22000 categories. This dataset has become an invaluable resource for computer vision and machine learning researchers and has become the benchmark for testing algorithm performance in the field of machine learning. The ImageNet Large-Scale Visual Recongnition Challenge (ILSVRC) is the most influential competition for image classification and target detection that had been held annually for six years. It uses a subset of ImageNet, containing 12 million training images and 50 thousand validation images. In the 2012 competition, Krizhevsky et al. proposed a novel convolutional neural network architecture called AlexNet and make a significant breakthrough by decreasing the error rate of top1 and top5 from 47.1% and 28.2% to 37.5% and 17% [16]. From then on, Alexnet has been widely used as a reference model in many research papers. Fig. 2.8 illustrates the architecture of Alexnet.

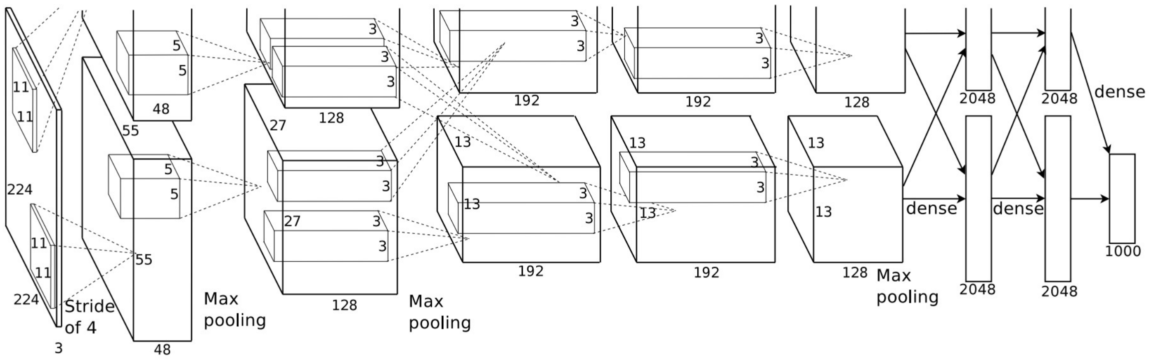


Fig. 2.8 An illustration of the AlexNet architecture (the figure is taken from [16])

AlexNet requires 60 million parameters and 1.5 billion FLOPS [32].

2.2 Data Quantization

As introduced in section 2.1, CNN models have a tremendous number of parameters and require huge computing power, it is a very challenging task to implement CNN on embedded systems that are resource-restricted. Researchers have proposed many possible solutions. More efficient network are designed from AlexNet [16] to ResNet [8], SqueezeNet [13], MobileNet [11], and NASNet [41]. Latest work tries to directly optimize the processing latency by searching a good network structure [31] or skip some layers at run-time to save computation [34]. But these schemes are designed to change the size of the layers or the way they are connected, the basic operations are the same and the differences hardly affect the hardware design. To make better use of the advantages of FPGA, we chose data quantization to lower down the computation and storage complexity. Data quantization refers to reduce the precision of the weights and/or activations from single precision which is usually 32-bits floating-point to lower bit representations.

Data quantization methods can be generally divided into two types: “Linear” and “Non-linear” .

Linear quantization uses symmetric thresholds to quantize high precision values. Every increment in the sampled value has a fixed size. Binary quantization, which binarizes weights and/or activations to -1 and +1, proposed by Hubara et al. [12] and Rastegari et al. [25] is a typical example. However, binary quantization generates a sizeable accuracy gap between the quantized model and their full-precision counterparts. Zhou et at. [40] proposed a higher precision model called DoReFa-Net to map floating-point numbers to their nearest fixed-point integers with arbitrary K bits quantization basis.

However, the problem with linear quantization is that each layer of CNN data is not evenly distributed, causing most of the weights and activations to suffer overflow or underflow. Non-linear quantization methods can help with these problems, in which the increment for small sample values is much smaller than the increment for large sample values. The step size is roughly proportional to the sample size. As an example of non-linear quantization, Li and Liu [18] proposed TWN (Ternary weight networks), which uses two symmetric thresholds $\pm\Delta l$ and a scaling factor W_l for each layer l to quantize weights into $-W_l$, 0, $+W_l$. Layer-wise values W_l and $\pm\Delta l$ are optimized during the training process. *Zhu et al.* further refined their scheme by assining two independent values W_l^p and W_l^n for positive and negative weights in each layer l. According to their paper, they even obtained better results than the full precision model using Alexnet on ILSVRC12 [26]. Furthermore, Zhang et al. [38] proposed a bit-operation-compatible CNN model called LQ-Nets based on arbitrary bit-width binary vector and trainable basis, which achieves a validation accuracy that is very close to TTQ proposed by Zhu et al. using 1-bit weights and 2-bit activations, while TTQ uses 2-bit weights and full precision activations. LQ-Net is a suitable quantization strategy for low-bit implementation on FPGA with acceptable accuracy loss. We adopt LQ-Nets to quantize both the weights and activations of CNN inference and apply the quantized inference to a multi-FPGA system. I will go into more detail about LQ-Nets and discuss why it is bit-operation compatible in section 3.2.2 of next chapter.

Chapter 3

Using Low-precision Binary Integer in Neural Networks

In this chapter, I will first introduce why it is not a good choice to implement traditional full-precision NN on FPGA from two aspects, data size and data calculation. In Sec.3.3, I will introduce two methods of implementing quantized NN on FPGA, BNN and LQ-Nets.

3.1 Gap Between NN Model size and FPGA Storage Size

Neural network computations are usually performed with 32 or 64 floating-point because they are easy to use on general processing platforms (CPUs or GPUs). And because of the large range of values that can be represented, the floating-point operation can achieve good results in model accuracy. However, high precision also means a large storage requirement.

On the other hand, the size of CNN has become larger and larger. For example, one of the fully connected layers in Alexnet and VGG uses a $4K \times 4K$ weight matrix.

When each weight is represented as a 32-bit number, storing the weight matrix would require 64MB of storage [22]. Besides, according to the investigation of [7], the model size of VGG-11, ResNet-152, ResNet-34, SqueezeNet are 531MB, 229MB, 86MB, 5MB, respectively. In contrast, on-chip RAMs of FPGA is relatively limited. As is shown in Fig. 3.1, common models implement 100-1000MB parameters while the largest available FPGA chip implements ¡50MB on-chip SRAM. The gap between the NN model size and the storage unit size on FPGAs is huge.

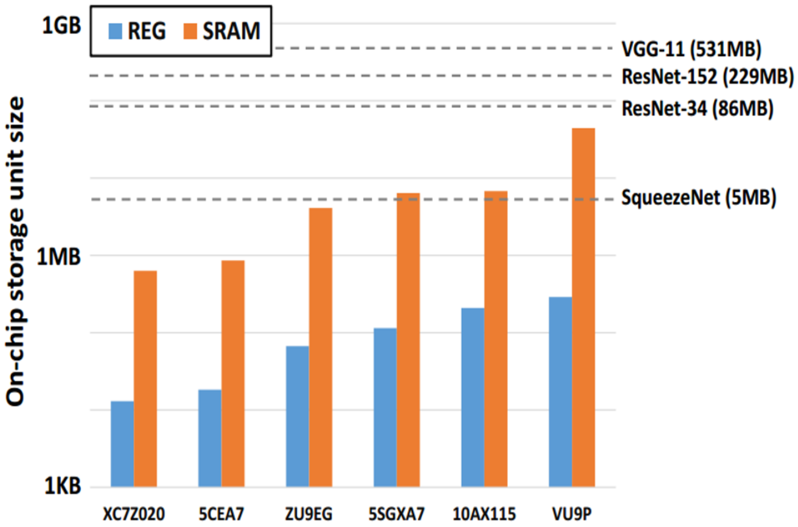


Fig. 3.1 The bar chart compares the register and SRAM sizes on FPGA chips in different scales. The dotted line denotes the parameter sizes of different NN models with 32-bit floating point parameters.

3.2 Floating-point Arithmetic on FPGAs

Inside microprocessors, numbers are presented as integers-one or several bytes stringed together. Numerical operations are usually performed in 32 bits, The 32 bits or four-byte can represent the numbers 0 to 4,294,967,295 or, alternatively, -2,147,483,648 to +2,147,483,647. A scientific representation of 20,422,242 is 2.0422242×10^7, while 1.001 can be represented as 1.001×10^0.

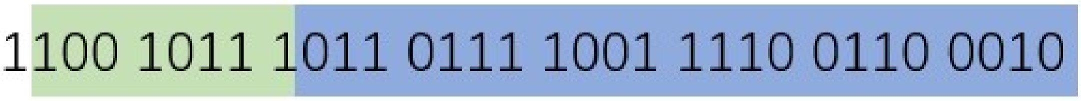
The 32-bit floating-point representation defined in IEEE standard 754 has 1 sign bit, 8 exponent bits and 23 mantissa bits. In the first example, 2.0422242 is the mantissa, 10 the exponent base, and 7 the exponent.

The sign bit simply defines the polarity of the number. A value zero means that the number is positive, whereas a 1 denotes a negative number. The exponent can not only represent positive values but also negative values. Such as 0.0014006 can be represented as 1.4006×10^{-3}. Thus the stored exponent is the sum of the actual exponent and a bias value. In the case of the single precision, the bias is 127. This means that the stored value of 130 indicates the actual exponent of 3. The exponent base is 2 by default.

Following the previous example of 20,422,242, the 32-bit representation of this value will be like:

The binary integer representation of 20,422,242 is 1 0011 0111 1001 1110 0110 0010. This can be written as $1.001101111001111001100010 \times 2^{24}$. The leading digit is omitted, and the fraction-the string of the digits following the radix point is 0011 0111 1001 1110 0110 0010. The sign is positive. Adding the bias of 127 to get the exponent value 151 and converting to binary yields an IEEE 754 exponent of 1001 0111.

Putting all of the pieces together, the single precision representation for 20,422,242 is shown in the Fig.3.2.



1100 1011 1011 0111 1001 1110 0110 0010

Fig. 3.2 20,422,242 represented in IEEE 754 single-precision format

A floating-point number representations on a computer uses something similar to a scientific notation with a base and an exponent.

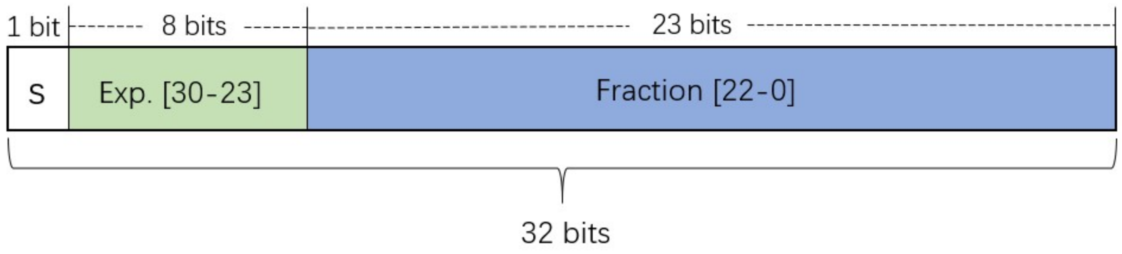
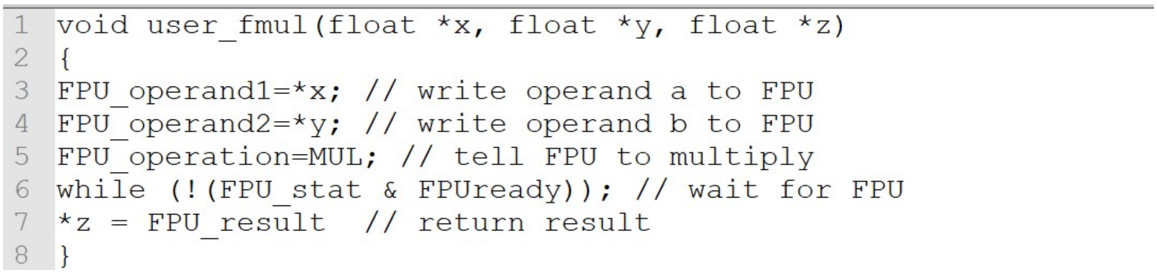


Fig. 3.3 IEEE 32-bit floating point format

But most embedded processor cores ALUs(arithmetic logic units) only support integer operations, so the circuits would simulate floating-point arithmetic in software. This severely affects processor performance. In a 32-bit CPU, adding two 32-bit integers only needs one machine code instruction, while in embedded system, a library routine including bit manipulations and multiple arithmetic operations is needed to add two IEEE single-precision floating-point values. With the increase of the number of multiplication and division, the performance gap just becomes bigger. Therefore, for many applications, software floating-point emulation is not practical.

With the MicroBlaze 4.00 processor, Xilinx makes an optional single precision FPU(floating-point coprocessor unit) available. But achieving real floating-point performance will cost extra logic. If we were to connect an FPU to the processor bus, FPU access would occur through specifically designed driver routines. For example, if to do the operation z = x*y, the driver function would be like Fig.3.4:



```
void user_fmul(float *x, float *y, float *z)
{
FPU_operand1=*x; // write operand a to FPU
FPU_operand2=*y; // write operand b to FPU
FPU_operation=MUL; // tell FPU to multiply
while (!(FPU_stat & FPUready)); // wait for FPU
*z = FPU_result  // return result
}
```

Fig. 3.4 Driver function of FPU access.

For small and simple operations, this may work reasonably well, but for complex operations like convolution operations in CNN, this approach has three major drawbacks: 1) The code will be hard to write, maintain and debug. 2) The overhead will greatly affect the performance. 3) Each operations will involves at least five bus transactions; as the bus is likely to be shared with other resources, this not only affects the performance, but may also increases the latency.

In conclusion, although 32-bit floating-point operation can hold relatively high precision, the implementation of 32-bit floating-point operation on embedded systems will consume additional resources and cause performance decline. Thus, even at the expense of losing some precision, with the integer-based implementation, we can benefit not only from ease-of-use but vast performance improvements as well.

3.3 FPGA Oriented Model Compression

As described in Sec.3.1, traditional neural networks are difficult to be implemented on FPGA. Many research works have proposed methods using external memory like DDR SDRAM. But DRAM accesses are significantly more energy consuming than on-chip operations and the bandwidth and the communication latency between internal and external memory will significantly limit the system performance.

Many works that reduce the size of CNN and computation complexity by quantizing the weights and activations have been proposed to address the issue. The implementation results show that the accuracy can be sufficient for inference [22] [23] [5] [33] [39]. In Sec.3.3.1, I will introduce some related works about implementing Binary Neural Network on FPGA. And in Sec.3.3.2, the LQ-Nets based FPGA Accelerator Design that we proposed will be stated.

3.3.1 BNN FPGA Accelerator Design

Binarized neural networks (BNNs) are widely used in FPGA implementation. As is described in Sec. 2.2, BNNs constraint weights and/or activations of CNN to either +1 or -1. Therefore, storage need can be dramatically reduced since the weights can be stored in a single bit(i.e, +1 stored as 1, and -1 as 0). Furthermore, multiply operations can be replaced by bit-wise exclusive NOR (XNOR) instead, thereby greatly reduce the computational complexity.

The convolution of CNN performed as an XNOR dot-product operation can be expressed as

$$Y[n][w'][h'] = \sum_{w=0}^{FW-1} \sum_{h=0}^{FH-1} \sum_{d=0}^{FD-1} \bar{W}[n][w][h][d] \oplus \overline{fmap}[w'+w][h'+h][d] .$$

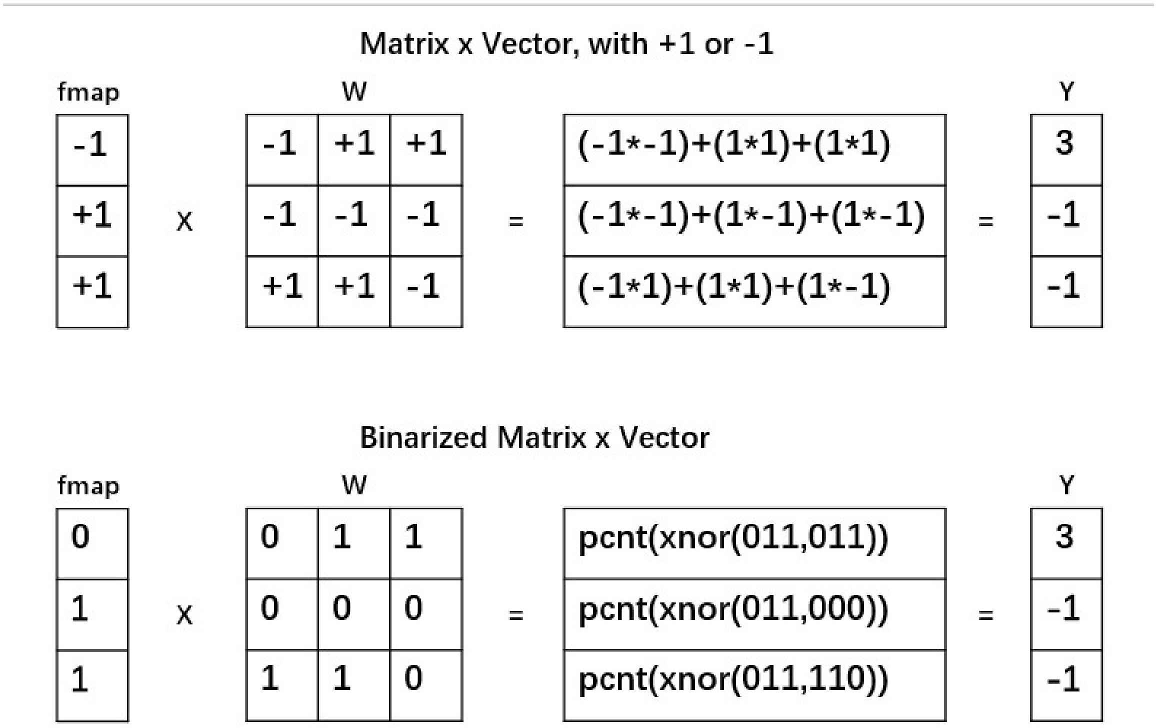
In which, w, h, d, n represent the width, height, depth, channel respectively. The output Y is obtained by the XOR operation between input feature map and the filters.

Figure 3.5 illustrates the detailed process of how a matrix x vector operation of +1 and -1 values can be binarized and computed using xnor and pcnt.

It is noteworthy that in practice, there are two types of BNN implementations: (a) binarized weights and full precision activations and (b) binarized weights and binarized activations. The two types of binarisation have been illustrated in Figure 3.6. Binarized weights and full precision activations generally reduce the storage requirements of the weight matrix by 32x and replace multiplication with a conditional negation. When performing the dot product within the neuron, the sign (+ for 1 and - for 0) of the binarized weights is applied to the activations, and the results are accumulated as normal. In the case of binarized weights and binarized activations, both the weights and activation matrices are reduced down to a single bit representation, and the standard multiply-accumulates are replaced by XNOR and a signed bit count.

3.3.2 LQ-Nets FPGA Accelerator Design

In this section, we introduce the quantization method of LQ-Nets.



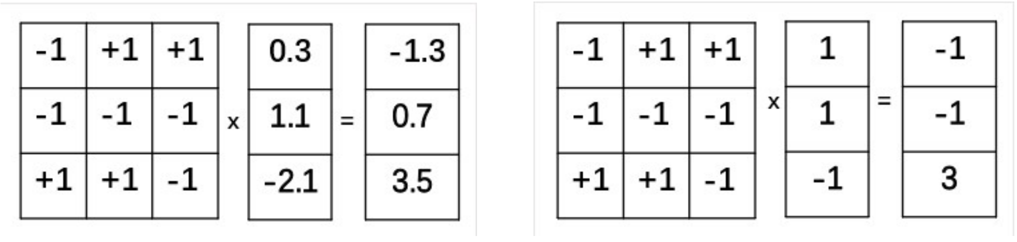
Matrix x Vector, with +1 or -1

$$\begin{array}{ccccccc} \text{fmap} & & W & & & & Y \\ \begin{bmatrix} -1 \\ +1 \\ +1 \end{bmatrix} & \times & \begin{bmatrix} -1 & +1 & +1 \\ -1 & -1 & -1 \\ +1 & +1 & -1 \end{bmatrix} & = & \begin{bmatrix} (-1*-1)+(1*1)+(1*1) \\ (-1*-1)+(1*-1)+(1*-1) \\ (-1*1)+(1*1)+(1*-1) \end{bmatrix} & = & \begin{bmatrix} 3 \\ -1 \\ -1 \end{bmatrix} \end{array}$$

Binarized Matrix x Vector

$$\begin{array}{ccccccc} \text{fmap} & & W & & & & Y \\ \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} & \times & \begin{bmatrix} 0 & 1 & 1 \\ 0 & 0 & 0 \\ 1 & 1 & 0 \end{bmatrix} & = & \begin{bmatrix} \text{pcnt(xnor(011,011))} \\ \text{pcnt(xnor(011,000))} \\ \text{pcnt(xnor(011,110))} \end{bmatrix} & = & \begin{bmatrix} 3 \\ -1 \\ -1 \end{bmatrix} \end{array}$$

Fig. 3.5 In binarized neural networks where weights and neurals are made +1 or -1, Matrix x Vector operation can be done using xnor and population count.



$$\begin{bmatrix} -1 & +1 & +1 \\ -1 & -1 & -1 \\ +1 & +1 & -1 \end{bmatrix} \times \begin{bmatrix} 0.3 \\ 1.1 \\ -2.1 \end{bmatrix} = \begin{bmatrix} -1.3 \\ 0.7 \\ 3.5 \end{bmatrix}$$

(a) Binary Weights and Real Activations

$$\begin{bmatrix} -1 & +1 & +1 \\ -1 & -1 & -1 \\ +1 & +1 & -1 \end{bmatrix} \times \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} = \begin{bmatrix} -1 \\ -1 \\ 3 \end{bmatrix}$$

(b) Binary Weights and Activations

Fig. 3.6 Two types of binary neural network implementations: (a) Binary Weights and real activations (b) both binary weights and binary activations.

LQ-Nets is originally proposed by Microsoft research group and like most of the data quantization methods, LQ-Nets finds the nearest fixed-point representation of each weight and activation. Specifically, a full precision number “q” represented by a K-bit binary encoding is actually the inner product between a basis vector and the binary coding vector $b = b_1, b_2, ..., b_K{}^T$ while $b_i \in 0, 1$,

$$q = \langle \begin{bmatrix} b_1 \\ b_2 \\ \dots \\ b_K \end{bmatrix}, \begin{bmatrix} basis_1 \\ basis_2 \\ \dots \\ basis_k \end{bmatrix} \rangle$$

Figure 3.7 illustrates the quantizer with the 2-bit and 3-bit cases.

Both the binary encoding and the basis vector are jointly trained during training process based on minimal quantization error criteria. The quantization error can be formulated as follows

$$Q^*(x) = argmin||B^TV - X||_2^2, \quad s.t. \quad B \in \{-1, 1\}^{K\times N}.$$

where $B = [b_1, ...b_N] \in \{-1, 1\}^{K\times N}$ is the encoding vector and $v \in R^K$ is the quantizer basis. The way to optimize the quatizers is through the forward passes during training. According the paper, this algorithm leads to much better performance in the experiments.

Similar to BCNN, LQ-Nets is capable of bit-wise operation. Let a weight vector $w \in R^N$ be encoded by the vector $b_i^w \in \{-1, 1\}^N$, i=1,...K_w where K_w is the bit-width for weights and b_i^w consists of the encoding of the i-th bit for all the values in w. Similarly, activation vector $a \in R^N$ is encoded by a $b_i^a \in \{-1, 1\}^N$, where j=1,...,K_a. It can be readily derived like

$$Q(w, v^w)^T Q(a, v^a) = \sum_{i=1}^{K_w} \sum_{i=1}^{K_a} v_i^w v_j^a (b_i^w \odot b_j^a)$$

where $v^w \in R^{K_w}$ and $v^a \in R^{K_a}$ are the learned basis vectors for the weight and activation quantizers respectively, and $\odot$ denotes the inner product with bitwise operations xnor and popcnt.

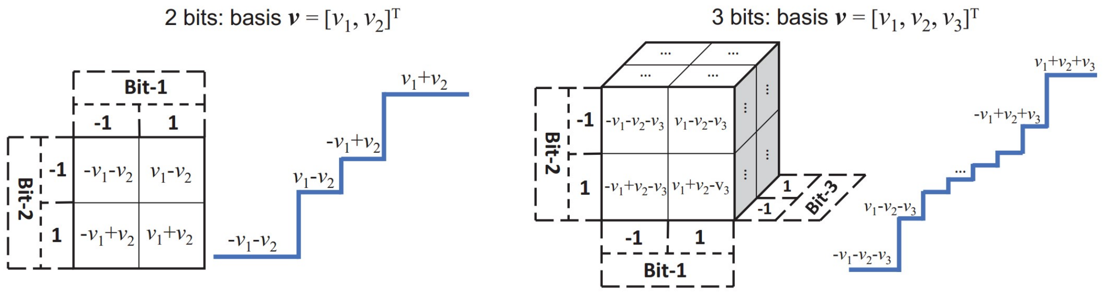


Fig. 3.7 Illustration of LQ-Nets's learnable quantizer on the 2-bit (left) and 3-bit (right) cases. For each case, the left figure shows how quantization levels are generated by the basis vector, and the right figure illustrates the corresponding quantization function. (The figure is taken from [38])

In our study, we use the source code of LQ-Nets and train Alexnet with 2-bit quantized weights and also quantize activations by 2 bits based on ImageNet dataset. According the paper, under the same circumstances(weights and activations are both 2 bit-width), the accuracy of Top-1 and Top-5 will reach up to 57.4% and 80.1%, respectively. Theoretically, the same accuracy can be achieved by deploying the same inference on FPGA.

The binary vector and quantizer basis and other required parameters are extracted from the model saved by Tensorflow. All of the parameters are numerically saved in text files. All these works were accomplished on a PC.

We adopt high-level synthesis(HLS) to generate the Neural Network form a high-level description in standard software language, C, and then apply the parameters aforementioned. The difficulty of building a neural network on HLS lies in that the mainstream neural network frameworks such as Tensorflow and Caffe are not supported with HLS, and all operations must be implemented in a basic way. Not only that, there are some differences in network construction between conventional Alexnet and LQ-

Nets' Alexnet. Besides, conventional Alexnet uses Local Response Normalization to scale variables, while LQ-nets adopts Batch Normalization instead.

In the next chapter, I will introduce our proposal to accelerate the convolution operation by taking advantage of LQ-Nets quantization method.

Chapter 4

System Architecture

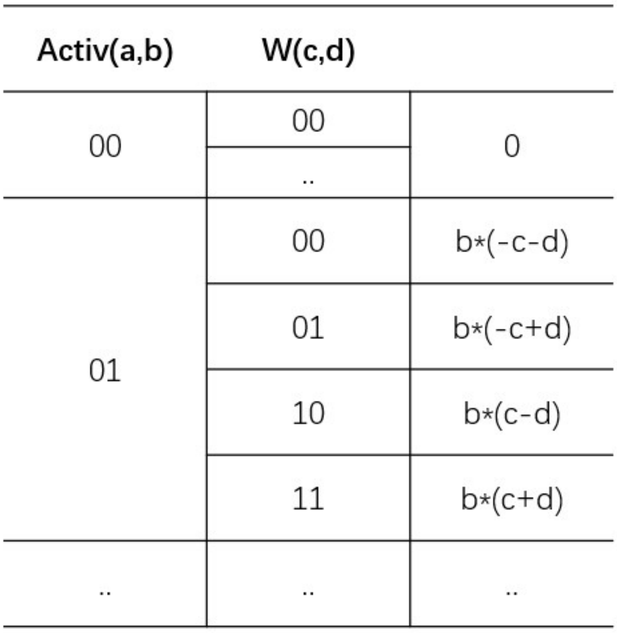
This chapter introduces the system architecture developed for accelerating the inference of neural networks on FPGAs. Section 4.1 discusses our proposals of the accelerator design from a software level and hardware level. Section 4.2 describes the whole system design.

4.1 Accelerator Design

There are many previous studies on accelerating neural networks in FPGAs with hardware level techniques. The design in [36] targets increasing the working frequency of the computation units. The design in [4] fuses two neighboring layers together to eliminate the intermediate result transfer between the two layers. Zhang et al. [37] propose a 2D DFT (Discrete Fourier Transformation) based hardware design for efficient CONV layer execution. [5] [20] [35] use the systolic array structure in which the shared data are transferred from one computational unit to the next in a chain mode,thus only local connections between different computation units are needed. In this paper, we also adopt several techniques to improve the system performance. In the next sections, we will introduce them respectively.

4.1.1 Fast Convolution Method

As is mentioned in Sec. 3.3.2, we use the LQ-Nets quantization method to represent both the weights and activations into 2 bit-width level. There are total 4 combinations, 00, 01, 10 and 11. For example, let an actual value of weight be 0.7 and the trained basis be 1.1 and 0.5. The nearest quantization level to 0.7 will be 0.6 among -1.6, -0.6, 0.6, and 1.6. Thus it will be quantized as {1, -1} and stored as 10. However, different from the weights, the activation value is always greater than 0 due to the presence of the ReLU layer. Therefore, the binary code 0 of the activation value represents the actual value of 0. This means that a binary coding vector 10 will not represent the quantization level of $basis_1 \times 1 + basis_2 \times (-1)$ but $basis_1 \times 1 + basis_2 \times 0$. The logic table is shown in the Fig. 4.1.



Activ(a,b)	W(c,d)	
00	00	0
	..	
01	00	b*(-c-d)
	01	b*(-c+d)
	10	b*(c-d)
	11	b*(c+d)
..	..	..

Fig. 4.1 Binary LQ-Nets implementation

Based on this table, we hypothesize three methods to accelerate the convolution layer. The first method that came to our mind is as shown in Fig. 4.2: multiplications can be replaced by if conditionals. But using too many if conditionals in HLS is a kind of low power technique. We ended up adopting a lookup table, by translating the binary code to its corresponding value. The calculated lookup table will be written to memory in advance, and referenced by an index. Fig. 4.3 shows the working mechanism of the look-up table with 2-2-bit inputs. This method helps remove the floating-point level operation in CONV layers and synthesis results show that quicker operations can be achieved.

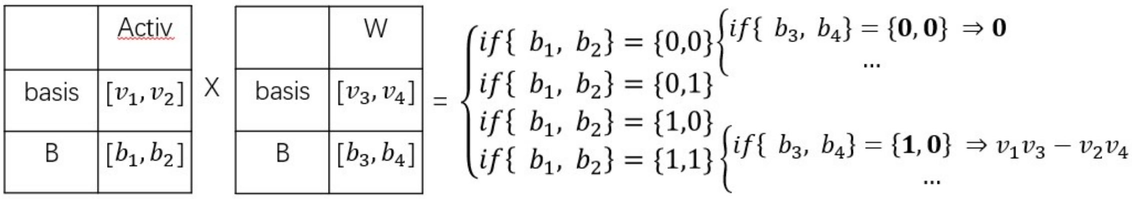

Activ
basis
$[v_1, v_2]$
B
$[b_1, b_2]$
X
W
basis
$[v_3, v_4]$
B
$[b_3, b_4]$
=
$if\{b_1, b_2\} = \{0,0\}$
$if\{b_3, b_4\} = \{\mathbf{0}, \mathbf{0}\} \Rightarrow \mathbf{0}$
...
$if\{b_1, b_2\} = \{0,1\}$
$if\{b_1, b_2\} = \{1,0\}$
$if\{b_1, b_2\} = \{1,1\}$
$if\{b_3, b_4\} = \{\mathbf{1}, \mathbf{0}\} \Rightarrow v_1v_3 - v_2v_4$
...

Fig. 4.2 Binary implementation by if-conditional

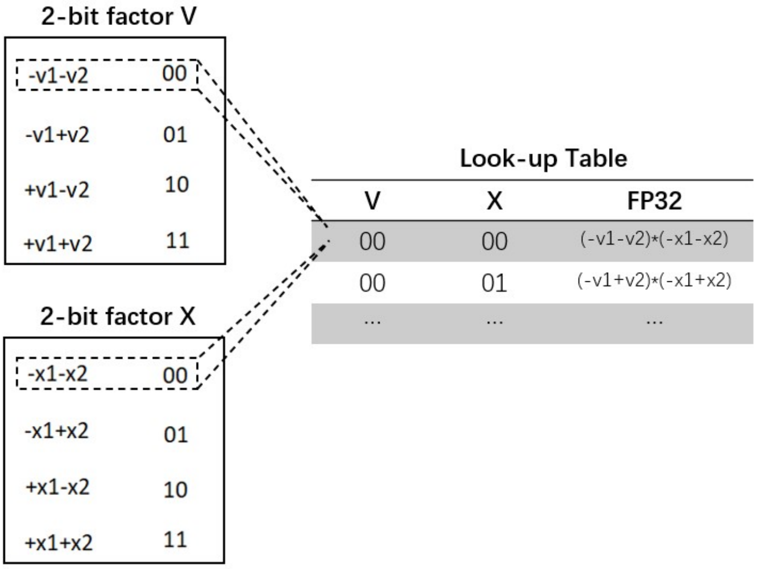


Fig. 4.3 We use a look-up table to replace the multiplication function.

The implementation is shown in the pseudo code in listing 4.1.1, in which window and filter are both binary numbers of 2bits. The table holds the corresponding 4-bit to 32-bit floating point results were read in advance.

```
for row in range(31−5) :
  for col in range(31−5) :
    for i in range(5) :
      for j in range(5) :
        window[i][j] = image[row + i][col + j]
    for i in range(5) :
      for j in range(5) :
        sh[i][j].range(3,2) = window[i][j]
        sh[i][j].range(1,0) = filter[i][j]
        res[i][j] = table[sh[i][j]]
    lm = 0
    for i in range(5) :
      for j in range(5) :
        lm += res[i][j]
    conv_out[row][col] += lm
```

Listing 4.1 Table accelerator pseudo code for convolution layer

Recently, we are also considering the third operation, the bit operation. As is shown in Fig. 4.4, let's assume that we have a window of size 2 by 2 doing the process of convolution. A, B, C, D represent the basis values. With *AND* operation between binary coding vectors of window and weight, activation can be obtained. Then, let the popcount operation calculate the sum of the number of 1 bits in each bit of the four activation binary vector, and assume that the numbers obtained are m,n,l,k, respectively. The formula for the final result is shown on the right.

The method is still theoretical and we will implement it in the future.

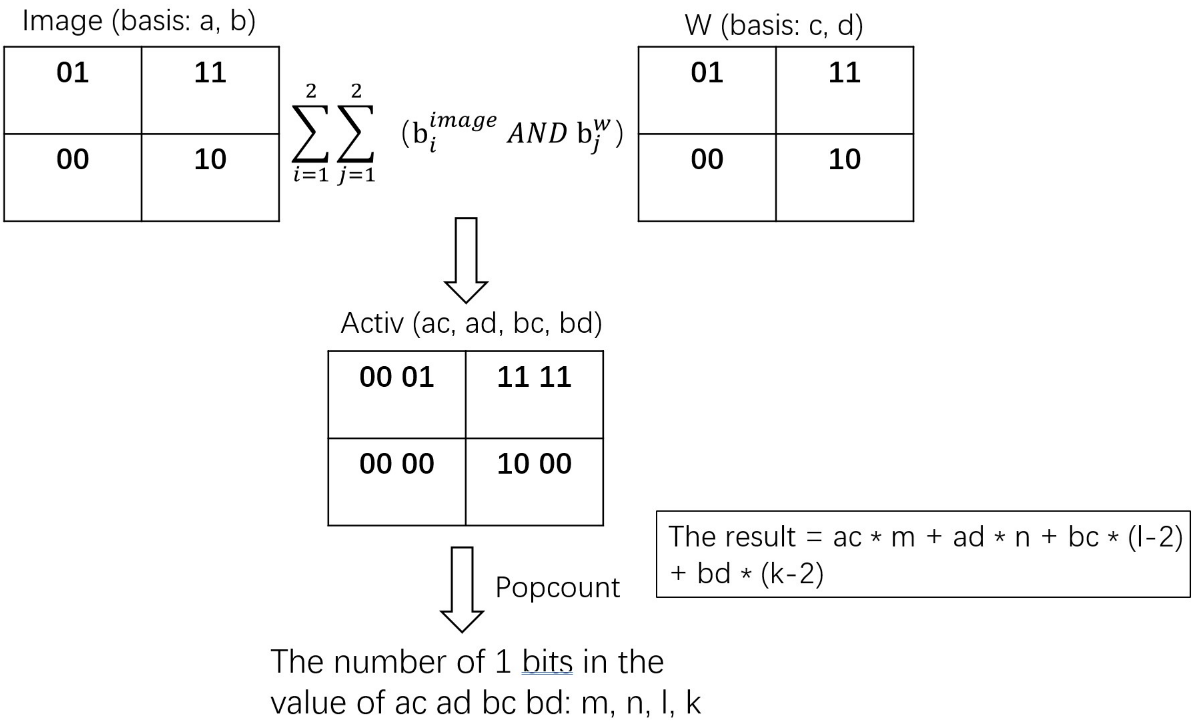


Image (basis: a, b)
01
11
00
10
$\sum_{i=1}^{2}\sum_{j=1}^{2}(b_i^{image}\ AND\ b_j^{w})$
W (basis: c, d)
01
11
00
10
Activ (ac, ad, bc, bd)
00 01
11 11
00 00
10 00
Popcount
The result = ac * m + ad * n + bc * (l-2) + bd * (k-2)
The number of 1 bits in the value of ac ad bc bd: m, n, l, k

Fig. 4.4 Bit operation we proposed based on LQ-Nets

4.1.2 Data Transfer and On-Chip Storage

As mentioned earlier, the section gap between NN model size and the storage unit size on the FPGA is huge. Using external memory like DDR and SDRAM will bring extra transfer latency and affect the system performance. Thus we chose to use only on-chip memory. However, even though the parameters of the model are quantized and the model size is reduced, our implementation results show that one FPGA BRAM is still not enough for the whole model. Therefore, we chose to realize cooperative storage and mathematical operation between multiple FPGAs with the aid of the FIC(Flow in Cloud) system [10]. Thus, the BRAM capacity will be proportional to the number of FPGAs within the system.

As shown in Fig. 4.5, a custom FPGA board of FIC is called FIC-SW, consisting of a mid-range economical Xilinx Kintex Ultrascale XCKU095 or XCKU115, 16GB DDR4-SDRAM, a Raspberry-Pi3 (RPi3), and 32 9.9Gbps serial links, has been de-

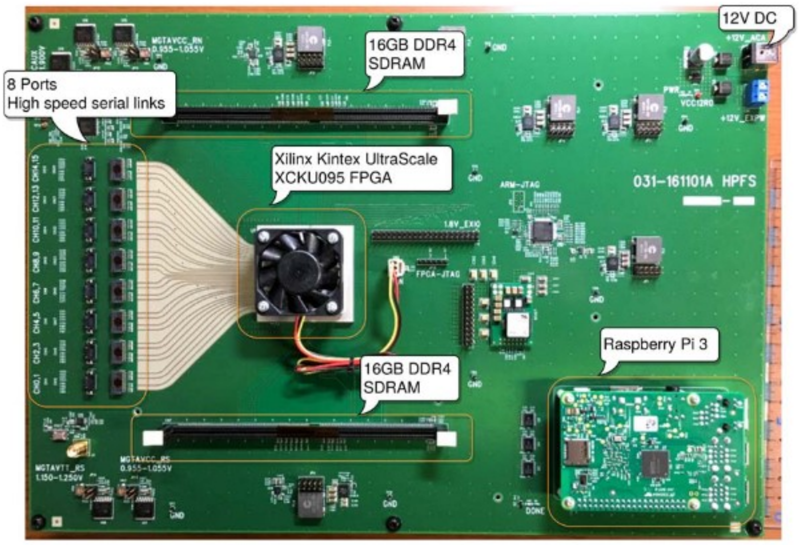


Fig. 4.5 A FIC-SW Custom FPGA board [10]

veloped. The two FPGA boards we currently use are XCKU095，which has 59.1MB total block RAM and 537,600 CLB LUTs, further resource list available at [1].

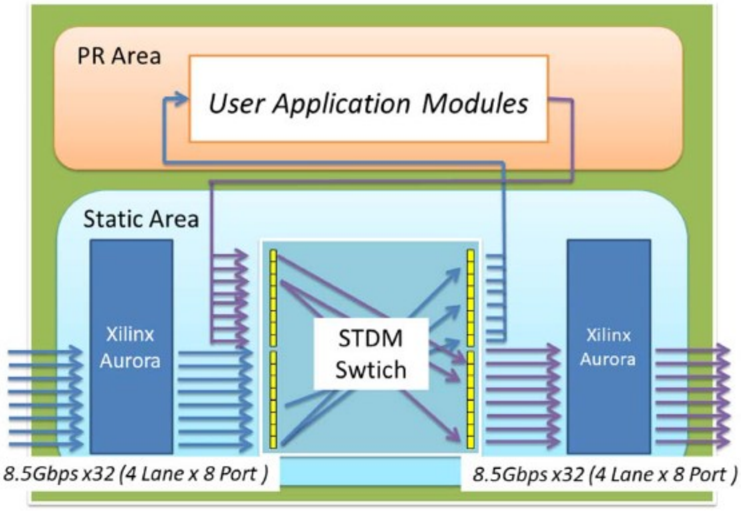


Fig. 4.6 A diagram of the FPGA on a FIC-SW board [10]

As shown in Fig 4.6, each FPGA has a zone that can be reconfigured to accommodate different user applications. And each board provides four 9×9 (9 inputs x 9 outputs) switches (8 channels + one port from/to internal HLS module) at the largest configuration. In the FIC system, each FPGA is connected directly with high-speed serial links that reach up to a maximum communication bandwidth of 34 Gbps per port.

Currently, we deploy the first two CONV layers of Alexnet to the FIC system with each layer implemented in one FPGA chip without any external memory capacity. According to our experimental estimation, one FPGA board resource is enough for one CONV layer(below 50% BRAM usage and below 30% logic resources usage) while FIC system contains over 20 boards. Thus we have reason to believe the whole Alexnet model can be deployed on FIC system.

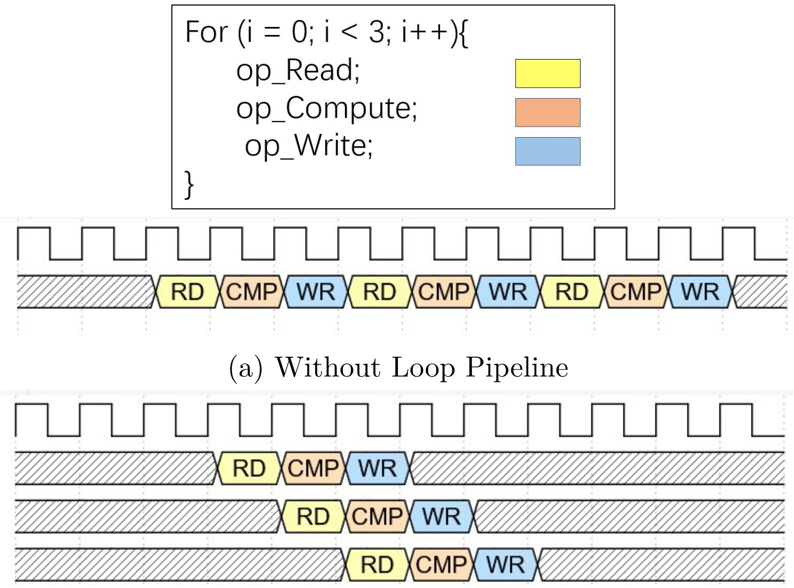
4.1.3 Loop Pipeing and Unrolling

In this section, I will first describe some basic concepts of *pipeline* and *unroll* and then how we use these pragma to optimize our program.

Pipelining reduces the initiation interval for a function or loop by allowing the concurrent execution of operations. As shown in the Fig. 4.7, let us assume that there are three operations in a FOR loop, READ, COMPUTE and WRITE, each of which takes three clocks. As is shown is (a), if no pipeline is performed, the total elapsed time will be 9 clocks. If the loop is pipelined like (b), the second round of READ will run concurrently with the first round of COMPUTE and each round staggers a clock. This reduces the final operation time to 5 clocks.

The UNROLL pragma, on the other hand, transforms loops by creating multiples copies of the loop body in the RTL design, which allows some or all loop iterations to occur in parallel. For example, given the following code:

```
for( int i=0; i < K; i++)
{
    pragma HLS unroll factor=2
    a[i] = b[i] + c[i];
}
```



```
For (i = 0; i < 3; i++){
      op_Read;
      op_Compute;
       op_Write;
}
```

(a) Without Loop Pipeline

(b) With Loop Pipeline

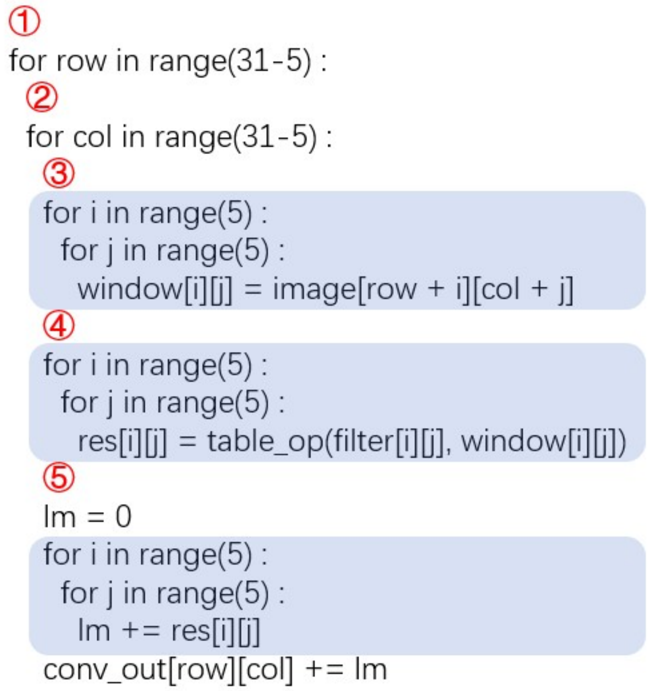
Fig. 4.7 Loop Pipeline

loop unrolling by a factor 2 will create two independent RTL design for the $a[i] = b[i] + c[i]$ operation. The total latency can be reduced by half.

Next, I will introduce our directive optimization for HLS. We mainly focused on the CONV layers which have the most computational burden. We designed a test program that is similar to our actual CONV layer implementation to analyze the performance under different compilation commands.

Figure 4.8 is the pseudocode for our test program and the red numbers from 1 to 5 represent the 5 possible places in which we add pragma. The total process can be divide into three parts circled by the blue boxes. The first part is to put the input image into the window, the second part is to calculate between convolution kernel and the window with our look-up table method, and the third part is to add the calculation results to get an activation.

As shown in the figure 4.1, we tried three different ways of adding pragma. Numbers below are the latency for each case. The results show that the first solution produces the best performance, which reduces latency to less than 1%.



```
①
for row in range(31-5) :
  ②
  for col in range(31-5) :
    ③
    for i in range(5) :
      for j in range(5) :
        window[i][j] = image[row + i][col + j]
    ④
    for i in range(5) :
      for j in range(5) :
        res[i][j] = table_op(filter[i][j], window[i][j])
    ⑤
    lm = 0
    for i in range(5) :
      for j in range(5) :
        lm += res[i][j]
    conv_out[row][col] += lm
```

Fig. 4.8 Pseudocode of our test code

	Solution 1	Solution2	Solution3
1	Pipeline		
2			Pipeline
3		Unroll	Unroll
4		Unroll	Unroll
5		Unroll	Unroll
Latency	**865**	**140022**	Synthesis failed
No directive	**207819**		

Table 4.1 Three optimization solutions we proposed

But by our calculations, solution 3 should work best if all of these instructions are executed correctly. In addition, solution 2 did not produce the results we anticipated. As I set the unroll factor to 5, the total latency should be reduced to one fifth.

We consider five reasons for this result: (1) Data dependency of the array may be a problem. (2) Xilinx Vivado HLS tool cannot know that there is no dependency relationship. (3) The circuit scale becomes too large for current resources. (4) Unknown reasons caused by synthesis tool. (5) The circuit becomes too complicated to be synthesised correctly.

We are still searching for further reasons.

4.2 System Design

As is shown in figure 4.9, in our current experiment, we deploy each layer on one FPGA board of the FIC system. The FPGAs can communicate with each other over the STDM switch. After the operations in one layer are completed, the processed data will be submitted to the next FPGA. After going through all the FPGAs, the final results will be obtained.

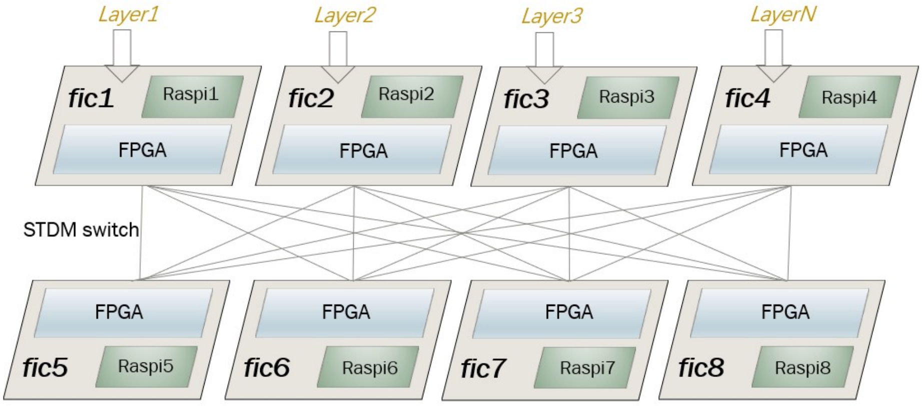


Fig. 4.9 Illustration of NN layer implementation in FIC system

Fig. 4.10 shows the implementation of two quantized layers in the FIC system. Weights and activations in layer2 are quantized into 2 bits, while only activations are quantized into 2 bits in layer1. Raspi, as host, passes parameters to the FPGA BRAM, including weights and bias and parameters required for BN layers. Since Raspi only supports 4-bit parallel transmission, the rx32 module is used to transform the 4-bit data into 32-bit data for FPGA inputs. FPGAs perform the calculations. The STDM switch is for the data transmission between the two FPGAs, which allows the speed to reach up to 34Gbps. It is worth noting that at the second layer, we divide the inputs into two parts: the 32-bit values and the 4-bit values. The former is for the lookup table and BN layer and the latter are merged from 2-bit binary weights. The mergence is for Raspbi to ship data easily. Similarly, the output of the layer2 will be merged from 2 bits to 4 bits.

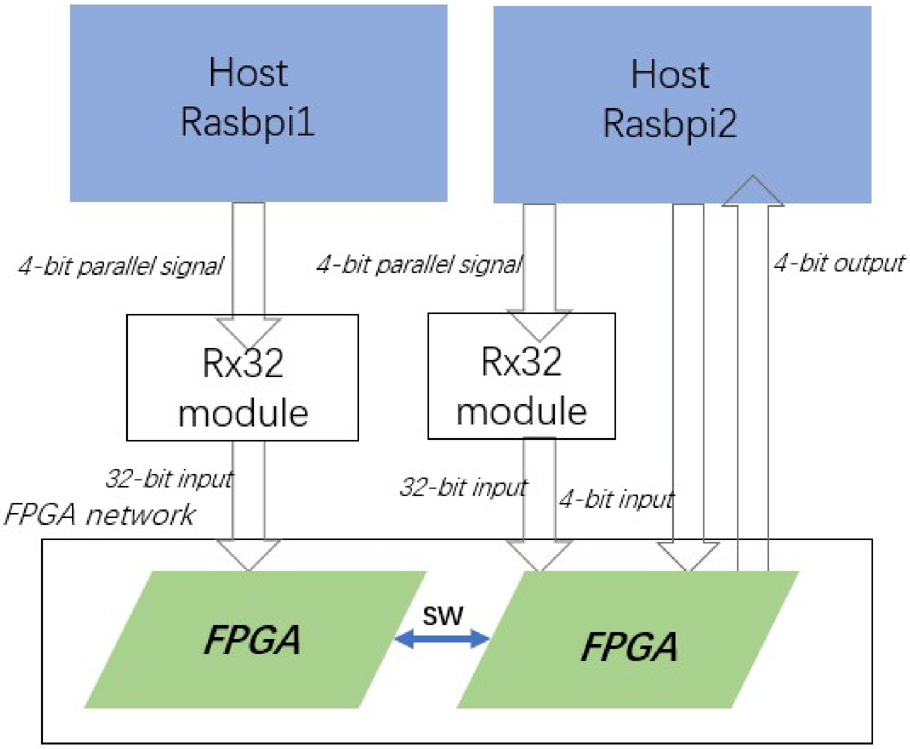


Fig. 4.10 Illustration of implementation of two quantized layers

Currently, we have only finished the implementation of the first layer, and the implementation of the two layers is still in progress. In addition, implementing one layer in one FPGA is obviously not an optimized solution. In the future, we will look for more efficient solutions to allocate resources. For example, let two layers be executed in one FPGA, or divide the computation operations in one layer into several parts and deploy each part in different FPGA to realize parallel computing.

Chapter 5

Synthesis Results

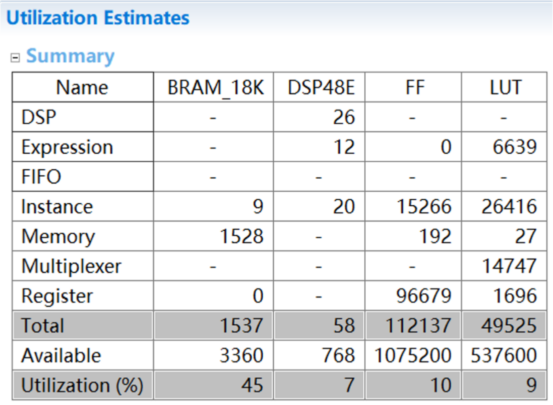
In this chapter, I present our synthesis results on resource consumption and total execution time of our proposed method and FP32 method, generated by the Xilinx Vivado HLS synthesis tool.

Figure 5.1 shows the Utilization estimates obtained by Xinlinx Vivado HLS synthesis tool. The logic modules and multiply-and-add modules are generated and automatically mapped to DSP/FF/LUT by Vivado. Our following discussion will base on these tables.

5.1 Resource Utilization Analysis

I summarize the total logical resource utilization into the table 5.1. “trad_1/2” denotes using FP32 parameters and traditional convolution operation for the first and the second layer. ”pps_1/2” denotes using quantized parameters and our proposed convolution method.

About layer1, there is a small reduction in the layer1 BRAM usage using our method. Even though the calculation operations in both methods are performed with FP32 parameters, the output activation value of the proposal is two binary bits. This may be the reason for the layer1 result.

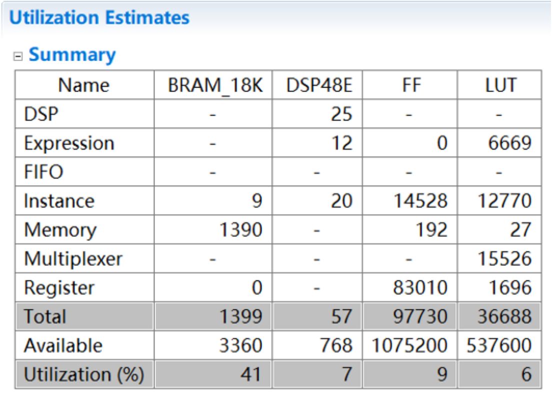


Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	26	-	-
Expression	-	12	0	6639
FIFO	-	-	-	-
Instance	9	20	15266	26416
Memory	1528	-	192	27
Multiplexer	-	-	-	14747
Register	0	-	96679	1696
Total	1537	58	112137	49525
Available	3360	768	1075200	537600
Utilization (%)	45	7	10	9

(a) Layer1 with traditional method

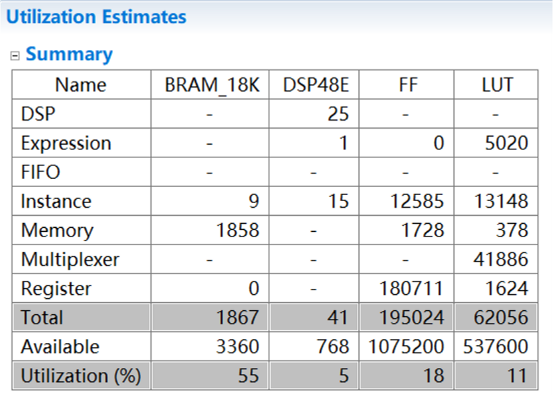


Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	25	-	-
Expression	-	12	0	6669
FIFO	-	-	-	-
Instance	9	20	14528	12770
Memory	1390	-	192	27
Multiplexer	-	-	-	15526
Register	0	-	83010	1696
Total	1399	57	97730	36688
Available	3360	768	1075200	537600
Utilization (%)	41	7	9	6

(b) Layer1 with proposal method

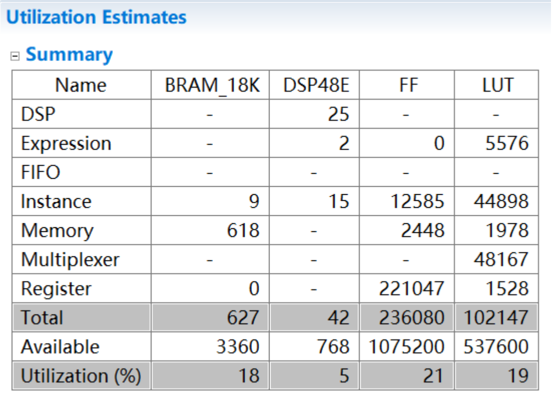


Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	25	-	-
Expression	-	1	0	5020
FIFO	-	-	-	-
Instance	9	15	12585	13148
Memory	1858	-	1728	378
Multiplexer	-	-	-	41886
Register	0	-	180711	1624
Total	1867	41	195024	62056
Available	3360	768	1075200	537600
Utilization (%)	55	5	18	11

(c) Layer2 with traditional method



Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	25	-	-
Expression	-	2	0	5576
FIFO	-	-	-	-
Instance	9	15	12585	44898
Memory	618	-	2448	1978
Multiplexer	-	-	-	48167
Register	0	-	221047	1528
Total	627	42	236080	102147
Available	3360	768	1075200	537600
Utilization (%)	18	5	21	19

(d) Layer2 with proposal method

Fig. 5.1 Utilization estimates generated by Xinlinx Vivado HLS synthesis tool

Methods	BRAM_18K	DSP48E	FF	LUT
trad_1	1537	58	112137	49525
pps_1	**1399**	**57**	**97730**	**36688**
trad_2	1867	41	195024	62056
pps_2	**627**	**42**	**236080**	**101247**

Table 5.1 Comparison of the resource utilization

Methods	Latency(clock cycles)	
	min	max
trad_1	261367581	262756056
pps_1	**261367581**	**262756056**
trad_2	2393524791	2393697847
pps_2	**1739580726**	**1739753782**

Table 5.2 Comparison of the execution time

As for layer2, the required BRAM for our method is scaled down to 1/3 compared with the traditional method, showing the validation of our design. Though the use of FF and LUT has increased as compensation, the current situation is more likely resource-bounded and the BRAM has the priority over FF and LUT, since the utilization of FF and LUT of both methods are less than 20% while the utilization of BRAM with the traditional method reaches up to 50%.

5.2 Performance Analysis

The latency of each method is summarized in table 5.2. It shows that the latency of trad_1 and pps_1 are the same, and that the layer2 with our proposed method reduces nearly 30% of the latency compared with trad_2.

In LQ-Nets, the first-layer convolution operation is no different from the previous method, which is to multiply the image of FP32 and the convolution kernel of FP32 to get the activation. We create the same network architecture to be consistent with LQ-Nets, so there is no difference in latency between the two methods of the first layer.

However, in layer2, we use a look-up table to replace the complex multiplication,thus our proposal has a good effect in reducing the total execution time.

It is worth noting that operations in layer3 and layer4 and the other layers after will be similar to operations in layer2 instead of layer1. Therefore, with the increase of the layer numbers, the decrease of BRAM usage and latency will be more obvious.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this research, we proposed a design for accelerating CNN inference in a multiple FPGA system with the LQ-Nets quantization method.

The difference between LQ-Nets and the previous quantization methods is that LQ-Nets has a basis vector that can be optimized by the training iterations. LQ-Nets improves the accuracy while retaining the support of bit operation. The multi-FPGA system we use is a pre-existing system called Flow in Cloud(FIC) system. In the FIC system, each FPGA is directly linked with high-speed serial links. Communication latency between FPGAs can be ignored.

A highly optimized FPGA implementation can be generated by Vivado HLS tools without involving any RTL programming. We added a pipeline pragma to the c code, which automatically generates efficient streaming hardware designs.

The innovation of our scheme is that we only use the on-chip memory of FPGA to store the parameters and use a look-up table to accelerate the convolution layer by taking advantage of narrow bit-width representations.

We achieved better resource utilization and execution time performance according to our synthesis results.

6.2 Future work

One future direction is to understand the underlying principles of HLS pragma. We tried to use the UNROLL pragma but it did not produce the expected effect. We summarize some reasons for this in section 4.1.3 based on the existing cognition. In the future, we will make more test programs or look at the Verilog code generated by HLS to figure out the mechanism. We are seeking a more efficient optimization scheme.

The other direction is to continue to implement on the physical FPGA boards. Presently, we finished implementing one layer on one FPGA board. In the next stage, we will work on implementing two layers on two FPGA boards. We will implement the whole Alexnet in the future.

Publication

国内研究会

- Hongyi Pan, Ben Ahmed Akram, Ikegami Tsutomu, Tominaga Kazuki, Kudoh Tomohiro. Quantization-based Optimization of CNN Inference. IEICE Technical report, vol. 120, no. 339, RECONF2020-69, pp. 63-68, Jan. 2021

References

[1] Ultrascale fpga product tables. https://www.xilinx.com/support/documentation/selection-guides/ultrascale-fpga-product-selection-guide.pdf.

[2] Standford vision lab. http://vision.stanford.edu.

[3] Ejaz Ahmed, Abdullah Gani, Mehdi Sookhak, Siti Hafizah Ab Hamid, and Feng Xia. Application optimization in mobile cloud computing: Motivation, taxonomies, and open challenges. *Journal of Network and Computer Applications*, 52:52–68, 2015.

[4] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. Fused-layer cnn accelerators. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12. IEEE, 2016.

[5] Utku Aydonat, Shane O'Connell, Davor Capalija, Andrew C Ling, and Gordon R Chiu. An opencl™ deep learning accelerator on arria 10. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 55–64, 2017.

[6] Y. Bengio, A. Courville, and P. Vincent. Representation learning: A review and new perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(8):1798–1828, 2013. doi: 10.1109/TPAMI.2013.50.

[7] Kaiyuan Guo, Shulin Zeng, Jincheng Yu, Yu Wang, and Huazhong Yang. [dl] a survey of fpga-based neural network inference accelerators. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 12(1):1–26, 2019.

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[9] Suzana Herculano-Houzel. The human brain in numbers: a linearly scaled-up primate brain. *Frontiers in human neuroscience*, 3:31, 2009.

[10] K. Hironaka, K. Iizuka, A. Ben Ahmed, M. M. I. Ullah, Y. Yamauchi, Y. Sun, M. Yamakura, A. Hiruma, and H. Amano. Demonstration of flow-in-cloud: A multi-fpga system. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 417–418, 2019. doi: 10.1109/FPL.2019.00074.

[11] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[12] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 29, pages 4107–4115. Curran Associates, Inc., 2016. URL https://proceedings.neurips.cc/paper/2016/file/d8330f857a17c53d217014ee776bfd50-Paper.pdf.

[13] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[14] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[15] S. Jiang, D. He, C. Yang, C. Xu, G. Luo, Y. Chen, Y. Liu, and J. Jiang. Accelerating mobile applications at the network edge with software-programmable fpgas. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, pages 55–62, 2018. doi: 10.1109/INFOCOM.2018.8485850.

[16] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6): 84–90, 2017.

[17] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[18] Fengfu Li, Bo Zhang, and Bin Liu. Ternary weight networks. *arXiv preprint arXiv:1605.04711*, 2016.

[19] Ruo Long Lian. *A framework for FPGA-based acceleration of neural network inference with limited numerical precision via high-level synthesis with streaming functionality.* PhD thesis.

[20] Yufei Ma, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. Optimizing loop operation and dataflow in fpga acceleration of deep convolutional neural networks. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 45–54, 2017.

[21] Pavel Mach and Zdenek Becvar. Mobile edge computing: A survey on architecture and computation offloading. *IEEE Communications Surveys & Tutorials*, 19(3): 1628–1656, 2017.

[22] Eriko Nurvitadhi, David Sheffield, Jaewoong Sim, Asit Mishra, Ganesh Venkatesh, and Debbie Marr. Accelerating binarized neural networks: Comparison of fpga, cpu, gpu, and asic. In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 77–84. IEEE, 2016.

[23] Eriko Nurvitadhi, Ganesh Venkatesh, Jaewoong Sim, Debbie Marr, Randy Huang, Jason Ong Gee Hock, Yeong Tat Liew, Krishnan Srivatsan, Duncan Moss, Suchit Subhaschandra, et al. Can fpgas beat gpus in accelerating next-generation deep neural networks? In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 5–14, 2017.

[24] Milan Patel, Brian Naughton, Caroline Chan, Nurit Sprecher, Sadayuki Abeta, Adrian Neal, et al. Mobile-edge computing introductory technical white paper. *White paper, mobile-edge computing (MEC) industry initiative*, pages 1089–7801, 2014.

[25] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European conference on computer vision*, pages 525–542. Springer, 2016.

[26] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3):211–252, 2015.

[27] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. The case for vm-based cloudlets in mobile computing. *IEEE pervasive Computing*, 8(4):14–23, 2009.

[28] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE internet of things journal*, 3(5):637–646, 2016.

[29] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[30] Tarik Taleb, Sunny Dutta, Adlen Ksentini, Muddesar Iqbal, and Hannu Flinck. Mobile edge computing potential in making cities smarter. *IEEE Communications Magazine*, 55(3):38–43, 2017.

[31] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2820–2828, 2019.

[32] Danai Triantafyllidou and A. Tefas. A fast deep convolutional neural network for face detection in big visual data. In *INNS Conference on Big Data*, 2016.

[33] Yaman Umuroglu, Nicholas J Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. Finn: A framework for fast, scalable binarized neural network inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 65–74, 2017.

[34] Xin Wang, Fisher Yu, Zi-Yi Dou, Trevor Darrell, and Joseph E Gonzalez. Skipnet: Learning dynamic routing in convolutional networks. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 409–424, 2018.

[35] Xuechao Wei, Cody Hao Yu, Peng Zhang, Youxiang Chen, Yuxin Wang, Han Hu, Yun Liang, and Jason Cong. Automated systolic array architecture synthesis for high throughput cnn inference on fpgas. In *Proceedings of the 54th Annual Design Automation Conference 2017*, pages 1–6, 2017.

[36] Ephrem Wu, Xiaoqian Zhang, David Berman, and Inkeun Cho. A high-throughput reconfigurable processing array for neural networks. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, 2017.

[37] Chi Zhang and Viktor Prasanna. Frequency domain acceleration of convolutional neural networks on cpu-fpga shared memory system. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 35–44, 2017.

[38] Dongqing Zhang, Jiaolong Yang, Dongqiangzi Ye, and Gang Hua. Lq-nets: Learned quantization for highly accurate and compact deep neural networks. In *Proceedings of the European conference on computer vision (ECCV)*, pages 365–382, 2018.

[39] Ritchie Zhao, Weinan Song, Wentao Zhang, Tianwei Xing, Jeng-Hau Lin, Mani Srivastava, Rajesh Gupta, and Zhiru Zhang. Accelerating binarized convolutional neural networks with software-programmable fpgas. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 15–24, 2017.

[40] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.

[41] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8697–8710, 2018.