DEGREE PROJECT IN INFORMATION AND COMMUNICATION TECHNOLOGY,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, *SWEDEN 2019*

Accelerating CNN on FPGA

An Implementation of MobileNet on FPGA

YULAN SHEN

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Accelerating CNN on FPGA: An Implementation of MobileNet on FPGA

YULAN SHEN

Master in Embedded Systems
Date: September 5, 2019
Supervisor: Masoumeh Ebrahimi
Examiner: Johnny Öberg
School of Electrical Engineering and Computer Science
Host company: Synective Labs AB
Swedish title: Accelerera CNN på FPGA: Implementera MobileNet på FPGA

Abstract

Convolutional Neural Network is a deep learning algorithm that brings revolutionary impact on computer vision area. One of its applications is image classification. However, problem exists in this algorithm that it involves huge number of operations and parameters, which limits its possibility in time and resource restricted embedded applications. MobileNet, a neural network that uses separable convolutional layers instead of standard convolutional layers, largely reduces computational consumption compared to traditional CNN models. By implementing MobileNet on FPGA, image classification problems could be largely accelerated.

In this thesis, we have designed an accelerator block for MobileNet. We have implemented a simplified MobileNet on Xilinx UltraScale+ Zu104 FPGA board with 64 accelerators. We use the implemented MobileNet to solve a gesture classification problem. The implemented design works under 100MHz frequency. It shows a 28.4x speed up than CPU (Intel(R) Pentium(R) CPU G4560 @ 3.50GHz), and a 6.5x speed up than GPU (NVIDIA GeForce 940MX 1.004GHz). Besides, it is a power efficient design. Its power consumption is 4.07w. The accuracy reaches 43% in gesture classification.

Keywords
CNN, FPGA acceleration, Deep Learning, MobileNet, Image classification, Computer vision

Sammanfattning

CNN-Nätvark är en djupinlärning algoritm som ger revolutionerande inverkan på datorvision, till exempel, bildklassificering. Det finns emellertid problem i denna algoritm att det innebär ett stort antal operationer och parametrar, vilket begränsar möjligheten i tidsbegränsade och resursbegränsade inbäddade applikationer. MobileNet, ett neuralt nätverk som använder separerbara convolution lager i stället för standard convolution lager, minskar i stor utsträckning beräkningsmängder än traditionella CNN-modeller. Genom att implementera MobileNet på FPGA kan problem med bildklassificering accelereras i stor utsträckning.

Vi har utformat ett acceleratorblock för MobileNet. Vi har implementerat ett förenklat MobileNet på Xilinx UltraScale + Zu104 FPGA-kort med 64 acceleratorer. Vi använder det implementerade MobileNet för att lösa ett gestklassificeringsproblem. Implementerade designen fungerar under 100MHz-frekvens. Den visar en hastighet på 28,4x än CPU (Intel (R) Pentium (R) CPU G4560 @ 3,50 GHz) och en 6,5x snabbare hastighet än GPU (NVIDIA GeForce 940MX 1,004GHz). Det är en energieffektiv design. Strömförbrukningen är 4,07w. Noggrannheten når 43% i gestklassificering.

Keywords
CNN, FPGA acceleration, Deep Learning, MobileNet, Image classification, Computer vision

Acknowledgements

The thesis is supported by Synective Labs. Many thanks to Gunnar Stjernberg, Viktor Wase and Roland Stenholm and Niklas Ljung.

I would like to acknowledge Zekun Du for his work in deciding the neural network structure and the training of the neural network.

Many thanks to Masoumeh Ebrahimi for her supervision and direction in the thesis work.

Stockholm, July, 2019

Yulan

List of Figures

List of Tables

List of Abbreviations

AXI	Advanced eXtensible Interface
CLB	Configurable Logic Block
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DSP	Digital Signal Processing
FF	Flip Flop
FPGA	Field Programmable Gate Array
fps	Frame Per Second
GOPS	Giga Operations Per Second
GPU	Graphic Processing Unit
HLS	High Level Synthesis
I/O	Input/Output
MAC	Multiply Accumulator
PL	Programmable Logic
PS	Processing System
RAM	Random Access Memory
ReLU	Rectified Linear Units
ROM	Read Only Memory
RTL	Register Transfer Level
SELU	Scaled Exponential Linear Unit
WHS	Worst Hold Slack
WNS	Worst Negative Slack

Contents

Chapter 1

Introduction

1.1 Background

Convolutional Neural Network (CNN) is a deep learning algorithm that recently have brought revolution in computer vision area. CNN shows excellent performance in solving complex computer vision problems including image classification [6] [7], object detection [7] [8], semantic segmentation [8] and image retrieval [9].

The wide usage of CNN in computer vision brings a rising interest in applying the algorithm to portable and real-time applications. However, the high performance of CNN algorithms comes at the price of large computational consumption. The algorithm involves large number of parameters and large number of mathematical operations, which brings challenges to time and space restricted applications. One optimization method is model simplification. Strategies such as pooling, pruning [10], variations of traditional CNN algorithms such as MobileNet [5] [11], ShuffleNet [12], CondenseNet [13] are developed in order to optimize resource usage. On the other hand, more and more applications turn to high performance hardware platforms. Graphics Processing Units (GPU) and Field-Programmable Gate Array (FPGA) stand out for their ability in doing massive parallel operations.

Both GPU and FPGA are growing fast in artificial intelligence acceleration area. GPU is now dominating the market as it has less engineering cost and it goes into market early. However, compared to GPU, FPGA has several outstanding features that make it a rising star in accelerating deep learning algorithms. The first is flexibility. FPGA allows engineers to reconfigure underlying hardware architecture, even down to bit-level. It is a competitive feature when lower precision deep learning algorithms, such as binary neural

network [14] and ternary neural network [15], are being explored by many people today. The second is low latency. Latency of FPGAs is at the magnitude of nanoseconds while it is microseconds for GPUs. The third is high power efficiency. Xilinx Virtex Ultrascale+, FPGA board produced by Xilinx, has general purpose compute efficiency of 277 GOP/s/W, while NVidia Tesla P4, GPU produced by Nvidia, the efficiency is of 208 GOP/s/W [16]. Both hit the market in 2017.

1.2 Purpose

The master thesis project aims at empowering CNN algorithms with less computational cost, faster speed and higher power efficiency, making the powerful and useful algorithm possible to be applicable to time and resource restricted applications.

Due to its characteristics of flexibility, low latency, and high power efficiency, FPGA is an ideal platform for neural network acceleration. Researches have proved that by applying CNN on FPGA, faster speed and higher power efficiency could be achieved. Many researches have put up their FPGA solutions to accelerate CNN. However, most researches focus only on the performance of FPGA accelerated CNN, but few of them have been used to solve realistic problems. In the master thesis project, not only the performance is discussed, but also the FPGA solution is applied to a realistic problem.

1.3 Goals

In this project, the following goals will be reached.

• Propose an FPGA-based acceleration solution to accelerate MobileNet [5] [11].

• Implement the proposed solution on Xilinx Zynq UltraScale+ MPSoC ZCU104 FPGA board to demonstrate the advantage of FPGA acceleration.

• Analyse the resource usage of the FPGA solution and make a comparison in performance and efficiency among FPGA solution, CPU solution and GPU solution.

• Use the implemented neural network to solve a specific problem – gesture classification.

1.4 Organization of the Thesis

The master thesis is organized in the following structure:

• Chapter 1 describes the problem.

• Chapter 2 is background. It introduces the structure of CNN and MobileNet. A comparison in resource usage between standard CNN and MobileNet is made. It includes an introduction to FPGA. It also includes related works, including researches of resource optimized CNN models, other implemented designs that accelerate CNN on FPGA, and the advantage of using FPGA to accelerate CNN.

• Chapter 3 presents the block design of the MobileNet in detail.

• Chapter 4 presents the implementation result of the block design on the target hardware platform.

• Chapter 5 introduces how the implemented MobileNet is used to solve gesture classification problem.

• Chapter 6 includes conclusion and future work.

Chapter 2

Background

2.1 CNN

Convolutional Neural Network is a deep learning algorithm that shows great capability in image classification. CNN extract features of images by convolution and use the features to classify objects. It is designed to automatically and adaptively learn spatial hierarchies of features [17] through training. An image can be classified when the features vote for the most possible class that the image belongs to.

Deep learning algorithms are deployed to two phases, one is training and another is inference. As a supervised learning algorithm, CNN uses a set of labeled images to train the network. Training process implements back-propagation algorithm that updates the parameters in CNN. After the model has been fine tuned and well trianed, the learned model will be used to classify new samples. It is known as inference. The structure and parameters of a neural network is fixed once the training process has done, while inference is implemented every time a new data sample comes. Therefore, the acceleration of the inference phase is mainly discussed.

2.1.1 Standard CNN

CNN is structured by layers. In an image classification problem, we expect an image as an input layer and values representing the possibility of different classes as an output layer. Between the input layer and the output layer, there are multiple hidden layers. The hidden layers include convolutional layers, activation function, pooling layers, fully connected layers etc. An illustration is shown in Figure 2.1.

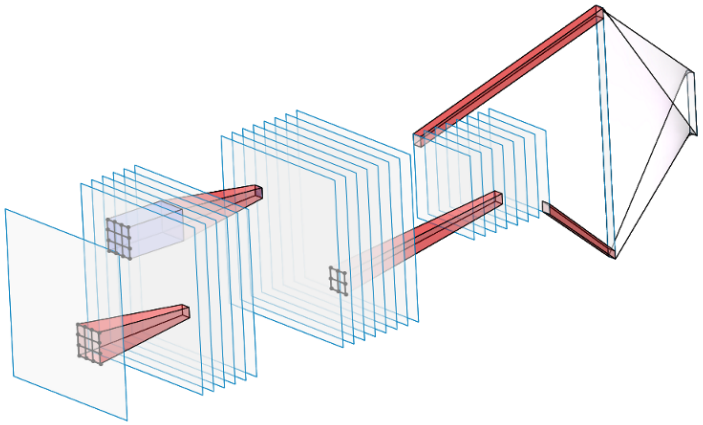


Figure 2.1: Structure of Convolutional Neural Network

Convolutional Layer

A convolutional layer has M input channels and N output channels. Each input channel contains a feature map sized $W_f \cdot H_f$. The $M \cdot W_f \cdot H_f$ input convolves with a convolution kernel sized $M \cdot W_k \cdot H_k$ and produces a $W_f \cdot H_f$ output feature map in one of the output channels. Figure 2.2 shows a convolution with a single kernel. In the convolution kernels are trained weights of the neural network. Convolution with N such kernels produces an output sized $N \cdot W_f \cdot H_f$.

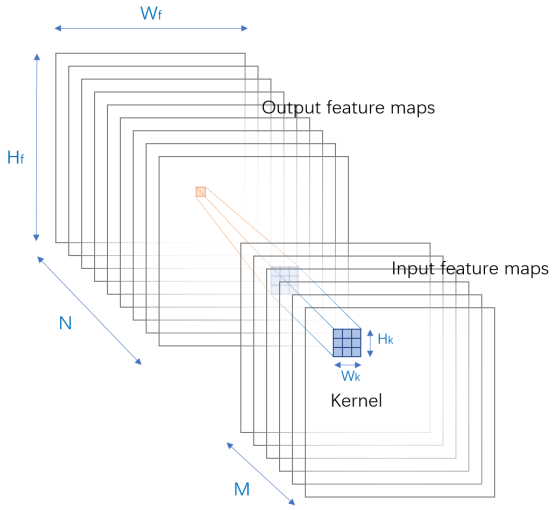


Figure 2.2: Structure of Standard Convolutional Layer

W_f is the feature map width, and H_f is the feature map height. W_k is the kernel width, and H_k is the kernel height. For each pixel in input C and output G, the expression is shown in Equation 2.1, where K represents the convolution kernel.

$$G[n,x,y] = \sum_{i=-\frac{W_k}{2}}^{\frac{W_k}{2}} \sum_{j=-\frac{H_k}{2}}^{\frac{H_k}{2}} \sum_{m=0}^{M-1} C[m, x+i, y+j] \cdot K[n,i,j] \tag{2.1}$$

Padding The size of output feature maps will shrink due to convolution. Therefore, padding is used in order to keep the size of output feature maps the same as the input. The idea is to attach values around the input feature map. There are several padding styles distinguished by the values attached around the boundary. Zero padding pads the inputs with zeros. Reflection padding pads with reflection of the input boundary. Replication padding pads with replication of the input boundary. Figure 2.3 shows an example with different padding styles.

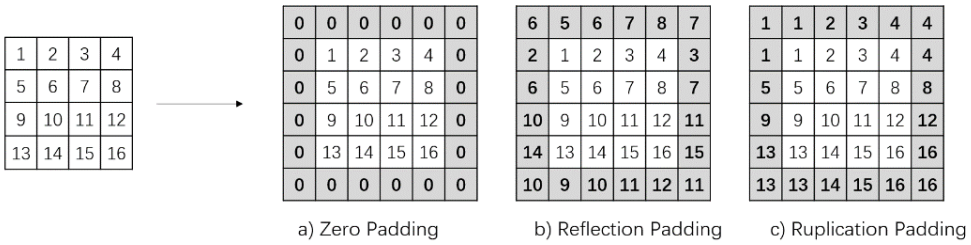


Figure 2.3: Different Padding Styles in Convolution

Stride Stride defines how many steps the convolution kernel will jump over when shifting. It also indicates the factor by which to downscale. Figure 2.4 shows a 2D convolution with a 3×3 convolution kernel and with stride of 2. The size of the output feature map is downsacled by the factor of the stride in both dimension.

Pooling Layer

Pooling layer, also called subsampling layer, is used to reduce the spatial size of feature maps as well as to reduce the number of parameters and mathematical

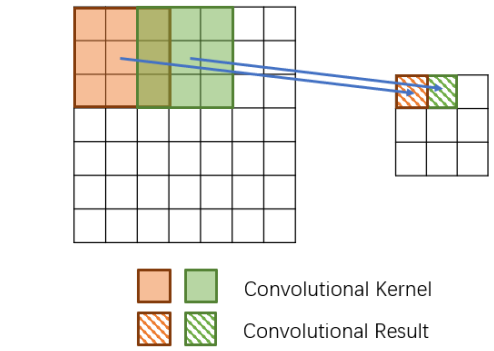


Figure 2.4: Stride-2 3×3 Convolution

operations in the network [18]. Pooling layer could be configured in different styles by defining its pooling window size, stride and method. Methods include max pooling and average pooling. For example, 2×2 max pooling with stride of 2 is commonly used in convolutional neural networks. It is shown in Figure 2.5. It separates the feature map into several 2×2 non-overlapping rectangles and takes the maximum value in each rectangle. The output size is reduced to $\frac{1}{4}$ of input size as a result.

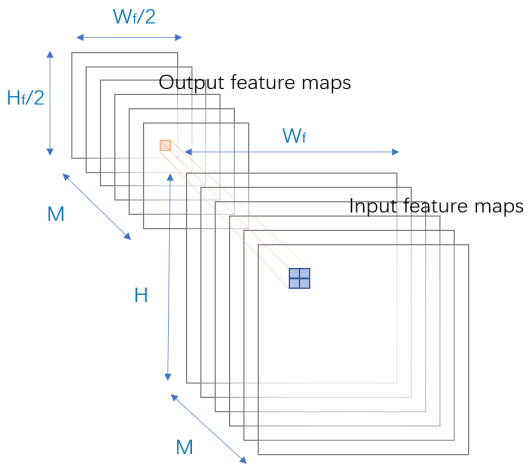


Figure 2.5: Structure of Pooling Layer

Activation

A differentiable and nonlinear function is applied to the feature map and then the result is sent to the subsequence layer as input. The function is called activation function. Activation introduces nonlinearity to the network. It aids the learning of high order polynomials [26] so that the network can learn and perform a more complex task.

Common activation functions are Sigmoid (Equation 2.2), Rectified Linear Units (ReLU) (Equation 2.3), Scaled Exponential Linear Unit (SELU) (Equation 2.4) etc.

$$Sigmoid(x) = \frac{1}{1+e^{-x}} \tag{2.2}$$

$$ReLU(x) = \begin{cases} x & x > 0 \\ 0 & x \leq 0 \end{cases} \tag{2.3}$$

$$SeLU(x) = \lambda \begin{cases} x & x > 0 \\ \alpha(e^x - 1) & x \leq 0 \end{cases} \tag{2.4}$$

Fully Connected Layer

In a fully connected layer, the feature map of the previous layer is flattened to linear structure. Each unit in the feature map acts as a neuron and has full connections to all neurons in the next layer. In a fully connected layer with M input neurons and N output neurons, the connection is shown in Figure 2.6.

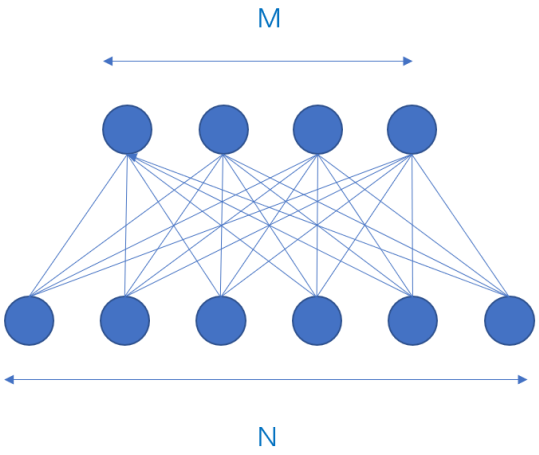


Figure 2.6: Structure of Fully Connected Layer

For each neuron in input X and output Y, the expression is shown in Equation 2.5, where W represents the weight of each connection, and B represents the bias of each output neuron.

$$Y[n] = \sum_{m=0}^{M-1} X[m] \cdot W[m,n] + B[n] \tag{2.5}$$

2.1.2 MobileNet

MobileNet is a variation of Convolutional Neural Network. It is an efficient model for mobile and embedded vision applications. It uses less parameters and less mathematical operations yet maintains reasonable accuracy compared to traditional CNNs. The idea is to use separable convolutional layers to take the place of standard convolutional layers. M kernels sized $M \cdot W_k \cdot H_k$ used in traditional CNN is replaced by M kernels sized $W_k \cdot H_k$ and N kernels sized $M \cdot 1 \cdot 1$, as shown in Figure 2.7. Despite the convolutional layer, pooling layer, activation and fully connected layer in MobileNet are the same with traditional CNN.

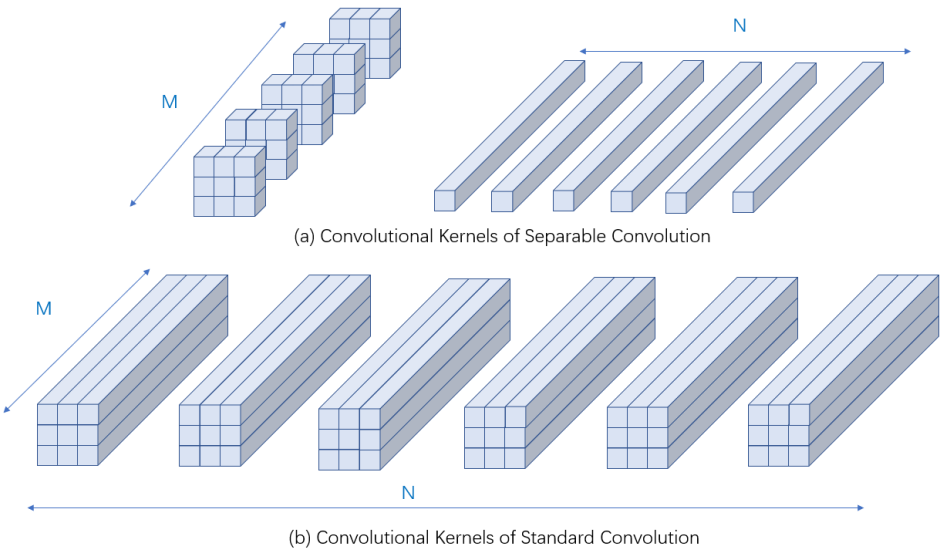


Figure 2.7: Convolution Kernels of Separable Convolution(a) and Standard Convolution(b)

Depthwise Convolutional Layer

The convolution with the $W_k \cdot H_k$ kernel is called depthwise convolution. In a depthwise convolution, the number of input channels and the number of output channels are the same. The $W_f \cdot W_f$ feature map in each input channel convolves with a $W_k \cdot H_k$ convolution kernel and produces an output feature map sized $W_f \cdot W_f$ to the corresponding output channel. Therefore, the input size and the output size keep the same in a depthwise convolutional layer. An illustration of depthwise convolution is shown in Figure 2.8.

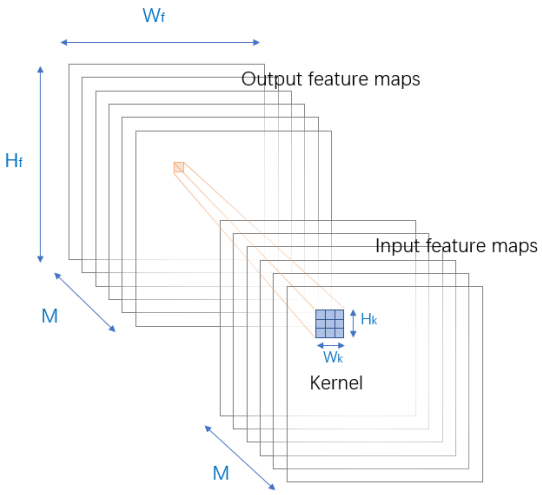


Figure 2.8: Structure of Depthwise Convolutional Layer

For each pixel in input C and output G of a depthwise convolutional layer, the expression is shown in Equation 2.6, where K represents the kernel.

$$G[m,x,y] = \sum_{i=-\frac{W_k}{2}}^{\frac{W_k}{2}} \sum_{j=-\frac{H_k}{2}}^{\frac{H_k}{2}} C[m,x+i,y+j] \cdot K[m,i,j] \tag{2.6}$$

Pointwise Convolutional Layer

The convolution with the $M \cdot 1 \cdot 1$ kernel is called pointwise convolution. In a pointwise convolution, there are N kernels sized $M \cdot 1 \cdot 1$. The $M \cdot W_f \cdot H_f$ input convolves with one kernel and produces a $W_f \cdot H_f$ feature map in one of the output channels, as shown in Figure 2.9. The output is of the size of $N \cdot W_f \cdot H_f$. Pointwise convolutional layer is a special case of standard convolutional layer, where the kernel height and kernel width are both 1.

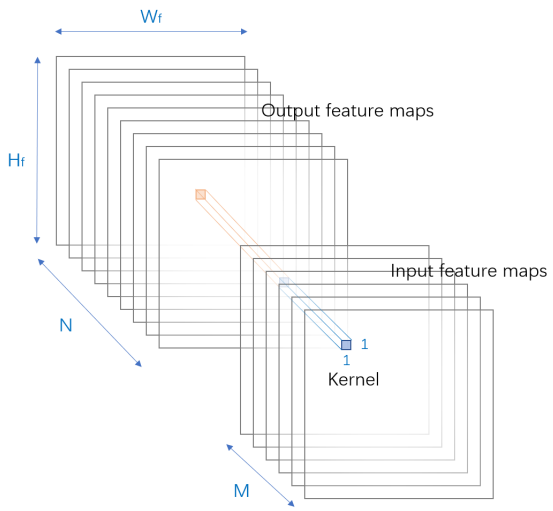


Figure 2.9: Structure of Pointwise Convolutional Layer

For each unit in input C and output G of a pointwise convolutional layer, the expression is shown in Equation 2.7, where K represents the convolution kernel.

$$G[n,x,y] = \sum_{m=0}^{M-1} C[m,x,y] \cdot K[n,m] \tag{2.7}$$

2.1.3 Computational Consumption Analysis

Separable convolutional layer saves computational cost in both time and space dimension compared to standard convolutional layer, which brings an advantage to MobileNet in resource restricted embedded applications. On one hand, separable convolutional layer uses less parameters than standard convolutional layer. Less memory space is required to store the network information. On the other hand, separable convolutional layer reduces the number of mathematical operations, mostly multiply-accumulate operations, which leads to the reduction in time used to process a frame.

Computational Consumption in Standard Convolutional Layer According to Equation 2.1, in a standard convolutional layer, each neuron in the output feature map is a result of $M \cdot W_k \cdot H_k$ multiply-accumulate operations. The output feature map size is $N \cdot W_f \cdot H_f$. Therefore, the total number of multiply-accumulate operations in a single layer is the product of the output size and

the number of MAC operations for each neuron in the output.

$$MAC_{Standard\ Convolution} = M \cdot W_k \cdot H_k \cdot N \cdot W_f \cdot H_f \tag{2.8}$$

Each output channel corresponds to a kernel of the size of $M \cdot W_k \cdot H_k$. Therefore, the number of parameters in a standard convolutional layer is

$$Parameters_{Standard\ Convolution} = N \cdot M \cdot W_k \cdot H_k \tag{2.9}$$

Computational Consumption in Separable Convolutional Layer Separable convolution separates the standard convolutional layer into a depthwise convolutional layer and a pointwise convolutional layer.

In depthwise convolutional layer, each output channel is a convolution result of a $W_k \cdot H_k$ convolution kernel and a $W_f \cdot H_f$ feature map. Therefore, the total number of MAC operations in a depthwise convolutional layer is

$$MAC_{Depthwise\ Convolution} = M \cdot W_k \cdot H_k \cdot W_f \cdot H_f \tag{2.10}$$

The total number of parameters in a depthwise convolutional layer is

$$Parameters_{Depthwise\ Convolution} = M \cdot W_k \cdot H_k \tag{2.11}$$

In pointwise convolutional layer, the convolution kernel is of the size of $M \cdot 1 \cdot 1$. Therefore, each neuron in the output feature map is a result of M multiply-accumulate operations. The total number of multiply-accumulate operations in a single layer is the product of the output feature map size and the number of MAC operations for each neuron in the output feature map.

$$MAC_{Pointwise\ Convolution} = M \cdot N \cdot W_f \cdot H_f \tag{2.12}$$

The total number of parameters in a pointwise convolutional layer is

$$Parameters_{Pointwise\ Convolution} = M \cdot N \tag{2.13}$$

Combining a depthwise convolution and a pointwise convolution is a separable convolution. The result of separable convolution is compared to standard convolution.

$$\begin{aligned} \frac{MAC_{Separable\ Convolution}}{MAC_{Standard\ Convolution}} &= \frac{M \cdot W_k \cdot H_k \cdot W_f \cdot H_f + M \cdot N \cdot W_f \cdot H_f}{M \cdot N \cdot W_k \cdot H_k \cdot W_f \cdot H_f} \\ &= \frac{1}{N} + \frac{1}{W_k \cdot H_k} \end{aligned} \tag{2.14}$$

$$\begin{aligned}\frac{Parameters_{Separable\ Convolution}}{Parameters_{Standard\ Convolution}} &= \frac{M \cdot W_k \cdot H_k + M \cdot N}{N \cdot M \cdot W_k \cdot H_k} \\ &= \frac{1}{N} + \frac{1}{W_k \cdot H_k}\end{aligned} \tag{2.15}$$

As $W_k \cdot H_k$ is usually fixed to 3×3 or 5×5, in most cases, by applying separable convolution could reduce the number of MAC and the number of parameters by 10 to 20.

Computational Consumption in Fully Connected Layer The fully connected layers in CNN are essentially matrix multiplications. A fully connected layer with M input and N output is a $N \cdot M$ matrix multiplied with a $M \cdot 1$ matrix and get a $N \cdot 1$ matrix as a result. The $N \cdot M$ matrix stores the weights for the fully connected layer. The $M \cdot 1$ matrix represents the input neurons and the $N \cdot 1$ matrix represents the output neurons.Therefore,

$$MACFully\ Connected\ Layer = M \cdot N \tag{2.16}$$

$$Parameters_{Fully\ Connected\ Layer} = M \cdot N \tag{2.17}$$

Computational Consumptions of Different Models In Table 2.1, a comparison among popular CNN models is made based on the number of multiply-accumulate operations, the number of parameters, and the accuracy. The accuracy is measured on the ImageNet [19] benchmark. ImageNet [19] is the test dataset for ImageNet Large Scale Visual Recognition Challenge (ILSVRC), an annually held challenge in the accuracy of object detecting. The dataset contains images collected from the Internet and these images are hand annotated to 1000 object categories.

Table 2.1: Computational Cost of Popular CNN structures [4]

	Number of MAC operations	Number of parameters	ImageNet Accuracy
VGG16 [20]	15.5G	138M	71.9%
AlexNet [6]	724M	61M	57.1%
GoogLeNet [21]	1.58G	6.99M	68.7%
ResNet50 [22]	3.86G	25.5M	75.3%

Table 2.2 shows the number of MAC operations and the number of parameters of MobileNet structures. On ImageNet [19] benchmark, 1.0 MobileNet-224 has achieved similar accuracy with relatively large CNNs such as VGG16, GoogleNet and ResNet50, and 0.50 MobileNet-160 has achieved similar accuracy with AlexNet. However, the computational cost is much less than traditional CNN models in both 1.0 MobileNet-224 and 0.50 MobileNet-160. MobileNet shows a capability in reducing both the number of multiply-accumulation operations and the number parameters, while maintaining the accuracy.

Table 2.2: Computational Cost of MobileNet structures [5]

	Number of MAC operations	Number of parameters	ImageNet Accuracy
1.0 MobileNet-224 [5]	569M	4.2M	70.6%
0.50 MobileNet-160 [5]	76M	1.32M	60.2%

2.2 FPGA Platform

FPGA is the abbreviation of Field programmable Gate Array. As indicated by its name, it comprises of an array of programmable logic blocks that are connected via programmable interconnects.

FPGA has three main advantages, flexibility, low latency and high energy efficiency. Therefore, FPGA is widely used in producing highly customizable SoCs, ASIC verification, high performance computing etc.

Modern FPGA evaluation board is usually an integration of Processor System (PS) and Programmable Logic (PL). Processor System is a general purpose system that is usually made by a powerful CPU processor. Programmable Logic contains the reconfigurable resources that is commonly recognized as FPGA resources including Look-up Table (LUT), Digital Signal Processor (DSP), and Block Random Access Memory (BRAM).

2.2.1 FPGA Resources

LUT

Look-up Table (LUT) works as function generators. In Xilinx Ultrascale+ architecture, 6-input LUTs are used. Combining several LUTs and some other components such as flip flops, arithmetic and carry chains, wide multiplexers

forms a Configurable Logic Block (CLB). CLB is the main resource for implementing general-purpose combinatorial and sequential circuits on FPGA.

DSP

Digital Signal Processor (DSP) slices on FPGA could perform various commonly used arithmetic operations. The use of DSP could take advantage of hardware parallelism to provide high data throughput and high efficiency for DSP applications. DSP48E2 is the DSP slice used in Xilinx Ultrascale+ MPSoC devices. The basic functionality is shown in Figure 2.10. One DSP slice could be configured to perform one of the arithmetic operations, including 4-input addition, multiplication, multiply-accumulation and etc. The data width of the input and the output could also be configured. It could be maximum 48 bit. The DSP slice is optimized to have low power consumption, high speed, small size while maintaining its flexibility.

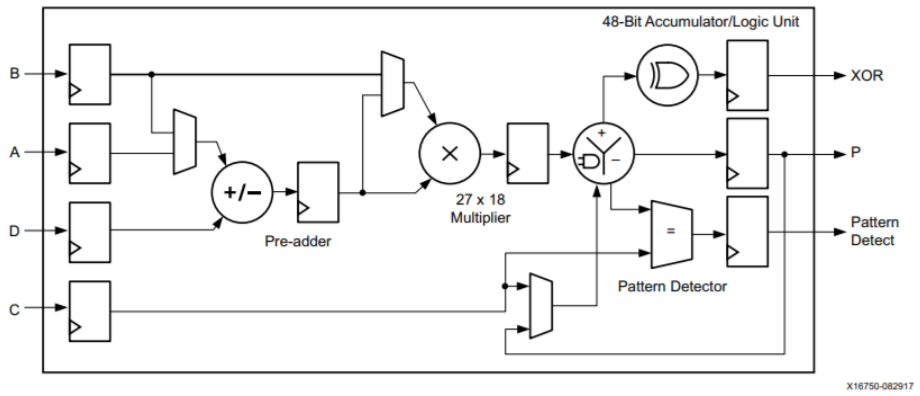


Figure 2.10: Basic DSP48E2 Functionality [1]

BRAM

Block Random Access Memory is the major memory element on FPGA. They are scattered and interleaved with other configurable components like DSP and LUT on FPGA. Close interaction with DSP and LUT offers BRAM great flexibility.

The block RAM in Xilinx UltraScale architecture-based devices stores up to 36 Kbits of data. It could be configured as various memory blocks. It can be either RAM or ROM. It can have either have one port or two ports. The port width and number of lines can be defined by users.

2.3 FPGA accelerated CNN

2.3.1 Resource Optimized CNN Models

Large resource consumption of CNN algorithm is not only a challenge for embedded systems, but also an abandon for PC based applications. Many methods have been explored to reduce the number of parameters and the number of mathematical operations.

Some methods focus on optimization of convolutional layer, as it is the most time and resource consuming layer in CNN algorithm. Factorization is one method. The idea is to use a low rank basis of kernels that are separable in the spatial domain to approximate the 3D kernels used in standard CNN [23]. MobileNet is one of the state-of-art models using separable convolution as introduced in 2.1.2. Another way is to implement the convolution algorithm by FFT. [24] performs convolutions as products in the Fourier domain, and reuses transformed feature maps many times.

Some models are designed for easier parallelization. Group convolution is an example. The concept is first introduced in AlexNet [6]. The convolution channels are separated to several groups and each group has its own set of convolution kernels. Convolution is independent in each group until at a certain depth, the channels will mix up. In this way, the interconnection between layers and layers is reduced and it allows parallelization of the groups. CondenseNet [13] is another group convolution example. Instead of manually setting the groups, it groups the channels by a learning process.

Some minimize the model by cutting off redundant information. Pruning is a strategy used in inference phase by iteratively removing the least important parameters—feature maps in this case—according to heuristic selection criteria [25]. Networks with low precision data format are being explored. In order to downscale the datawidth while maintaining its accuracy, various researches focus on quantization methods. [26] uses low precision dynamic fixed point to train the network. Some models use extreme low precision, as introduced in [14] [15] [27].

Recent models are usually combinations of several strategies. ShuffleNet [12] is a combination of separable convolution and group convolution. SEP-Nets [28] applies binary weights to MobileNet.

2.3.2 Related Works

Due to its complexity, acceleration solutions of traditional CNN are usually aided by high-level language and high-level synthesis tools. [29] uses unrolling and pipelining strategies aided by Vivado HLS. The solution reaches 17.42x speedup than CPU. A solution based on OpenCL FPGA framework reaches 4.14x speedup on VGG model than CPU. [30] Distributed architecture provides another solution in accelerating CNN on FPGA. [31] proposed an energy-efficient model using a deeply pipelined FPGA cluster. Another solution called FPDeep uses FPGA clusters to train CNNs [32].

Besides the challenge of implementing CNN models, allocating memory space to store large number of weights and intermediate results is another challenge. Off-chip memory is a common solution. However, large number of external memory accesses result in long latency and large energy consumption. [33] builds a 2-level memory cache using on-chip memories. The cache hierarchy reduces latency and energy consumption by several orders of magnitude than external memory access. [34] use layer fusion technique to avoid frequent access to external memories. Compared with AlexNet, the fused-layer method gives a 28% savings in off-chip data transfer, even when applied to the first two layers.

In order to reduce complexity and resource usage, low precision CNN algorithms are being explored. By applying low precision data format, less memory space is required to store the weights and intermediate results. Instead of 32-bit floating point number used in most software applications, some hardware solutions adopts 32-bit integer wights, 16-bit integer weights, or even bitwise weights. [35] is a C-based HLS methodology to accelerate Binarized Convolutional Neural Network on FPGA. The solution has reached high throughput with low resource usage. Another solution FP-BNN is presented in [36]. It also reveals the fact that reduction in precision results in accuracy loss. Binarized AlexNet in the design suffers from a 13% accuracy drop compared to the original one.

Since MobileNet is proposed, interest in applying it on FPGA has kept high. Because it involves less parameters and multiply-accumulations than standard CNN, RTL design could be possible to implement for MobileNet. Matrix multiplication engine (MME) array is proposed in [37]. It divides the feature map into several $W_f \cdot H_f \cdot 32$ submatrices. There are arrays of multipliers to ensure parallelization in each submatrix. The network is implemented on Arria 10 SoC and reaches the performance of 266.2 f/s. Another solution is a streaming architecture with two passes [38], one for depthwise convolution

and another for pointwise convolution. Multipliers are shared between passes. It is implemented on Stratix 5GSD8 and reaches the performance of 43.01 f/s.

2.3.3 Advantages of Using FPGA to Accelerate CNN

Flexibility, low latency and high efficiency are three major advantages that FPGA have over CPU and GPU. Flexibility shows in that FPGA allows engineers to reconfigure underlying hardware architecture, even down to bit-level. It is the main characteristic that distinguish FPGA with general purpose hardware platforms. From the perspective of latency, latency of FPGA is at the magnitude of nanoseconds. As for GPU, it is at the magnitude of microseconds. FPGA generally works on relatively slow clock frequency compared to CPU and GPU, but shows better or no worse calculation capacity. Therefore, FPGA could have higher power efficiency. For example, Xilinx Virtex Ultrascale+, FPGA board produced by Xilinx, and NVidia Tesla P4, GPU produced by Nvidia are both hardware platforms that came to market in 2017. The former one has general purpose compute efficiency of 277 GOP/s/W, while the general purpose computer efficiency of the latter one is 208 GOP/s/W [16].

Besides the nature of FPGA platform, CNN is an algorithm that has huge potential in acceleration. CNN involves large number of multiply-accumulation operations that could be parallelized. The calculations in a convolutional layer largely depend the results of previous layer. However, the calculations in the same convolutional layer are mostly independent, which builds the base for mass parallelization. From another perspective, CNN algorithm is now flourishing with the development of numeric new models. Each has its unique datapath, neural network pattern, data format and etc. FPGA could meet the specific requirements of the models and reach rather high bandwidth. For example, BinaryNet [14] is a neural network that uses binarized weights and data. It uses XOR operations to replace multiplication operations in traditional CNN. The network could be implemented by FPGA as it could describe a model precisely in bit-level. However, on general purpose platform like CPU and GPU, low precison data format is supported. Fitting the neural network on CPU or GPU could cost more resources than the binarized neural network actually requires.

2.4 Summary

The structures of traditional CNN and one of its variation MobileNet are introduced in this chapter. It also highlights the difference between standard

CNN and MobileNet. A comparison is also made between standard CNN and MobileNet in resource usage and computational cost. Conclusion is made that MobileNet, a neural network that uses separable convolutional layer instead of standard convolutional layer, uses less parameters and involve less MAC operations than standard CNN.

An introduction to FPGA is made in this chapter. The resource elements on FPGA are introduced.

How people use FPGA to accelerate CNN is introduced in this chapter. It introduces how researches optimize the resource usage of CNN. It also introduces previous work of accelerating CNN on FPGA. It also includes the features of FPGA and explains why FPGA is an ideal platform to perform CNN algorithms.

Chapter 3

Block Design

In this chapter, we have designed a MobileNet accelerator block. We have also designed a system to integrate the accelerators to accelerate a specific MobileNet. The MobileNet can be used to solve image classification problems.

3.1 MobileNet Structure

Images to be classified are 128×128 grey-scale images. They are preloaded on PS. The image data is 8-bit in width. The images will be sent to PL and processed by a predefined MobileNet on PL. When the calculation is done, the classification result will be sent back to PS. The classification result is in form of an array of 32-bit integers. Each integer in the array represents the possibility of one class. The larger the integer is, the more possibly the image belongs to the class.

The structure of the network is listed in Table 3.1. The original MobileNet in [5] is designed for images sized 224×224. It has 14 depthwise convolutional layers, 13 pointwise convolutional layers, 5 pooling layer, and 1 fully connected layer, with maximum 1024 output channels. The network is designed for complex tasks with 1000 classes and it is not designed for embedded applications. Therefore, the MobileNet structure used in this thesis is a compressed one. It is originated from [39]. The compressed MobileNet has an input image sized 128×128 and an output of 6 classes. It has 6 depthwise convolutional layers, 5 pointwise convolutional layers, 4 pooling layers, and 2 fully connected layers, with maximum 64 output channels. In the design, all the depthwise convolutional layers adopt kernels of the size of 3×3. Zero paddings are applied to all depthwise convolutions. All the pooling

layers are 2-stride 2×2 max pooling.

The total number of parameters used in the MobileNet structure is 34256 according to Eq. 2.15, Eq. 2.13 and Eq. 2.17. The total number of multiply accumulate operations is 7.40M according to Eq. 2.10, Eq. 2.12 and Eq. 2.16.

3.2 System Design

The images to be classified are stored in PS. The trained weights used in the network are stored in PL. There are three ROMs in PL, storing the weights of depthwise convolutional layer, the weights of pointwise convolutional layer and the weights of fully connected layer respectively. The network takes the image from PS and takes the weights from ROMs in PL. When the calculation is done in the network, it sends the results back to PS. The block design of the acceleration System is shown in Figure 3.1.

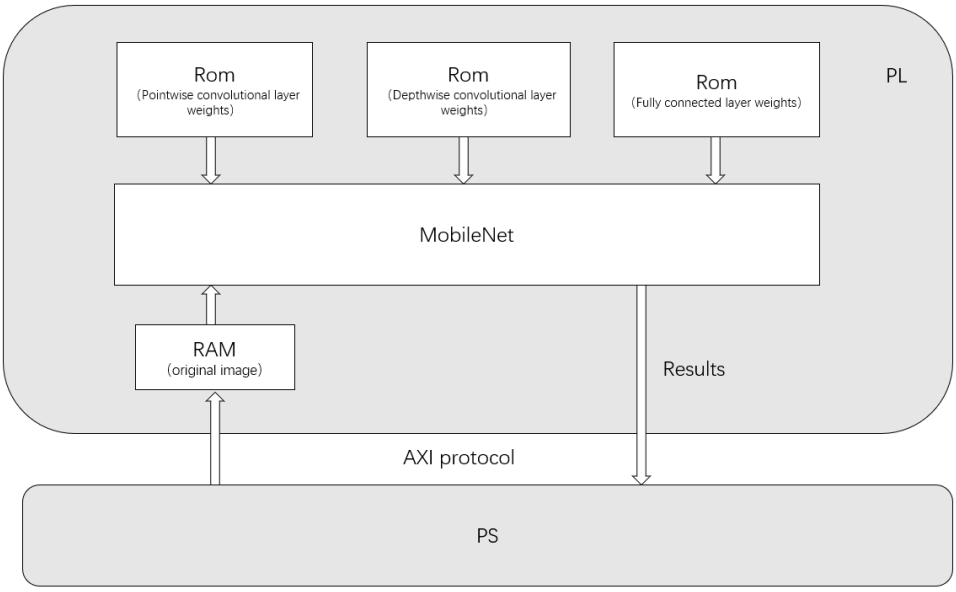


Figure 3.1: Block Design of the Acceleration System

The MobileNet module consists of 64 accelerators and a fully connected layer processor, as shown in Figure 3.2. The calculation in different channels are independent. There are maximum 64 channels in the designed MobileNet. Therefore, parallelization capacity can be fully occupied when 64 accelerators working in parallel. Each accelerator can operate depthwise convolution, pointwise convolution, pooling and ReLU. The structure of the accelerator will

Table 3.1: MobileNet structure to be implemented

Procedure Number	Layer Number	Input Channel	Output Channel	Input Size	Output Size	Execution
Procedure 0	Layer 0	1	8	128*128	128*128	Depthwise Convolution
	Layer 1	8	8	128*128	64*64	Pooling
	Layer 2	8	8	64*64	64*64	ReLU
Procedure 1	Layer 3	8	8	64*64	64*64	Depthwise Convolution
	Layer 4	8	8	64*64	64*64	ReLU
Procedure 2	Layer 5	8	32	64*64	64*64	Pointwise Convolution
	Layer 6	32	32	64*64	64*64	ReLU
	Layer 7	32	32	64*64	32*32	Pooling
Procedure 3	Layer 8	32	32	32*32	32*32	Depthwise Convolution
	Layer 9	32	32	32*32	32*32	ReLU
Procedure 4	Layer 10	32	64	32*32	32*32	Pointwise Convolution
	Layer 11	64	64	32*32	16*16	Pooling
	Layer 12	64	64	16*16	16*16	ReLU
Procedure 5	Layer 13	64	64	16*16	16*16	Depthwise Convolution
	Layer 14	64	64	16*16	16*16	ReLU
Procedure 6	Layer 15	64	64	16*16	16*16	Pointwise Convolution
	Layer 16	64	64	16*16	16*16	ReLU
Procedure 7	Layer 17	64	64	16*16	16*16	Depthwise Convolution
	Layer 18	64	64	16*16	16*16	ReLU
Procedure 8	Layer 19	64	64	16*16	16*16	Pointwise Convolution
	Layer 20	64	64	16*16	8*8	Pooling
	Layer 21	64	64	8*8	8*8	ReLU
Procedure 9	Layer 22	64	64	8*8	8*8	Depthwise Convolution
	Layer 23	64	64	8*8	8*8	ReLU
Procedure 10	Layer 24	64	16	8*8	8*8	Pointwise Convolution
	Layer 25	16	16	8*8	8*8	ReLU

Fully Connected Layer

Procedure Number	Layer Number	Input Neurons	Output Neurons
Procedure 11	Layer 26	1024 (8*8*16)	16
	Layer 27	16	6

be explained in detail in Section 3.3. Data is transferred from accelerator to accelerator through data bus. The data bus is managed by a controller.

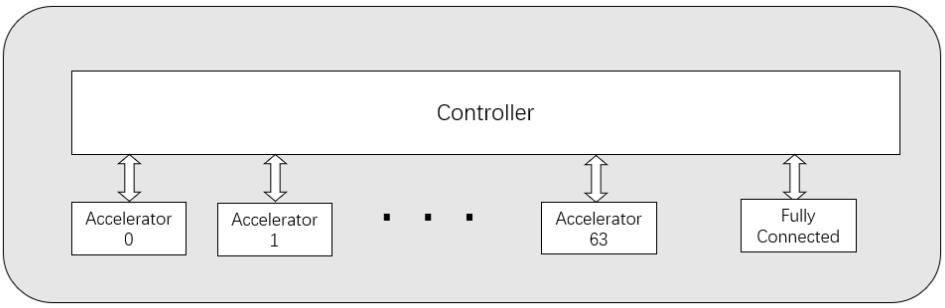


Figure 3.2: Block Design of MobileNet

3.2.1 Memory Allocation

As the MobileNet used in the project is a compressed one, the number of parameters used in the neural network is small enough to be stored in on-chip memories. The weights for depthwise convolution, pointwise convolution and fully connected layers are stored separately in 3 ROMs. All the weights are 8-bit signed integers. The size of the ROMs are listed in Table 3.2.

ROM_{Dep} is the ROM for depthwise convolution, All the weights for a single layer are stored in a line. There are 6 depthwise convolutional layer in the design and the maximum number of parameters in a layer is 576. Therefore, it is 4608 bit in width and has 6 lines. If the number of parameter in a layer is less than 576, zeros are filled in blank space.

ROM_{Pnt} is the ROM for pointwise convolution, all the weights for a single layer are stored in a $N \cdot M$ array. Because the maximum N in the design is 64, the weight array is fit in a $64 \cdot M$ array. If the number of output channel is less than 64, the blank space is filled with zero. Therefore, the ROM is 512-bit in width and have 232 lines.

The first fully connected layer has 1024 input neurons and 16 output neurons. Weights for the first fully connected layer are stored in a 16×1024 array. The second fully connected layer has 16 input neurons and 6 output neurons. Weights are stored in a 6×16 array. ROM_{FC}, the ROM for the weights of fully connected layer, is 128-bit in width and have 1040 lines.

There is a RAM for the storage of original images. It is 8-bit in width and have 16384 lines as the image is 128×128 in size.

Table 3.2: Size of ROMs in System Design

	Width(bit)	Lines	Size(kB)
ROM_{Dep}	4608	6	3.375
ROM_{Pnt}	512	232	14.5
ROM_{FC}	128	1040	16.25

Inside each accelerator in Figure 3.2, there is a RAM to store intermediate result. It will be explained in detail in Section 3.3.

3.2.2 Data Transaction

Data transaction Between PL and PS

In the system, the image to be classified should be transferred from PL to PS and the classification result should be transferred from PL to PS after calculation.

Data transaction between PL and PS adopts Advanced eXtensible Interface (AXI) protocol [2]. AXI protocol ensures high speed data transformation from point to point.

AXI Interface consists of five channels: Read Address Channel, Write Address Channel, Read Data Channel, Write Data Channel, and Write Response Channel. Different types of messages are transmitted separately via independent channels. A handshake-like procedure is required before all transmissions in each channel. A valid signal represents that the address or the data from the source is valid. A ready signal indicates that the terminal is ready for receiving messages.

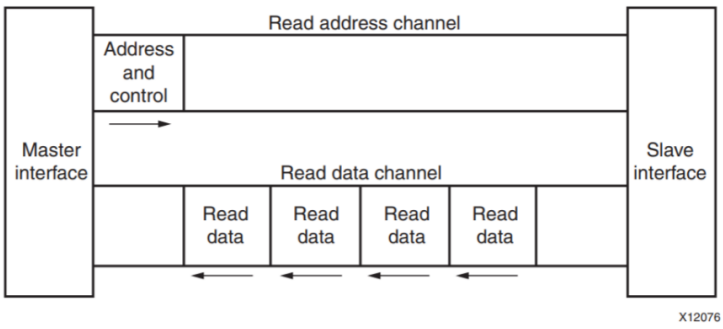


Figure 3.3: AXI Read [2]

As shown in Figure 3.3, in a read operation, the master sends a read address valid signal and read address to the slave when the read address channel is ready. Then the slave will send read data back through read data channel.

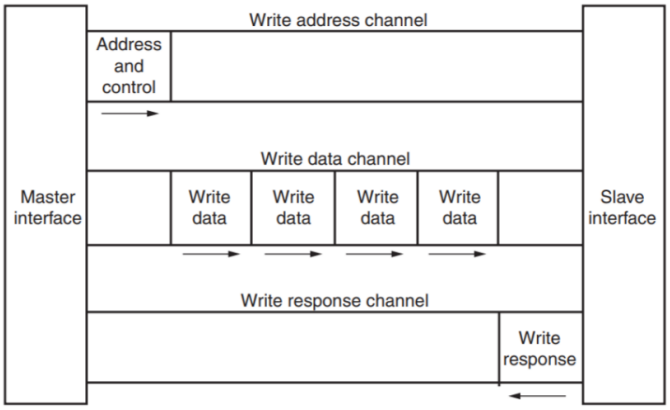


Figure 3.4: AXI Write [2]

As shown in Figure 3.4, in a write channel, the master sends a write address valid signal and write address to the slave when the write address channel is ready. It also sends a write data valid signal and write data to the slave when the write data channel is ready. Then the slave will send a write response back to the master, indicating whether the write operation is successful or not.

In the thesis project, data bandwidth between PL and PS in this system is 32-bit. Read requests are sent from PL side. Data read is done in a read burst mode, meaning that 16 32-bit will be sent consecutively from PS to PL, and each 32-bit data takes one clock cycles. Once a read burst is done, PL will send another read request, until the whole image has been transferred. 256 read requests are needed to transfer a 128×128 image. In a write process, when the results are produced by MobileNet. PL will send a write request to PS. If PS side has prepared to receive data, data will be sent to PS in a write burst mode. 16 32-bit data will be sent consecutively to PS.

Data Transaction Inside PL

Data is transferred inside PL through data bus. As shown in Figure 3.2, data transaction inside PL is organized by a data bus controller.

Data bus controller is responsible for all internal data transaction in the network. The data bus controller has three major functions.

The first is to inform the accelerators with layer information. An accelerator in the network requires layer information to choose correct data path,

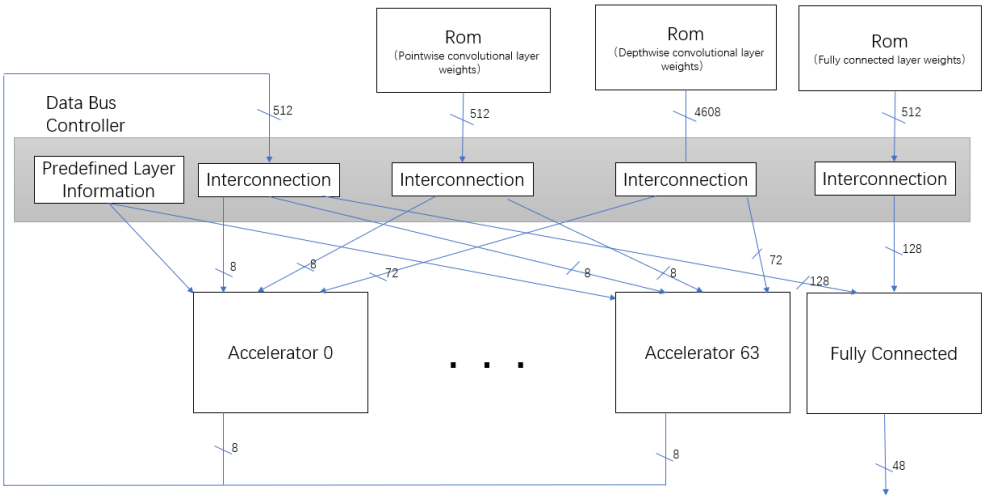


Figure 3.5: Structure of Datapath Controller

weights and corresponding input data to do calculation. Layer information includes layer number, number of input channels, number of output channels and feature map size. Data bus controller is responsible for providing all information that the accelerators require. It sends control instructions indicating layer structure to the accelerators when a layer execution is going to start, so that the accelerator could work properly.

The second, the data bus controller is responsible for data transaction from ROM to the accelerators. Weights for depthwise convolution, pointwise convolution and fully connected layer are stored in separate ROMs. The data bus controller reads weights from correct ROM and sends them to corresponding accelerators.

The third, the controller is in charge of data transaction between accelerators. It requires data from the RAM in the accelerators which contains the feature map data of the previous layer and send the data to the accelerators that are in need of the data.

The interconnection between data bus controller, the accelerators and the ROMs is shown in Figure 3.5.

3.3 Accelerator Design

As shown in Figure 3.6, an accelerator in the network is composed of a RAM to store intermediate results, a depthwise convolution block, a pointwise con-

volution block, a pooling block, and a ReLU block.

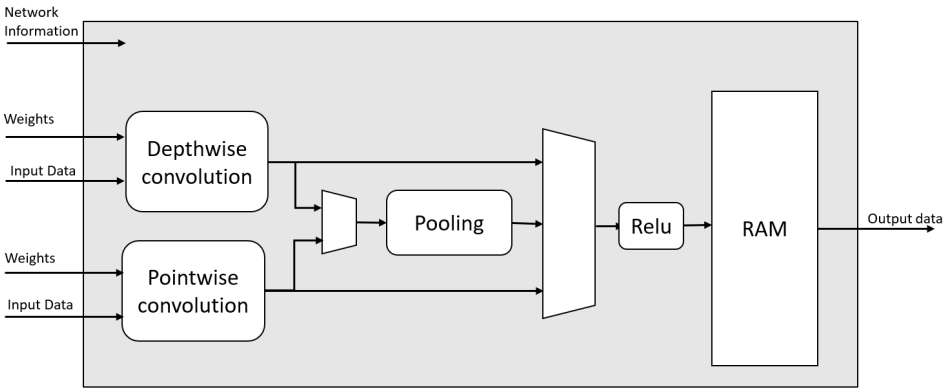


Figure 3.6: Block Design of an Accelerator in MobileNet

The accelerator can cope with four procedures corresponding to four datapaths: depthwise convolution followed by ReLU, depthwise convolution followed by pooling and then ReLU, pointwise convolution followed by ReLU, and pointwise convolution followed by pooling and then ReLU. Datapath is chosen depending on layer information passed to the accelerator.

3.3.1 Depthwise Convolution Block

Depthwise convolution block is responsible for depthwise convolution in MobileNet. The structure is shown in Figure 3.7

A feature map is read and sent to depthwise convolution block pixel by pixel from left to right and from top to down. A result buffer that covers 2 complete lines and the first 3 pixels of a third line is used to store the temporary result. Every time a pixel is shifted in, it is multiplied by 9 kernel weights in the 3×3 convolution window and then added to corresponding registers in the result buffer. It is done by 9 MAC blocks. The result is then shifted out. It will be passed to the next block for further calculation.

The weights are 8-bit signed integers and the feature map pixels are 8-bit unsigned integers. The MAC block performs the multiply-accumulate operation of the two and get an output of 16-bit signed integer. The result buffer that the stores the temporary result is also 16-bit in width. The maximum feature map size is 128×128. Therefore, the length of the result buffer is 259. However, as the neural network has different feature map sizes for different

layers. The depthwise convolution block can adapt the use of the shift registers according to input feature map size. In this design, feature maps with the size of 128×128, 64×64, 32×32, 16×16 and 8×8 can be processed in the depthwise convolutional block. For example, with the feature map size of 64×64, only 131 of the shift registers are used.

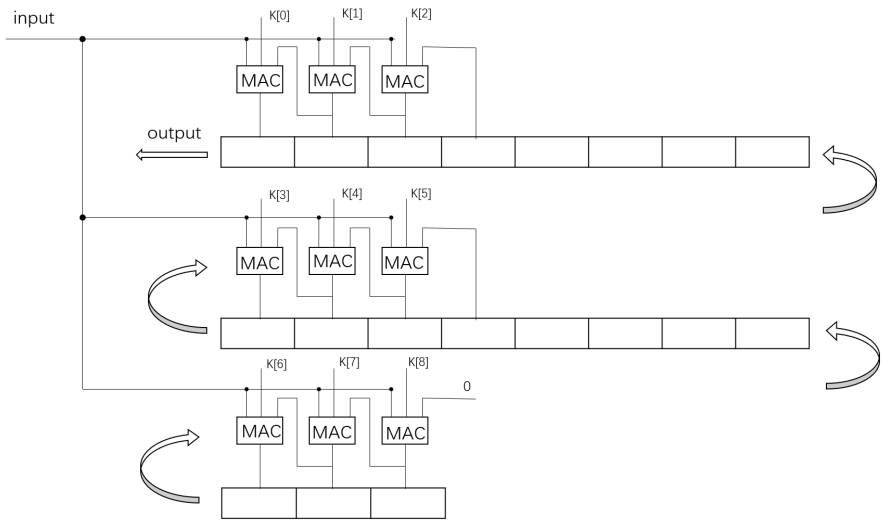


Figure 3.7: Structure of Depthwise Convolution Block

Padding

Zero padding is applied to depthwise convolution. Padding in the top row and the bottom row can be implemented by inserting zeros as inputs. For the padding at the left most column and the right most column, when the input pixel is on the left or right edge of the feature map, the corresponding weights are multiplied with zeros but not the input pixel.

3.3.2 Pointwise Convolution Block

In a pointwise convolutional layer with M input channels, a result pixel of output feature map is the sum of products of M different pixels and their corresponding weights. M is 64 in this MobileNet structure. Therefore, two buffers are used in the pointwise convolution block, one for the weights from ROM, one for the data to be convolved. The buffer sizes are both 8-bit in width and 64 in length in this case. Since there are limited DSPs on the FPGA board, the MAC operations are divided into 8 groups. Each group employs a

MAC block for calculation and a buffer for the storage of intermediate result. The result of one pixel in the output is the sum of the multiply accumulation results of the 8 groups. The MAC block in pointwise convolution block has the same configuration with the MAC used in depthwise convolution block and the buffers for intermediate results are also 16-bit in width.

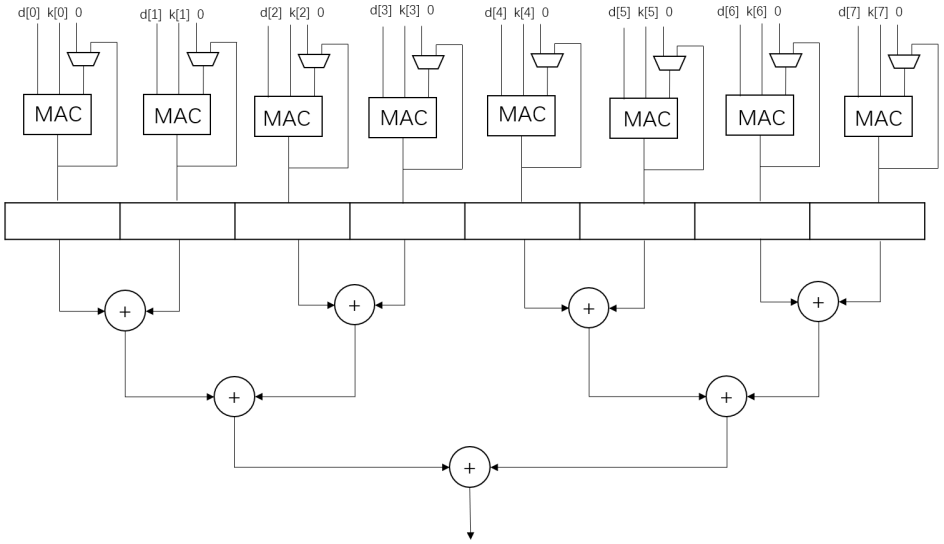


Figure 3.8: Structure of Pointwise Convolution Block

As shown in Figure 3.8, eight MAC blocks are placed in a pointwise convolution block. The weights are preloaded from ROM before the start of the calculation. The data to be convolved is taken from the RAMs for the results of previous layer. When doing calculation, the correct data are taken from the weight buffer and the data buffer, and then sent to MACs. The result of each group comes out every 8 clock cycles. Then they are added together. The addition of the 8 results adopts tree adder structure. They are added two by two until the output is the sum of 8 data. The output is sent to subsequence block for further calculation.

3.3.3 Pooling Block

Like depthwise convolution, shift registers are used to store part of the feature map. The shift register covers the first line and the first two pixels of the second line and it is 16-bit in width. As the maximum feature map size in the design is 128×128. The shift register is 130 in length. When the shift register is

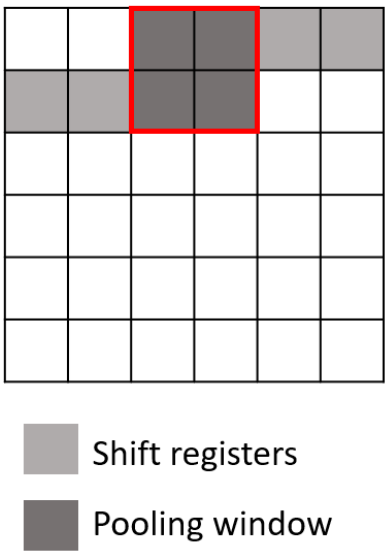


Figure 3.9: Shift Registers in Pooling Block

filled with corresponding data, the pooling result is the maximum value in the pooling window.

A Mux is used to choose from different sizes of the register buffer, since the neural network has different feature map sizes for different layers. Feature maps with the size of 128×128, 64×64, 32×32 and 16×16 can be processed in pooling block.

3.3.4 ReLU Block

In a ReLU block, it has an input of 16-bit signed integer and an output of 16-bit signed integer. If the input value is less than zero, the output is zero, otherwise, the output is the same as input.

3.3.5 RAM Block

The RAM in the accelerator stores the intermediate result produced. The RAM is able to store 4096 8-bit data. As the output data of a ReLU block is 16-bit in width, it should be normalized to 8-bit in order to fit in the RAM.

3.4 Fully Connected Layer Processor

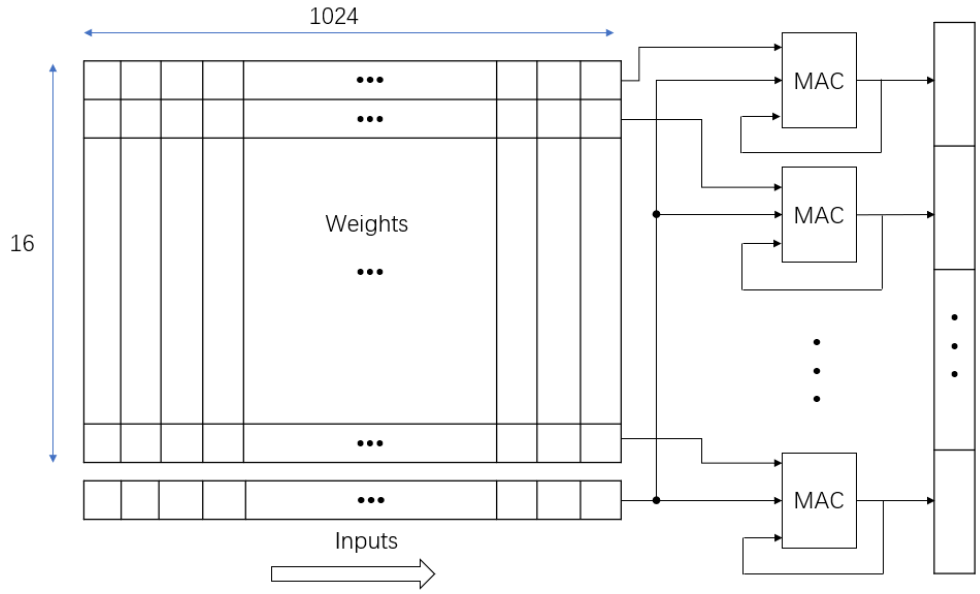


Figure 3.10: Hardware Structure of the First Fully Connected Layer

Fully connected layer processor is specially designed for the fully connected layers used in Table 2.2. The first fully connected layer is essentially a 16×1024 matrix multiplied with a 1024×1 matrix and the result is a 16×1 matrix, and the second fully connected layer is a 6×16 matrix multiplied with the 16×1 matrix and the result is a 6×1 matrix. Therefore, the fully connected layer processor is consisted of 16 multiply accumulators. The multiply accumulators are shared in both the first and the second fully connected layer. In the first fully connected layer with 1024 inputs and 16 outputs, 16 multiply accumulators work in parallel. It takes weights from ROM and input data from previous layer. The results are produced after 1024 clock cycles. In the second fully connected layer with 16 inputs and 6 outputs, 6 of the 16 multiply accumulators are working in parallel. The results of the first fully connected layer are used as input in the second fully connected layer. The results are produced after 16 clock cycles. An illustration of the first fully connected layer is shown in Figure 3.10.

3.5 Summary

A block design of the MobileNet presented in Table 2.2 is introduced in this section. However, the structure is not only designed for the specified network structure, but also flexible to other network structures that are within the hardware capability and design limits.

Since the network is composed of arrays of accelerators, one could add or decrease the number of accelerators according to the network structure with little effort.

All calculation blocks in the accelerator, including depthwise convolutional block, pointwise convolutional block and pooling block, support feature map of different sizes. The feature map size could choose from 128×128, 64×64, 32×32, 16×16 and 8×8.

It also allows users to choose from different processing procedures, since the accelerator provide data path among depthwise convolution followed by ReLU, depthwise convolution followed by pooling and then ReLU, pointwise convolution followed by ReLU, or pointwise convolution followed by pooling and then ReLU. The order of the procedures could be rearranged to fit other MobileNet structures.

Chapter 4

Hardware Implementation and Results

The designed MobileNet is implemented on Xilinx Vivado. Simulation, synthesis, implementation and bitstream generation are the four steps that can turn RTL source code to running application. Simulation simulates the behavior of the RTL blocks. Synthesis transforms an RTL design into a gate-level netlist. [40]. Implementation is to place and route the netlist onto device resources, within the logical, physical, and timing constraints of the design [41]. Bitstream generation generates the bitstream that can be read and programmed by the target FPGA board according to the implementation results.

4.1 FPGA Platform

The hardware platform in the project is Xilinx Zynq UltraScale+ ZU104. The board is populated with the Zynq UltraScale+ XCZU7EV-2FFVC1156 MPSoC, which integrates a powerful processing system (PS) and programmable logic (PL) in the same device. The PS features the Arm® flagship Cortex®-A53 64-bit quad-core processor and Cortex-R5 dual-core real-time processor [3]. The appearance of the board is shown in Figure 4.1.

4.2 Resource Utilization

One of the characteristics of embedded systems is that it is resource limited. Therefore, all designs on FPGA should not exceed its resource limit.

In Xilinx Zynq UltraScale+ ZU104, reconfigurable components include

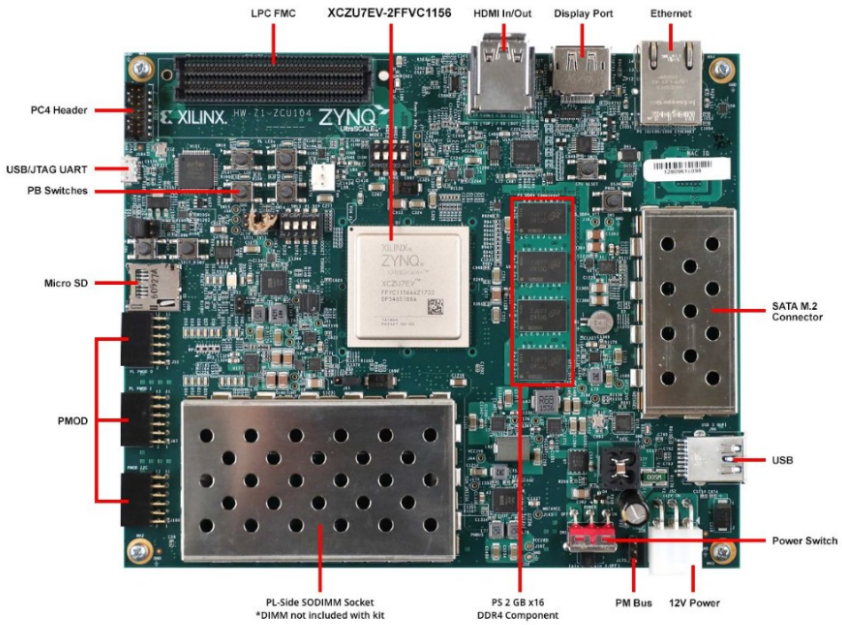


Figure 4.1: Zynq UltraScale+ ZU104 Appearance [3]

504K system logic cells, 461K CLB flip-flops, 11Mb block RAM, 27Mb UltraRAM and 1723 DSP slices.

Table 4.1 shows the resource usage for the designed MobileNet after implementation. None of the resources exceeds its limitations, meaning that the design fit the board well.

Table 4.1: Resource Usage of the Implemented MobileNet

	LUT	LUTRAM	FF	BRAM	DSP	I/O	BUFG
Utilization	161209	19167	168522	159.5	1104	6	2
Availability	230400	101760	460800	312	1728	360	544
Utilization%	69.97	18.84	36.57	51.12	63.89	1.67	0.37

152.5 BRAMs are used, from which 64 BRAMs are for the weights of depthwise convolution, 7.5 BRAMs are for the weights of pointwise convolution, 6 BRAMs are for the weights of fully connected layer, 4 BRAMs are used to store the original image. Each of the 64 accelerator uses a BRAM to store intermediate result. The other BRAMs are used for debug hub.

Table 4.2 shows the resource usage for each accelerator and the resource usage for fully connected layer processor.

Table 4.2: Resource Usage of the Accelerator and the Fully Connected Layer Processor

		LUT	LUTRAM	BRAM	DSP
Accelerator	Total	2433	289	1	17
	Depthwise convolution	500	204	0	9
	Pointwise convolution	1536	0	0	8
	Pooling	228	85	0	0
	ReLU	85	0	0	0
	RAM	1	0	1	0
Fully Connected Layer		1189	0	6	16

In each accelerator, a single depthwise convolution uses 9 DSPs, and a single pointwise convolution block uses 8 DSPs. A fully connected layer processor uses 16 DSPs. Thus, there are 1104 DSPs used for 64 accelerators and a fully connected layer processor.

4.3 Timing Performance

The implemented design is working under 100 MHz frequency. It passed all timing constraints. Its worst hold slack (WHS) is 0.471 ns. Its worst negative slack (WNS) is 0.010 ns. Its worst pulse width slack is 3.498ns. The limits are all 10 ns in 100 MHz.

It takes 69191 clock cycles to deal with one image frame in the MobileNet. It is 691.19us under 100Hz frequency. The number of clock cycles used for each procedure is listed in Table 4.3.

4.4 Power consumption

The power consumption in the design is 4.075W. The power consumption of different part is listed in Table 4.4, from which 68% of the power is for the processor system, and 32% of the power is for programmable logic.

4.5 Comparison with CPU and GPU

Performance is measured by validating 1800 images on CPU (Intel(R) Pentium(R) CPU G4560 @ 3.50GHz), GPU (NVIDIA GeForce 940MX 1.004GHz)

Table 4.3: MobileNet structure to be implemented

Procedure Number	Layer Number	Execution	Clock Cycles
Procedure 0	Layer 0	Depthwise Convolution	16585
	Layer 1	Pooling	
	Layer 2	ReLU	
Procedure 1	Layer 3	Depthwise Convolution	4173
	Layer 4	ReLU	
Procedure 2	Layer 5	Pointwise Convolution	32787
	Layer 6	ReLU	
	Layer 7	Pooling	
Procedure 3	Layer 8	Depthwise Convolution	1069
	Layer 9	ReLU	
Procedure 4	Layer 10	Pointwise Convolution	8211
	Layer 11	Pooling	
	Layer 12	ReLU	
Procedure 5	Layer 13	Depthwise Convolution	285
	Layer 14	ReLU	
Procedure 6	Layer 15	Pointwise Convolution	2063
	Layer 16	ReLU	
Procedure 7	Layer 17	Depthwise Convolution	285
	Layer 18	ReLU	
Procedure 8	Layer 19	Pointwise Convolution	2067
	Layer 20	Pooling	
	Layer 21	ReLU	
Procedure 9	Layer 22	Depthwise Convolution	85
	Layer 23	ReLU	
Procedure 10	Layer 24	Pointwise Convolution	527
	Layer 25	ReLU	
Fully Connected Layer			
Procedure Number	Layer Number	Execution	Clock Cycles
Procedure 11	Layer 26	Fully Connected	1054
	Layer 27	Fully Connected	
Total			69191

Table 4.4: Power Usage of the Implemented MobileNet

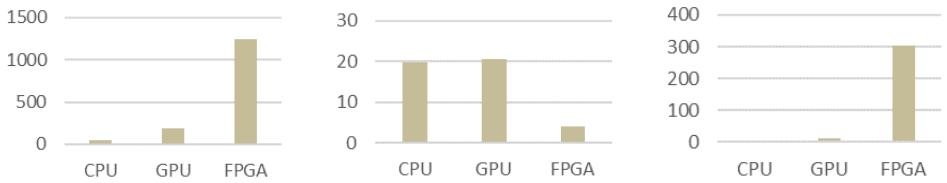
	Dynamic							Static	
	Clocks	Signals	Logic	BRAM	DSP	I/O	PS	PL	PS
Power(W)	0.284	0.036	0.054	0.175	0.124	0.004	2.678	0.621	0.099
Utilization%	6.97	0.88	1.33	4.29	3.04	0.10	65.72	15.24	2.43

and FPGA (Xilinx Ultrascale+ Zu104 100MHz) using the same MobileNet structure in Table 2.2. The result is shown in Table 4.5. Performance is defined as how many frames could be processed in one second. Power is the average power in validation. Efficiency is defined as how many frames could be processed by 1 Watt.

FPGA implementation in this project shows an advantage in higher performance. It has a 28.4x speed-up than CPU and a 6.5x speed-up than GPU. The FPGA implementation also costs lower power consumption and has higher power efficiency over CPU and GPU.

Table 4.5: Comparison of CPU, GPU, and FPGA in Efficiency

	CPU	GPU	FPGA
Performance	44fps	193fps	1250fps
Power	19.9w	20.7w	4.1w
Efficiency	2.2fps/w	9.3fps/w	304.9fps/w



(a) Performance (b) Power (c) Efficiency

Figure 4.2: Performance (a), Power (b) and Efficiency (c) of CPU, GPU and FPGA

4.6 Summary

The designed MobileNet is implemented on FPGA. It passed all the resource and timing constraints of Xilinx UltraScale+ ZU104 FPGA board. The imple-

mented MobileNet on FPGA has higher performance, lower power consumption and higher power efficiency than CPU (Intel(R) Pentium(R) CPU G4560 @ 3.50GHz), GPU (NVIDIA GeForce 940MX 1.004GHz).

Chapter 5

Gesture Classification example

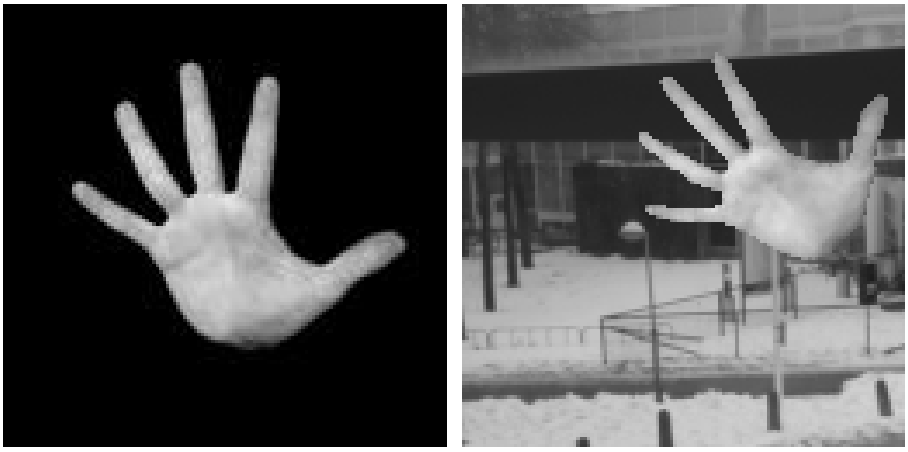
An example of using the network on FPGA to solve actual problems will be presented in this chapter. We will use the MobileNet to solve a gesture classification problem. People use 6 gestures to represent 0 to 5. The task is to recognize which number the gesture represents in an image.

5.1 Dataset

The dataset is adjusted from Fingers [42] from Kaggle.com. The original dataset contains 9800 images with gestures representing 0 to 5. 8000 images are used for training and 1800 images are used for validation. Each image is 128×128 pixels in size and each pixel is 8-bit in width. Figure 5.1(a) is an image from original dataset. The backgrounds of the images are all black. The gestures are in the same size and the same direction. In order to simulate real-life cases, interference factors including background, rotation, scaling, shifting, and noise are added to the original images. Figure 5.1(b) is an example of the adjusted dataset.

5.2 Network Structure and Training

The neural network follows the structure shown in Table 2.2. The network is trained on GPU by Python using Pytorch package. The training process is explained in [39]. The original weights are floating points.



(a) Performance (b) Power

Figure 5.1: An Image from Original Dataset (a) and an Image from Adjusted Dataset (b)

5.3 Accuracy Result

An independent dataset of 1800 images is used for validation. There are 300 images for each gesture representing 0 to 5. Validation on software is done by a Python program. As shown in Table 5.1, the trained network with floating point weights has an overall accuracy of 61%.

Table 5.1: Software Simulation Result with Floating Point Weights

Classes	0	1	2	3	4	5
Accuracy%	78	61	62	56	73	91

In order to fit the hardware structure, all the weights are quantized to 8-bit signed integers ranging from -128 to 127. Then the new weights are applied to the same network for validation. The validation result of the network with 8-bit integer weights in Python environment is shown in Table 5.2. It has an overall accuracy of 56%. The validation result of the implemented MobileNet on FPGA is shown in Table 5.3. It has an overall accuracy of 43%.

The hardware implementation suffers from a accuracy loss. It is because the software model is not exactly same with the hardware model. It is explained in Section 5.4.

Table 5.2: Software Simulation Result with Integer Weights

Classes	0	1	2	3	4	5
Accuracy%	73	50	48	38	52	79

Table 5.3: Hardware Tested Result with Integer Weights

Classes	0	1	2	3	4	5
Accuracy%	71	36	33	24	22	73

5.4 Analysis

The actual result on FPGA is less accurate than on software, but still evidence shows that it acts as a classifier. There are several possible reasons for accuracy loss.

Precision Loss

The whole network uses 8-bit signed integer as the data format. It represents integers from -128 to 127, which is far from enough in neural network, as the precision losses fast when the neural network goes deeper and deeper.

Case 1: Accuracy decreases when the weights are integers rather than floating point.

Table 5.1 shows the accuracy result of the network with 32-bit floating point, while Table 5.1 is the accuracy result of the network with integers. Both are simulated in software environment. Results show that accuracy loses a lot when shifting from 32-bit floating point to 8-bit signed integers. For example, accuracy of classifying number 4 decreases from 73% to 52% when using integers instead of floating point as weight format.

Case 2: How the floating point are rounded to integers will affect the final result.

An image representing 2 is validated in two networks. The weights of one of the networks are rounded to the nearest integers, while the weights of another network are rounded to integers by round down strategy. Table 5.5 and Table 5.4 shows the result data of the two networks. Though there is miner difference in the network weights, the final results show huge difference between each other. It even leads to misclassification. In the network with weights rounded to the nearest integer, the image is classified to 2, which is the expected result. However, in the network with the weights rounded down, the image is classified to 1.

Table 5.4: An Image Classification Result with Rounded Weights

Classes	0	1	2	3	4	5
Results	3165	34252	54786	15610	-70377	-112112

Table 5.5: An Image Classification Result with Rounded-Down Weights

Classes	0	1	2	3	4	5
Results	14665	57989	47953	1009	-52666	-76242

Overflow

Integers also brings the problem of overflow. The intermediate results of multiply-accumulators in both depthwise convolution and pointwise convolution are in 17-bit signed integer format. In depthwise convolution, the convolution is the sum of 9 17-bit signed integers. The case is even worse in pointwise convolution, in some situations, the result for one pixel is the sum of 64 17-bit signed integers. Overflow may occur in high possibility when doing accumulation. This problem could be fixed in hardware, by setting the overflowed value to the maximum representable value. However, in software validation process, the situation is not included. It causes the difference in software behavior and hardware behavior.

5.5 Summary

Gesture classification problem is solved by the implemented MobileNet on FPGA. Though accuracy is a bit away from software simulation, the FPGA design works as a classifier. The reason for accuracy loss is analyzed in this chapter. Precision loss and overflow caused by using 8-bit integers instead of floating-point numbers could be the problem.

Chapter 6

Conclusion

In this thesis, we implemented a simplified MobileNet on FPGA and used the MobileNet to solve a realistic gesture classification problem. We specially designed an accelerator block to accelerate the depthwise convolution, pointwise convolution, and pooling in MobileNet. We also designed a fully connected layer processor. The simplified MobileNet is built based on a system which integrates 64 accelerators and a fully connected layer processor. The system largely accelerates the inference phase of the MobileNet. It works under 100MHz frequency on Xilinx UltraScale+ Zu104 FPGA board. It shows a 28.4x speed up than CPU (Intel(R) Pentium(R) CPU G4560 @ 3.50GHz), and a 6.5x speed up than GPU (NVIDIA GeForce 940MX 1.004GHz). Besides, it is a power efficient design. Its power consumption is 4.07w. We used the MobileNet to solve a gesture classification problem. The dataset contains gestures representing zero to five with backgrounds. We trained the MobileNet in Python environment and applied the trained weights to the implemented MobileNet on FPGA. The accuracy of the gesture classification problem reaches 43% on FPGA.

As a future work, we plan to explore deeper into improving the implemented design. The thesis work implemented a specific MobileNet on FPGA. However, the design is applicable to other MobileNet structures as explained in Section 3.5. Further research could be made in applying the design to more complex MobileNet structures. Besides gesture classification problem, the implemented MobileNet could be applied to other image classification problems, especially some commonly recognized benchmarks, for example, ImageNet [19]. Testifying the implemented design's performance and accuracy on these benchmarks makes it comparable to other MobileNet implementations and it may help readers understand the advantage of the FPGA

design better. The implementation is a correct one but not a perfect one. One major disadvantage is that there exists an accuracy loss migrating the software trained neural network to hardware platform. Further research could focus on minimizing the effect of quantization. From hardware perspective, there exists room for the improvement of efficiency. The implemented FPGA design now works at 100MHz. As the frequency increases, it will reach faster speed but consume more power. By applying different frequencies to the hardware implementation, the relationship between power and speed can be explored. In this way, finding a balance between speed and power can improve the efficiency of the design.

References

[1] Xilinx. Ultrascale architecture dsp slice. Technical report, Xilinx, 2019.

[2] Xilinx. Axi reference guide. Technical report, Xilinx, 2011.

[3] Xilinx. Zcu104 evaluation board. Technical report, Xilinx, 2018.

[4] Kamel Abdelouahab, Maxime Pelcat, Jocelyn Sérot, and François Berry. Accelerating CNN inference on fpgas: A survey. *CoRR*, abs/1806.01683, 2018.

[5] 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. *CoRR*, abs/1704.04861, 2017.

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

[7] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Fei-Fei Li. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2009), 20-25 June 2009, Miami, Florida, USA*, pages 248–255, 2009.

[8] Bharath Hariharan, Pablo Arbelaez, Ross B. Girshick, and Jitendra Malik. Simultaneous detection and segmentation. *CoRR*, abs/1407.1808, 2014.

[9] Filip Radenovic, Giorgos Tolias, and Ondrej Chum. CNN image retrieval learns from bow: Unsupervised fine-tuning with hard examples. In *Computer Vision - ECCV 2016 - 14th European Conference, Amsterdam,*

The Netherlands, October 11-14, 2016, Proceedings, Part I, pages 3–20, 2016.

[10] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning convolutional neural networks for resource efficient inference. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, 2017.

[11] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*, pages 4510–4520, 2018.

[12] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. Shufflenet V2: practical guidelines for efficient CNN architecture design. In *Computer Vision - ECCV 2018 - 15th European Conference, Munich, Germany, September 8-14, 2018, Proceedings, Part XIV*, pages 122–138, 2018.

[13] Gao Huang, Shichen Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Condensenet: An efficient densenet using learned group convolutions. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*, pages 2752–2761, 2018.

[14] Matthieu Courbariaux and Yoshua Bengio. Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1. *CoRR*, abs/1602.02830, 2016.

[15] Hande Alemdar, Vincent Leroy, Adrien Prost-Boucle, and Frédéric Pétrot. Ternary neural networks for resource-efficient AI applications. In *2017 International Joint Conference on Neural Networks, IJCNN 2017, Anchorage, AK, USA, May 14-19, 2017*, pages 2547–2554, 2017.

[16] Cathal Murphy and Yao Fu. Xilinx all programmable devices: A superior platform for compute-intensive systems, 2017.

[17] Rikiya Yamashita, Mizuho Nishio, Richard Kinh Gian Do, and Kaori Togashi. Convolutional neural networks: an overview and application in radiology. *Insights into Imaging*, 9(4):611–629, Aug 2018.

[18] F. Schilling. The effect of batch normalization on deep convolutional neural networks. *KTH*, 2016.

[19] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[20] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[21] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*, pages 1–9, 2015.

[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 770–778, 2016.

[23] Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman. Speeding up convolutional neural networks with low rank expansions. In *British Machine Vision Conference, BMVC 2014, Nottingham, UK, September 1-5, 2014*, 2014.

[24] Michaël Mathieu, Mikael Henaff, and Yann LeCun. Fast training of convolutional networks through ffts. In *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, 2014.

[25] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning convolutional neural networks for resource efficient inference. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, 2017.

[26] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024*, 2014.

[27] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional

neural networks. In *Computer Vision - ECCV 2016 - 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part IV*, pages 525–542, 2016.

[28] Zhe Li, Xiaoyu Wang, Xutao Lv, and Tianbao Yang. Sep-nets: Small and effective pattern networks. *CoRR*, abs/1706.03912, 2017.

[29] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, February 22-24, 2015*, pages 161–170, 2015.

[30] Jialiang Zhang and Jing Li. Improving the performance of opencl-based FPGA accelerator for convolutional neural network. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2017, Monterey, CA, USA, February 22-24, 2017*, pages 25–34, 2017.

[31] Chen Zhang, Di Wu, Jiayu Sun, Guangyu Sun, Guojie Luo, and Jason Cong. Energy-efficient CNN implementation on a deeply pipelined FPGA cluster. In *Proceedings of the 2016 International Symposium on Low Power Electronics and Design, ISLPED 2016, San Francisco Airport, CA, USA, August 08 - 10, 2016*, pages 326–331, 2016.

[32] Tong Geng, Tianqi Wang, Ang Li, Xi Jin, and Martin C. Herbordt. A scalable framework for acceleration of CNN training on deeply-pipelined FPGA clusters with weight and workload balancing. *CoRR*, abs/1901.01007, 2019.

[33] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017.

[34] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. Fused-layer CNN accelerators. In *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*, pages 22:1–22:12, 2016.

[35] Ritchie Zhao, Weinan Song, Wentao Zhang, Tianwei Xing, Jeng-Hau Lin, Mani B. 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, FPGA 2017, Monterey, CA, USA, February 22-24, 2017*, pages 15–24, 2017.

[36] Shuang Liang, Shouyi Yin, Leibo Liu, Wayne Luk, and Shaojun Wei. FP-BNN: binarized neural network on FPGA. *Neurocomputing*, 275:1072–1086, 2018.

[37] Lin Bai, Yiming Zhao, and Xinming Huang. A CNN accelerator on FPGA using depthwise separable convolution. *IEEE Trans. on Circuits and Systems*, 65-II(10):1415–1419, 2018.

[38] Ruizhe Zhao, Xinyu Niu, and Wayne Luk. Automatic optimising CNN with depthwise separable convolution on FPGA: (abstact only). In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2018, Monterey, CA, USA, February 25-27, 2018*, page 285, 2018.

[39] Zekun Du. Algorithm design and optimization of convolutional neural networks implemented on fpgas, 2019.

[40] Xilinx. Vivado design suite user guide: Synthesis, 2019.

[41] Xilinx. Vivado design suite user guide: Implementation. *Xilinx*, 2019.

[42] Pavel Koryakin. Fingers, 2019.

TRITA-EECS-EX-2019:659