Master's Thesis

A Study on Optimization of Residual Binarized Neural Network

1910260 CHEN Yan

Supervisor	Tanaka Kiyofumi
Main Examiner	Tanaka Kiyofumi
Examiners	Inoguchi Yasushi
	Kaneko Mineo
	Lim Yuto

Graduate School of Advanced Science and Technology
Japan Advanced Institute of Science and Technology
(Information Science)

February 2021

Abstract

Convolutional Neural Networks (CNNs) show their high ability in image classification and are widely used in many fields in recent years. Modern CNNs models contain millions of parameters and require billions of floating-point operations to infer an image. Lightweight CNNs models such as MobileNet [1] and ShuffleNet [2] were proposed to enable CNNs to run on mobile devices. On the other hand, Binarized Neural Networks (BNNs) for hardware with higher speed and lower power consumption was also introduced. BNN is a technology that realizes speedup and weight reduction by converting the conventional CNN floating-point matrix operation to a binary (-1 and $+1$) bit XNOR operation at the expense of some accuracy. It is suitable for BNNs to run inference on specific hardware accelerators in, for example, FPGA devices. Residual Binarized Neural Network (ReBNet) [3] greatly improves accuracy by introducing binarize factor and performing multi-level binarization. Since the binarize factors in ReBNet are decimals, it is necessary to multiply the fixed-point numbers by DSP in the FPGAs. DSPs are a scarce resource in today's FPGAs, so their degree of parallelism is limited.

In this thesis, we introduce the basic knowledge of CNNs and BNNs and recent related work. And we propose a state-of-the-art method to accelerate and optimize the ReBNet by replacing fixed-point number multiplication with logical shift operation. We designed an end-to-end framework for training Binarized Neural Networks, on which the conversion to logical-shift-based multiplication on software and hardware accelerators implemented on FPGA is performed. We propose Isometric Residual-Binarization, which reduces the elements of binarize factor from the number of levels to 1, and reuses the single element to express binarize factor vector of multiple levels. Like ReBNet, this binarize factor can be determined through training. Then, we show how to transform the parameters in the convolutional layer, fully connected layer, and batch normalization layer, so that the binarize factors become integers. Benefiting from this, no more DSPs are required to multiply the binarize factors, and a large quantity of hardware recourse can be saved. We redesign processing elements (PEs):

- only 1 DSP and 1 accumulator are required per PE,
- encoder becomes much simpler than that in ReBNet,
- it can compute multiple levels at the same time but requires more popcount modules.

We also apply throughput optimization which makes Initiation Interval from the number of levels to 1 in our design. However, this optimization causes the data stream to become wider and logic between layers to become larger. We propose Adaptive Bit Width to resolve these problems without any performance degradation.

We implement our design and compare it with baseline research in different settings. First, we train BNNs models with Isometric Residual-Binarization we proposed on multiple datasets in 3 neutral network architectures. Compared with the original work, ours has fewer parameters but reaches similar accuracy. Then, we transform parameters in models and evaluate the accuracy changes. The accuracy does not change obviously. Next, we make the testbenches and simulate the behavior of hardware accelerators, and start implementation . We try to find the possibly highest degree of parallelism for 2 levels and 3 levels on the resource-limited device for architectures of small datasets, and our design achieves 8 times higher throughput than ReBNet on average. We also implement the maximum parallelism of architectures of small datasets on the large device to find out the hardware resource usage in highly parallel. For the model of the large dataset which requires a huge amount of on-chip memory, we implement it on the Virtex UltraScale device. After that, we measure the scenario of the resource-limited device on the development kit. We design the software evaluation programs and measure the accuracy, throughput, and power usage. All of the results are the same with software models, and the throughput is very close to the theoretical values. We compare these with GPU implementations. At last, we analyze the effect of each optimization quantitatively. Hardware resource usage of each PE is much lower than ReBNet when input data width is less than or equal to 32. Our Data Stream with Adaptive Bit Width can reduce up to $\frac{7}{8}$ Block RAMs in the buffer of sampling modules of convolutional layers.

Finally, we conclude this thesis and describe the future works.

Keywords: Binarized Neural Networks, FPGA, Convolutional Neural Networks.

Acknowledgment

Firstly, I would like to express my sincere gratitude to my supervisor, Professor Tanaka Kiyofumi, who helped me finish this research. My supervisor provided great support for my research and writing my master's thesis. Secondly, I would like to thank my second supervisor, Inoguchi Yasushi, who gave good suggestions and comments about my research in the presentation. Thirdly, I would like to thank my graduate school, JAIST, which provided computation resources to run training programs and synthesis tools. JAIST also gave me a scholarship that exempted all tuition fees in the second year. And I would like to thank Japan Student Services Organization (JASSO) which gave me the honors scholarship. Finally, I would like to thank my family who has been supported me both physically and mentally.

List of Abbreviations

AXI	Advanced eXtensible Interface
BN	Batch Normalization
BNNs	Binarized Neural Networks
CNNs	Convolutional Neural Networks
FPGA	Field Programmable Gate Array
GPUs	Graphics Processing Units
HLS	High Level Synthesis
II	Initiation Interval
ILSVRC	ImageNet Large Scale Visual Recognition Competition
IoT	Internet of Things
LUT	Look-Up-Table
MVTU	Matrix Vector Threshold Unit
PEs	Processing Elements
ReLU	Rectified Linear Units
SGD	Stochastic gradient descent
SWU	Sliding Window Unit

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

With the computing ability of GPU improving, Convolutional Neural Networks (CNNs) can be trained in a reasonable time. CNNs show their high classification ability and are widely used in many fields in recent years. The problems to be solved by CNNs are becoming much more complex, and it is difficult for low-power devices, especially Internet of Things (IoT) devices, to run CNNs locally. For some hard real-time tasks, such as semantic segmentation of autopilot, the CNNs have to finish in several microseconds. Computing on a remote server requires high bandwidth, low latency and high network connection quality. They are really difficult problems. Therefore, this type of task needs to be processed locally.

Modern CNNs' architecture has become larger and more complex to perform difficult tasks, and it usually needs millions of parameters and billions of floating-point operations to run one picture. For instance, the winner of the ImageNet Large Scale Visual Recognition Competition (ILSVRC) in 2015 was ResNet [9], and ResNet-50 has 25.5 millions of parameters and needs 3.8 billion floating-point operations to infer one picture. The winner of the ILSVRC in 2014 was Inception [10], which has 7.0 million parameters and needs 1.6 billion floating-point operations to infer one picture.

So many parameters and floating-point operations make it difficult for even high-end Graphics Processing Units (GPUs) to run the modern CNNs in real-time, and GPUs need to inference multiple pictures at one time which is called "mini-batch" to maximum the throughput. The batch-based acceleration of GPUs results in high average latency. Or inferring only one picture at one time, it will make throughput significantly lower.

Several ways have been proposed to reduce operations or parameters in CNNs, such as reducing operations by using depth-wise convolution [1, 2],

Binarized Neural Networks (BNNs) [11, 12], parameter quantization [13, 14], and Pruning [15, 16] and sparse convolution [17, 18].

CNNs' models should be trained with 32-bit floating-point precision or 32-bit/16-bit mixed floating-point precision [19], while the inference process does not require such high precision. 8-bit integer quantization [13] is popular for inference in CNNs on embedded or mobile devices. Even 1-bit binarization can achieve high accuracy.

In BinaryNet [11], Courbariaux et al. convert almost all floating-point operations to binary operations where floating-point number multiplication becomes logical XNOR. In a Field Programmable Gate Array (FPGA), a logical XNOR gate can be implemented as Look-Up-Table (LUT). LUT is one of main resources in a modern FPGA. Today's representative LUT has 6 inputs and 2 outputs. The two outputs can share several inputs by rules, so one LUT can be configured as 2 XNOR gates. With those LUTs, even small FPGA devices can perform up to trillions of XNOR operations in one second. In addition, compared with floating-point or quantized CNNs, binary weights in BNNs spend less memory space to store them. This means that binary weights can be stored in Block RAM or Distributed Memory on FPGA devices, and it takes a few clock cycles to load them. Compare with GPUs, FPGA devices can process data in pipeline and latencies for each input data are identical, and the latency can be calculated by by taking the running clock frequency into account, so they are more suitable to do inference in CNNs.

In Xnor-Net [12], Rastegari et al. added scaling factor to weight, which makes their design get higher accuracy in the ImageNet dataset. FINN [20] is a BNNs' framework which provides the fastest processing, and it is flexible so that it can be configured to adapt to various FPGA devices of different sizes by changing the degree of parallelism. FINN also uses threshold comparison to avoid multiplications in Batch Normalization (BN) [21].

BNNs with 1-bit activation and 1-bit weights lead to very limited accuracy. On the other hand, Residual Binarized Neural Network (ReBNet) [3] which has multiple-bit activation and 1-bit weights gets much better accuracy, and is comparable with floating-point precision. ReBNet introduces multiple-bit activation in FINN's framework by getting the sign of the difference between the residuals and the binarize factors. When bit width of activation is supposed to be M, this is called M levels of residual binarization, and M binarize factors for activation are required. Here, the scaling factors

which are fixed-point values need to be multiplied with accumulated values in the next layer. Multiplication of fixed-point numbers is basically mapped to DSP48 resources which are embedded multiply-accumulate calculators in Xilinx FPGA families. However, since the number of DSP48 resources in a device is limited, this results in that ReBNet easily runs out of DSP48s, and therefore needs a great amount of LUTs to make up for the lack of DSP48s. This imbalance limits the maximum degree of parallelism, makes place-and-route processes hard in implementing the whole design, and degrades the maximum clock frequency.

1.2 Objective

In this thesis, we try to resolve the problems in ReBNet [3] mentioned above. ReBNet [3] takes too many DSP48s, and we try to reduce the utilization of DSP48s by redesigning the binarization algorithm. The new binarization algorithm would achieve similar accuracy to ReBNet [3], and the hardware implementation of this algorithm does not run out of DSP48s and saves many LUTs so that we can apply throughput optimization and increase the degree of parallelism. Besides, the buffering method of Sliding Window Unit (SWU) in ReBNet wastes too many BRAMs, and we try to resolve this problem in our design. We implement hardware accelerators for several CNNs' architectures on FPGA devices and compare them with the baseline design, ReBNet, in different ways.

1.3 Outline

The rest of this thesis is organized as follows:

- **Chapter 2** shows the techniques used in CNNs and BNNs.
- **Chapter 3** proposes our accelerator which improves ReBNet.
- **Chapter 4** shows implementation of our design and compares it with baseline research in different settings.
- **Chapter 5** concludes this thesis.

Chapter 2

Related Works

In this section, we introduce the CNNs and BNNs, and show the techniques used in BNNs and explain residual binarization in ReBNet [3] in particular. In addition, we present how the FINN framework [20] implements BNNs in parallel on FPGA efficiently. For more details, refer to the original papers.

2.1 Convolutional Neural Networks

In 1962, David Hubel and Torsten Wiesel published a research [22] on the functional architecture in the cat's visual cortex, and firstly proposed "receptive fields" which is a very important notion in CNNs. In 1980, Kunihiko Fukushima proposed Neocognitron [23], which made a neural network model running on the computer and suggested step-by-step convolution filter, Rectified Linear Units (ReLU) and average pooling. His work also achieved Sparse Interaction and Translation Invariant. In 1990, Y. Le Cun proposed back-propagation for CNNs, and it absorbed the advantages from Neocognitron [23] and added Parameter Sharing to reduce parameters. This improvement makes the model run or be trained faster and reduce the risk of overfitting. Back-propagation based supervised learning makes CNNs easier to be applied to various fields.

The first modern CNNs as known as LeNet-5 [24] was announced in 1998. LeNet-5 includes 7 layers, which are 2 convolutional layers, 2 pooling layers, 2 fully connected layers and a special output layer. And its basic architecture, convolutional layer $\rightarrow$ pooling layer $\rightarrow$ fully connected layer, is the same as CNNs widely used today. However, CNNs stopped growing in the next over ten years, because it needs too much calculation and takes too much time to train a model.

AlexNet [25] won the ImageNet Large Scale Visual Recognition Competition 2012, and it got 84.7% of Top-5 accuracy (sorting all outputs and

getting the highest 5 classes, Top-5 accuracy means the rate of ground true in that 5 classes). This accuracy was 10.8% higher than the runner-up. AlexNet uses ReLU as activation function, and imports Data Augmentation (Performing translation, rotation, zoom and other operations on the input image, to augment the data set. [26]). Gradient descent algorithm used in AlexNet is mini-batch Stochastic gradient descent (SGD). It also uses dropout to avoid overfitting. AlexNet made CNNs evolve in the next several years, and much more powerful architecture like Inception, VGG, ResNet, etc. was introduced.

Modern CNNs usually contain conventional layers, pooling layers, batch normalization, activation function and fully connected layers. A typical CNNs model is shown in Figure 2.1 We need to define a loss function when we train a model. The loss function is often the error between the output of the model and the ground true in supervised learning. We can calculate gradient matrices of all trainable parameters in each layer via chain rule. Then we need a Gradient Descent Optimization Algorithm to update parameters. We introduce these techniques in the following subsections.

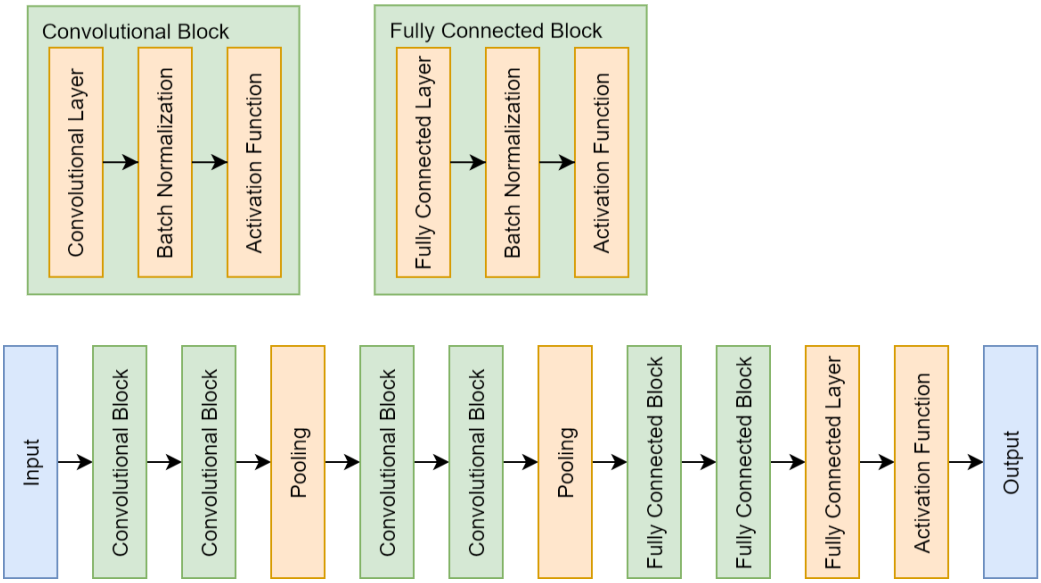


Figure 2.1: A typical CNNs model

2.1.1 Fully Connected Layers(Dense)

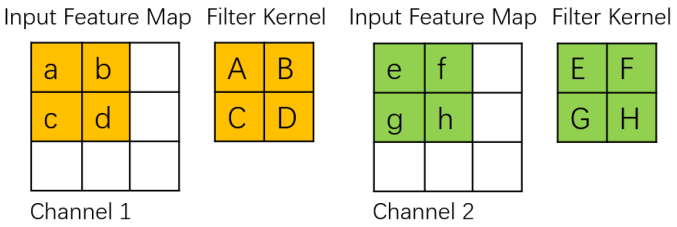
The fundamental of Neural Networks is affine transformation: calculate the dot product of the input vector and weight matrix, then add bias vector optionally. Assume 2×2 matrix x is input, the 1^{st} row is the vector of mini-batch 1 and 2^{nd} row is the vector of mini-batch 2. Matrix W is weight, each column is the parameter for each output node, and the vector b is the bias for each output node. We can get

$$\begin{aligned}
&\begin{bmatrix} x_{1,1} & x_{1,2} \\ x_{2,1} & x_{2,2} \end{bmatrix} \cdot \begin{bmatrix} W_1 & W_2 & W_3 \\ W_4 & W_5 & W_6 \end{bmatrix} + \begin{bmatrix} b_1 & b_2 & b_3 \\ b_1 & b_2 & b_3 \end{bmatrix} \\
&= \begin{bmatrix} x_{1,1}W_1 + x_{1,2}W_4 + b_1 & x_{1,1}W_2 + x_{1,2}W_5 + b_2 & x_{1,1}W_3 + x_{1,2}W_6 + b_3 \\ x_{2,1}W_1 + x_{2,2}W_4 + b_1 & x_{2,1}W_2 + x_{2,2}W_5 + b_2 & x_{2,1}W_3 + x_{2,2}W_6 + b_3 \end{bmatrix},
\end{aligned} \tag{2.1}$$

where the axis definition of output is the same as input.

2.1.2 Convolutional Layers(Conv)

As mentioned before, a fully connected layer needs to calculate dot product of the input and weight, but if we input an image, it is difficult to get dot product, because the amount of the pixels and the Weight matrix is really huge. It is a low efficiency. We can use convolution operation in picture processing to instead of calculating the dot product of the whole input picture. Convolution operation usually needs a filter, but in convolutional layers, it will be a set of filters. These filters slide across the input picture and calculate the dot product of the small picture sampled and kernel (Weights in convolutional layers). So, convolutional layers can share the parameters and the connection to the input is sparse. Input "pictures" of hidden layers is the feature map. Assume that the input feature map is $3 \times 3 \times 2$ in Height, Width and the number of Channels, the kernel size of filters is 2×2, and the number of output channels is 2, the output of this convolutional layer is illustrated in Figure 2.2.



$$output = aA + bB + cC + dD + eE + fF + gG + hH$$

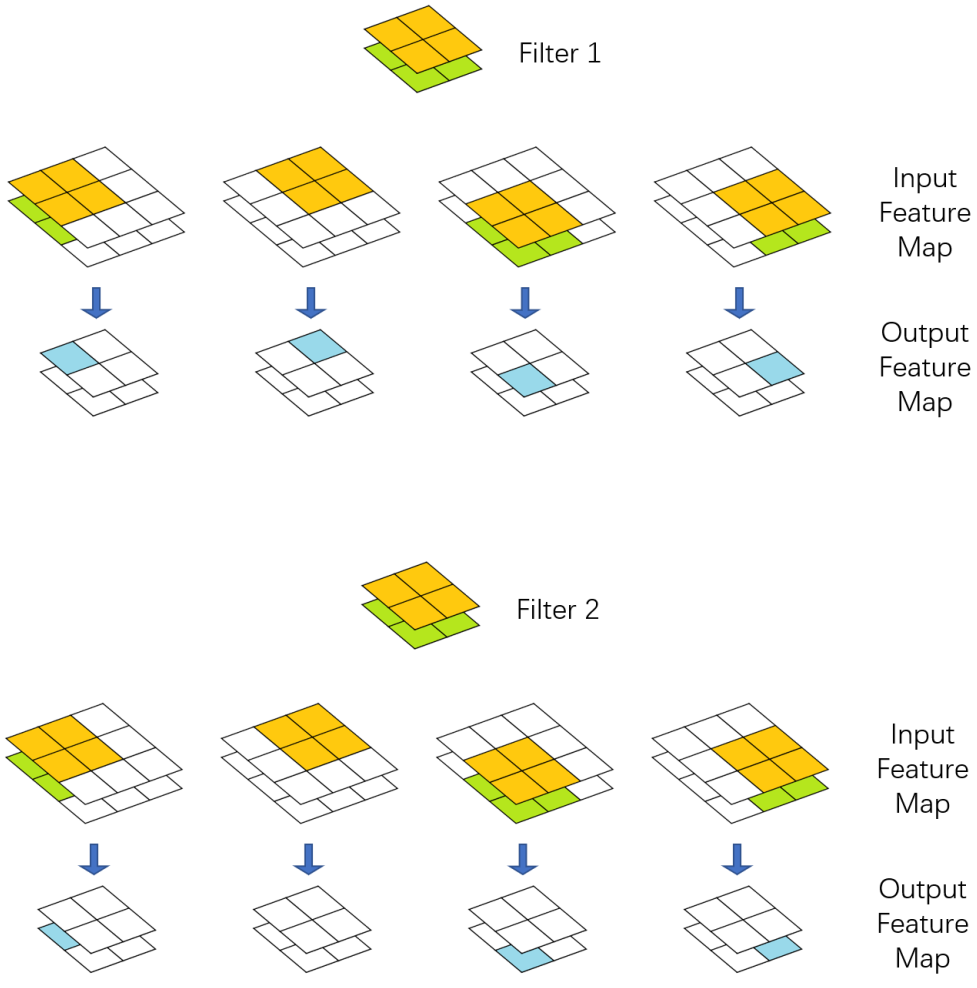


Figure 2.2: convolution operation

As same as fully connected layers, convolutional layers can add bias to each output and add an axis to process multiple data in a mini-batch. Besides, convolutional layers have extra hyperparameters such as stride (sampling step), padding (fill data in the border), dilation (sample data with spacing).

2.1.3 Pooling Layers

Pooling layers can reduce the size of the feature map (downsampling) while keeping the Translation Invariant. As same as convolutional layers, pooling layers also use "filters" which have no kernel parameters to slide across the input feature map. Pooling layers' filters output one value for each channel of input feature map in a sampling window by a specific rule. The rules can be average, maximum and others. Assume input feature map has a size of $4 \times 4 \times 2$ (in Height, Width and the number of Channels). The stride is assumed to be 2. In Figure 2.3, the output size is 2×2, and $func()$ can be $max()$, $avg()$ or others.

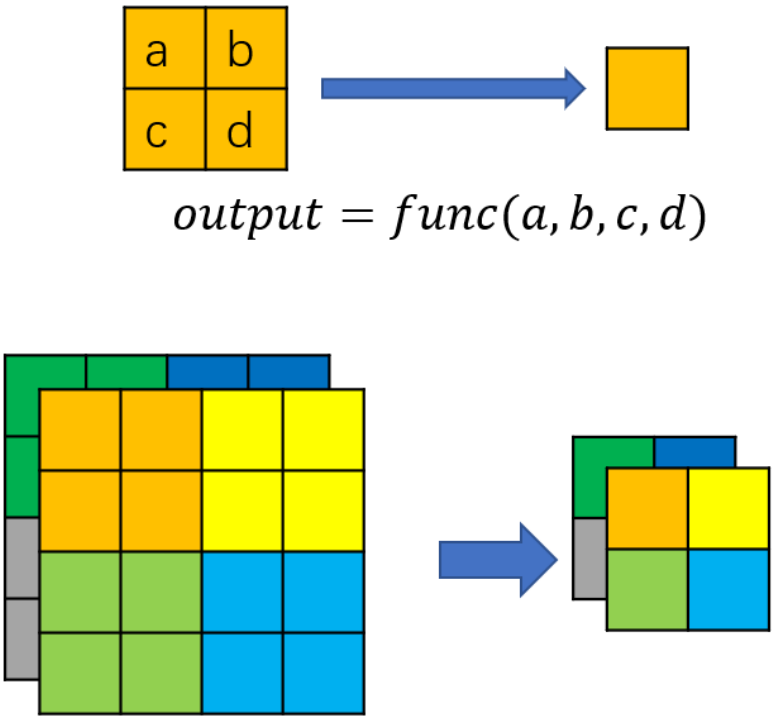


Figure 2.3: Pooling operation

2.1.4 Batch Normalization(BN)

When training a deep neural network model, the distribution of the input feature map will change along with parameters in the previous layer being updated. So it requires to slow down the learning rate and initialize the parameters carefully, and takes too much time to train a model. Batch Normalization(BN) [21] was proposed to resolve this problem. Adding BN to CNNs makes it take less training steps to achieve the same accuracy, and get a bit higher final accuracy. There are typically four parameters in batch normalization layers: moving-mean μ, moving-variance σ^2, γ and β. Moving-mean μ and moving-variance σ^2 are updated by feeding data when the model is being trained. they will be average values over multiple mini-batch to represent the whole dataset. γ and β are parameters of affine transformation and are updated by back propagation during learning. Processing in batch

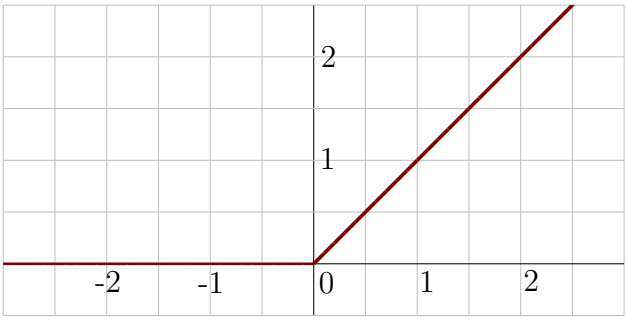
normalization is:

$$output = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} \cdot \gamma + \beta, \tag{2.2}$$

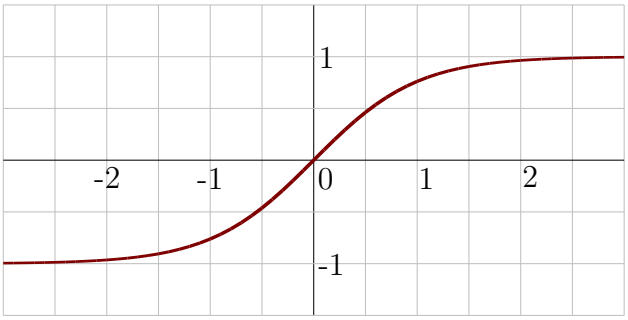
where ϵ is a fixed value which is close to 0 to avoid 0 to be divisor, and x is the input value. After training finished, the moving parameters also are fixed.

2.1.5 Activation Functions

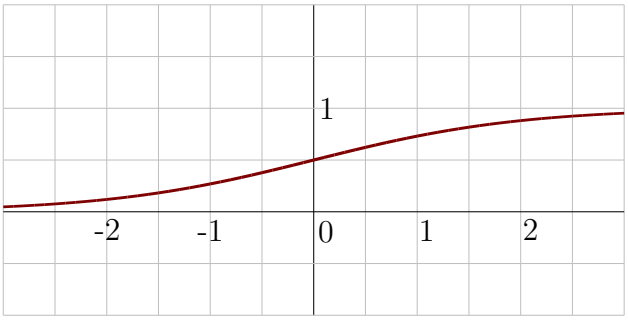
CNNs need to use activation functions to resolve nontrivial problems via a small number of nodes. Activation functions make the output of each layer to be nonlinear. The commonly used activation functions, Rectified Linear Units (ReLU), hyperbolic tangent($tanh$) and sigmoid, are shown in Figure 2.4, and softmax is often used as the last layer of CNNs. The output of softmax is the probability distribution on the categories.



(a) ReLU



(b) tanh

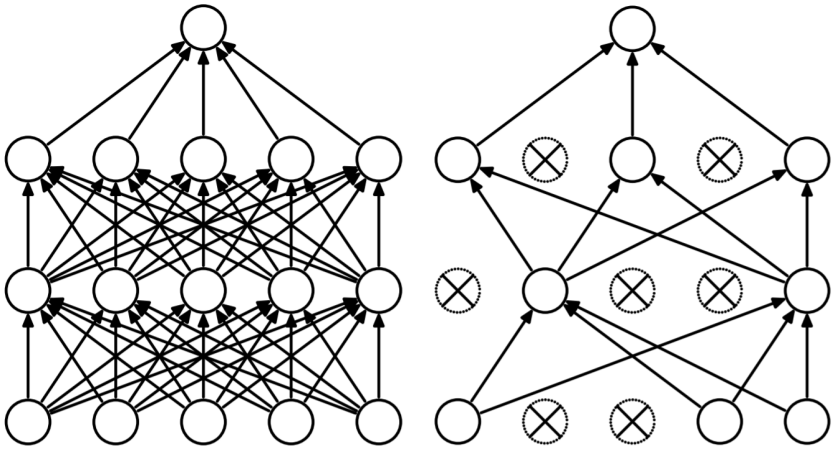


(c) sigmoid

Figure 2.4: Activation functions

2.1.6 Dropout

Dropout [4] is a technique to reduce overfitting. It randomly disables output in specific ratios of nodes/filters during training. (shown in Figure 2.5)



(a) Without dropout (b) With dropout

Figure 2.5: Dropout [4]

2.2 Binarized Neural Networks

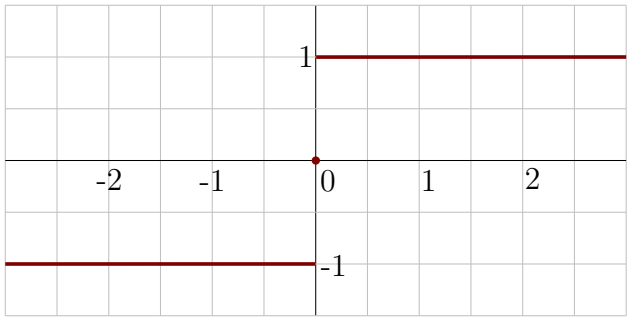
As mentioned in Charter 1, even if parameters and output products of hidden layers in CNNs reduce from 32-bit floating-point numbers to 1-bit binary numbers, the accuracy does not become much lower. And Binarized Neural Networks (BNNs) can run quickly within a low power budget. We introduce techniques used in BNNs and RebNet in the following subsections.

2.2.1 Binarization Activation

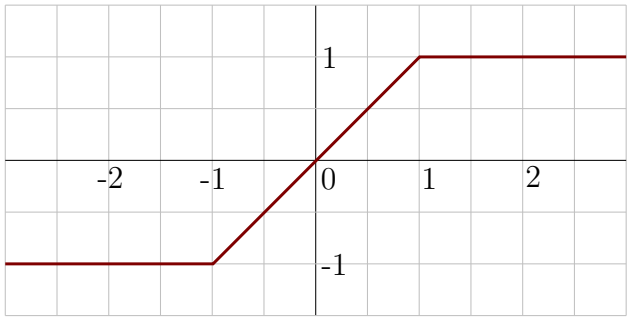
Compare with activation functions in CNNs which can output floating-point numbers, activation functions in BNNs can only output binary numbers(-1 as $false$, $+1$ as $true$). The function to convert floating-point numbers to binary numbers is $sign$(shown in Figure 2.6a). $sign$ function outputs the sign of a floating-point number, where input 0 will output 0, however, there is a very low probability for a floating-point number to be an integer after back propagation update or dot product. $sign$ function is not continuous and does not have a derivative. Hence, the error cannot be back-propagated through a $sign$ function in backpropagation. According to the literature of BinaryNet [11], $sign$ function should be approximated to $clip(x, -1, +1)$ in backpropagation, meaning that if the absolute value of input x is over 1, it

should be cut off. The derived function of *clip* is:

$$clip'(x,-1,+1) = 1_{|x|\leq 1} = \begin{cases} 1 & |x| \leq 1 \\ 0 & otherwise \end{cases} \tag{2.3}$$



(a) *sign* function



(b) *clip* function

Figure 2.6: Activation functions in BNNs

2.2.2 Residual-Binarization Activation

ReBNet proposed a multiple-level binarize activation function. To binarize an input in multiple levels, binarize factor $\gamma_e = \{\gamma_{e_1}, \gamma_{e_2}, \cdots, \gamma_{e_i}\}$ is necessary. Figure 2.7 shows how to convert a fixed-point input x to an approximate binary value e_i and encode it to a binary output b_i. First, we get the sign of the input $r_1(= x)$ as the Level 1's encoded bit b_1. If the sign is positive, b_1 is 1, otherwise 0. Next, r_1 is added by γ_{e_1} when the sign is negative or subtracted by γ_{e_1} when the sign is positive. The residual result r_2 is the input to the Level 2. Repeating this process, we obtain the results in Levels $2, \ldots, M$. Consequently, $e = \sum_{i=1}^{M} \gamma_{e_i} \times sign(r_i)$ is the approximate binary

value for input x, and the relationship between input x and approximate binary value e when M=2 is illustrated in Figure 2.8. γ_e is learned during the training phase.

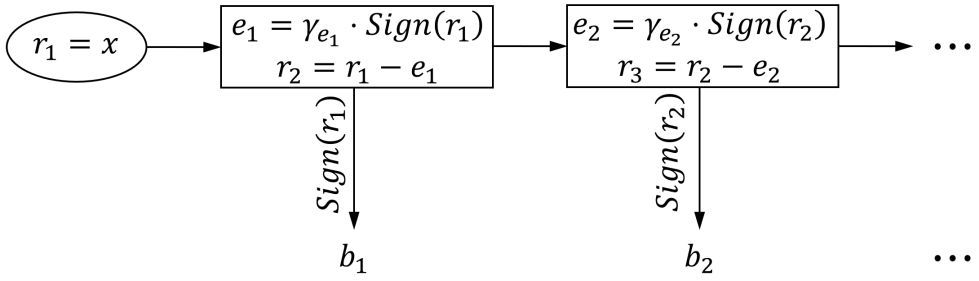


Figure 2.7: Encoding multiple-level residual-binarized bits.

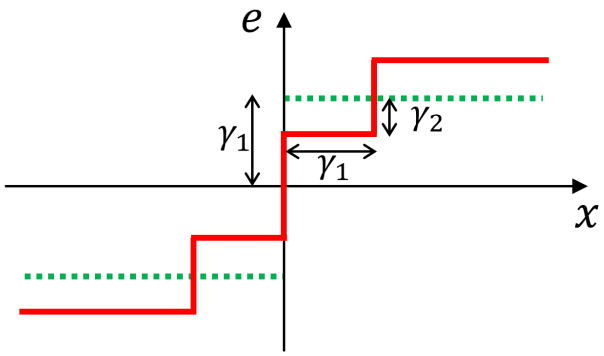


Figure 2.8: Relationship between x and e

2.2.3 XNOR-based dot product

The main calculation in the neural network is the dot product in the fully-connected layers and convolution layers. In a fully-connected layer, it calculates dot products between input vector $\vec{x}$ and weight vector $\vec{w}$. In a convolution layer, it calculates dot products between input feature map vector $\vec{x}$ and kernel vector $\vec{w}$. These dot products operations are as follows.

$$dot(\vec{x}, \vec{w}) = \sum \vec{x}_i \times \vec{w}_i \tag{2.4}$$

In BNNs, $\{\vec{x}, \vec{w}\}$ are restricted to binary values which are $\{\pm\gamma_x, \pm\gamma_w\}$, where $\{\gamma_x, \gamma_w\}$ are scalar values. According to [11], dot products of bina-

rlized $\{\vec{b}_x, \vec{b}_w\}$ can be calculated via XNOR popcount (XnorPopcount). In XnorPopcount, after XNOR operation, the number of positive bits is doubled and then subtracted by the bit width, N (Figure2.9), where p is the number of positive bits and N is the bit width of input. Let $\vec{x} = \gamma_x \vec{s}_x$ and $\vec{w} = \gamma_w \vec{s}_w$, and $\{\vec{s}_x, \vec{s}_w\}$ are sign vectors which only contain ± 1. We replace -1 values in sign vectors $\{\vec{s}_x, \vec{s}_w\}$ by 0s and then obtain $\{\vec{b}_x, \vec{b}_w\}$. We can get:

$$\begin{aligned} dot(\vec{x}, \vec{w}) &= \gamma_x \gamma_w dot(\vec{s}_x, \vec{s}_w) \\ &= \gamma_x \gamma_w XnorPopcount(\vec{b}_x, \vec{b}_w). \end{aligned} \tag{2.5}$$

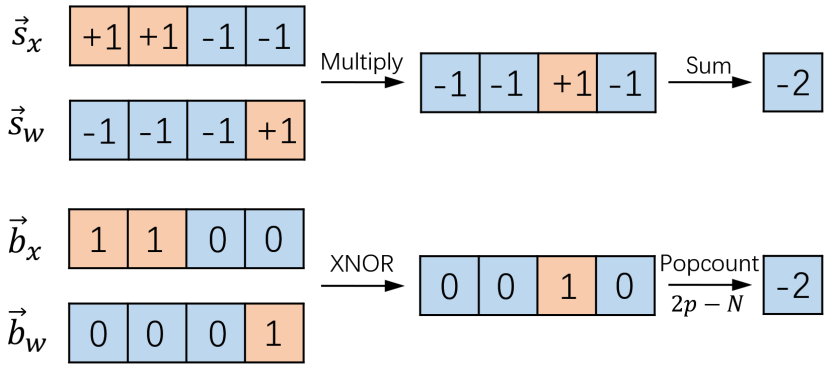


Figure 2.9: Dot-Product and XnorPopcount

In ReBNet [3], dot products between an M-level residual-binarized feature vector $\vec{e}$ and weight vector $\vec{w}$ are calculated in M subprocesses, and the result is the summation of the subprocesses. When i is the level number, The dot product in ReBNet becomes:

$$\begin{aligned} dot(\vec{e}, \vec{w}) &= \sum_{i=1}^{M} \gamma_{e_i} \gamma_w dot(\vec{s}_{e_i}, \vec{s}_w) \\ &= \sum_{i=1}^{M} \gamma_{e_i} \gamma_w XnorPopcount(\vec{b}_{e_i}, \vec{b}_w), \end{aligned} \tag{2.6}$$

where $\vec{s}_{e_i}$ is the binarize factor vector of $\vec{e}$ and $\vec{b}_{e_i}$ is the vector of binary encoded $\vec{e}$.

2.2.4 Threshold-based Batch Normalization

CNNs generally include a batch normalization layer between fully-connected layer or convolution layer and activation function. As mentioned before, software implementation of the batch normalization needs multiplication twice. According to FINN, we can find a threshold τ such that if the input is lager than τ, the output is 1, otherwise, the output is 0, when batch normalization runs on a hardware accelerator. FINN shows that we can calculate

$$\tau = \mu - \frac{(\beta \times \sqrt{\sigma^2 + \epsilon})}{\gamma}. \tag{2.7}$$

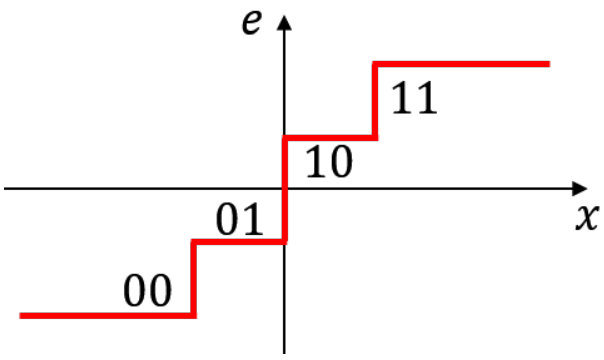
RebNet uses τ to get a difference $\mathcal{D}$ with input x, and then uses this difference $\mathcal{D}$ to multiply a scaling factor

$$\alpha = \frac{\gamma}{\sqrt{\sigma^2 + \epsilon}}, \tag{2.8}$$

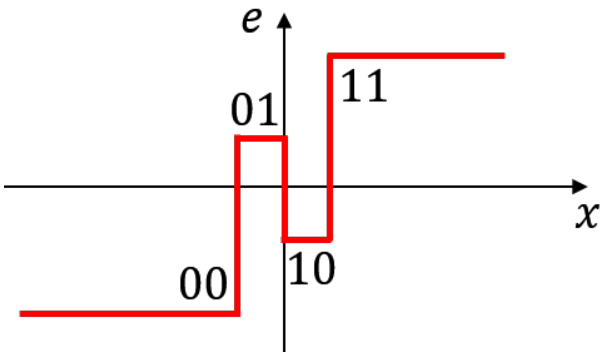
and put the product to residual-binarization activation function.

2.2.5 MaxPooling

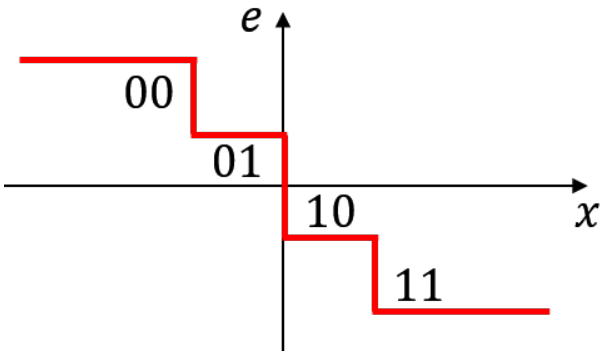
FINN proposed efficient OR-based MaxPooling in their framework. ReBNet cannot calculate MaxPooling correctly by this method since multiple levels' data need to be processed. Instead, ReBNet buffers concatenated levels' bit, from low-level bit to high bit and high-level bit to low bit, and outputs the maximum number in each sampling window. The input data of maxpooling e is encoded by γ_{e_i}, and multiplied by γ in batch normalization. So the value of γ_{e_i} and γ can make the max pooling result wrong in low probability. Figure 2.10 shows those problems when $M = 2$. Figure 2.10a is the normal case, and the maxpooling in ReBNet will output correct maximum valuse in this case. Figure 2.10b is the case when $\gamma_{e_1} < \gamma_{e_2}$, and 01 is larger than 10 in this case, so the maxpooling in ReBNet will outout the wrong result when comparing 01 and 10. Figure 2.10c shows the case when γ in batch normalization is negative, where we can see that the maxpooling in ReBNet will output a totally reverse result in this case.



(a) Normal case



(b) $\gamma_{e_1} < \gamma_{e_2}$ case



(c) $\gamma < 0$ case

Figure 2.10: Problems in MaxPooling of ReBNet

2.2.6 Hardware Accelerator Architecture

Since ReBNet inherited the design of FINN, it maps each layer of neural network to "compute array" modules in hardware accelerator. They can run

in a large "pipeline" (dataflow of High Level Synthesis (HLS) , Figure 2.11), where, after compute 1 starts generating outputs, compute 2 starts computation and compute 1 starts computation of the next data after finished. Running in this way will maximize the throughput, but parameter must be restored on-chip. Accelerator needs a Advanced eXtensible Interface (AXI) master device (such as CPU) to feed images and receive classification results.

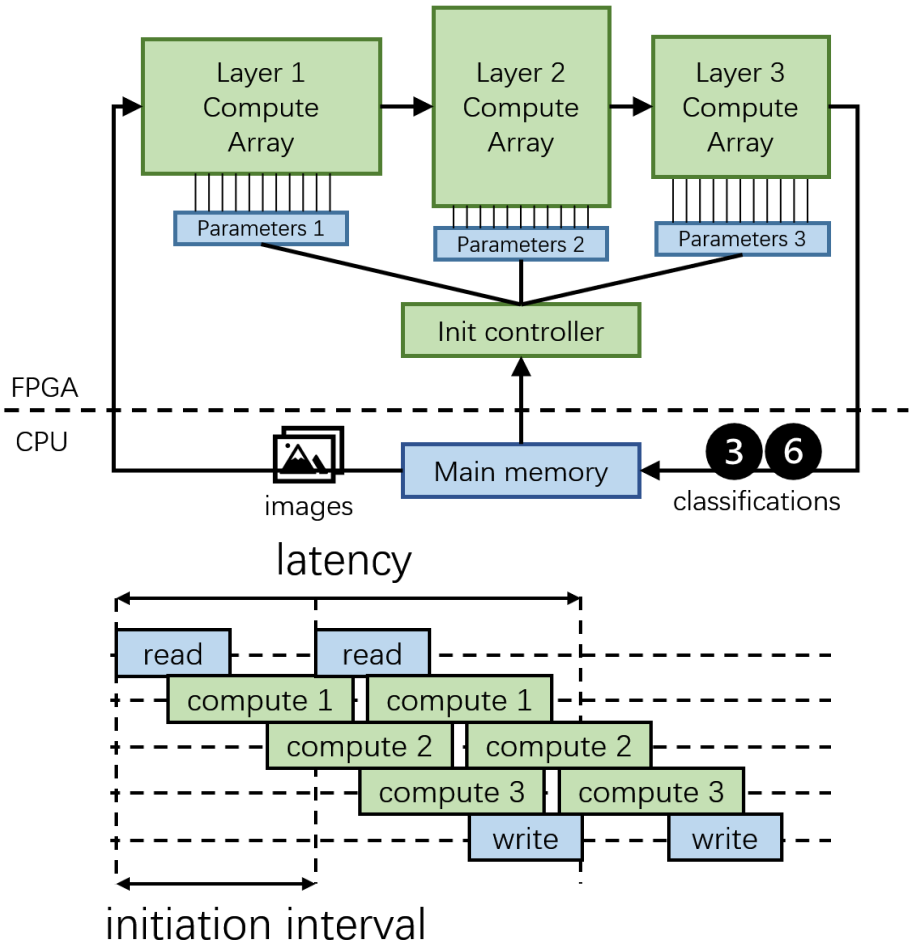


Figure 2.11: Overview of Accelerator and dataflow

The main module in compute array is Matrix Vector Threshold Unit (MVTU) depicted in Figure 2.12, A fully-connected layer uses a MVTU, and a convolution layer uses a Sliding Window Unit (SWU) and an MVTU. SWU is for sampling the input to convolution layers. There are several Processing

Elements (PEs) (Figure 2.13) in an MVTU, and each PE can process multiple 1-bit data in parallel, and the number of data pairs is "SIMD width". The number of PEs and the SIMD width decide the degree of parallelism.

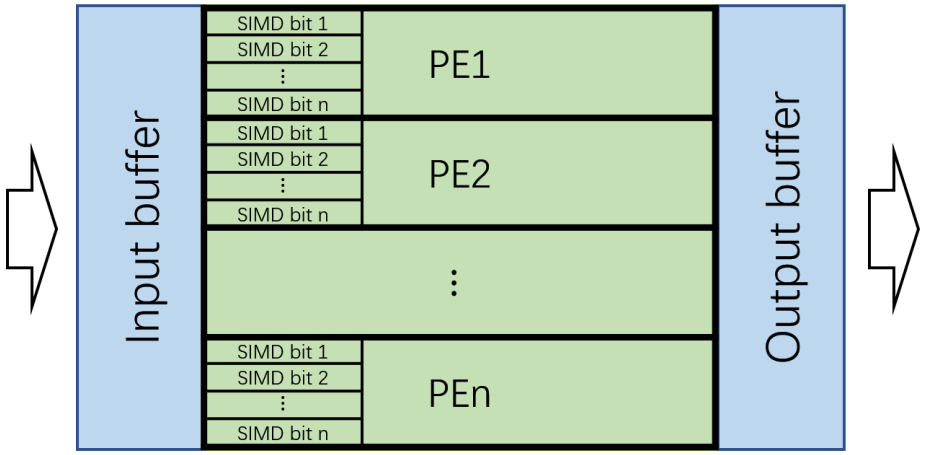


Figure 2.12: MVTU

The processing in each PE before encoding is:

$$\sum(\sum_{i}^{M}(\gamma_{e_i} \cdot \gamma_w \cdot XnorPopcount(\vec{b}_{e_i}, \vec{b}_w))) \cdot \alpha - \tau \cdot \alpha, \tag{2.9}$$

where the outer summation is to process all neurons which are divided to SIMDs, and the inner summation is to process multiple levels input. $\gamma_{e_i} \cdot \gamma_w$ and $\tau \cdot \alpha$ are calculated in advance and stored as constant in memory.

The PE in ReBNet is illustrated in Figure 2.13. While index1 is PE index, index2 is layer index. S is data width of input (SIMD width), P is data width of popcount accumulator, T is data width of all of the fixed-point values, and M is the number of residual-binarization levels. Each PE contains M XNOR modules, a popcout module, M accumulators to store popcounted values in M levels, a MAC module, and an encode module. A MAC module contains $M + 2$ DSP48s, M of which are for accumulated values to multiply γ_{e_i} and 2 of which are for multiplying α and difference $\mathcal{D}$. The encoding module consists of M comparators, $M - 1$ adders, and subtractors.

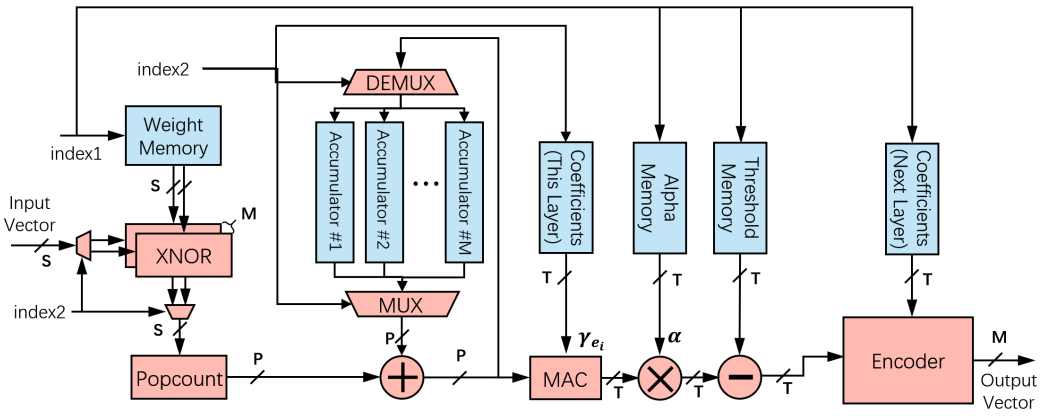


Figure 2.13: Processing Element in ReBNet

Chapter 3

Improvement of ReBNet

In this section, we discuss how we resolve the problems in ReBNet. First, we show the difference in the computation processes. Then we explain the modifications to the hardware accelerator and how to optimize the performance and reduce resource usage.

3.1 Isometric Residual-Binarization

We found that most elements of γ_e in the hidden layers of models which have been well-trained form a geometric sequence with a common ratio of $\frac{1}{2}$, and the relationship between x and e is illustrated in Figure 3.1.

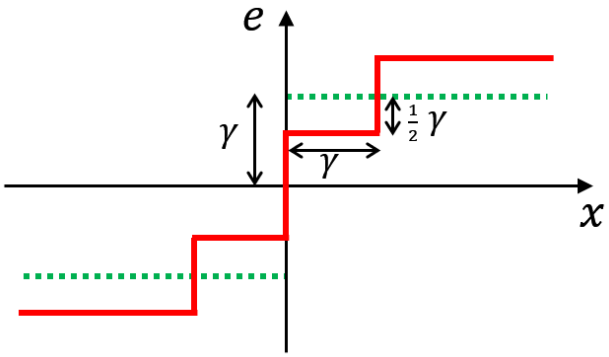


Figure 3.1: relationship between x and e

Then we tried to fix the gamma in the pattern corresponding to this relationship, e.g., $\gamma_e = \{4, 2, 1\}$ when $M = 3$, to train the model so that we can use logical shift for the multiplications. However, we failed to get adequate accuracy. After considering it carefully, we concluded that training in this way would destroy the feature that γ_e can be trained, and fixed values could not express the feature of the input. Therefore, we redesigned the

Residual-Binarization activation function, reduced the elements of binarize factor γ_e from M to 1, and decided to reuse the γ_e to express binarize factor vector $\gamma_{e_i} = \frac{1}{2^{i-1}}\gamma_e$ of multiple levels, where the γ_e could be decided via training, and approximate binary value becomes

$$e = \sum_{i=1}^{M} \frac{1}{2^{i-1}}\gamma_e \times sign(r_i). \tag{3.1}$$

Assumme that $\mathcal{L}$ is loss function, that it uses full-precision in backpropagation, and that sign function causes vanishing gradient. As described in Section 2.1.5, it should be replaced as $clip(x, -1, +1)$ in backpropagation as follows:

$$e = \sum_{i=1}^{M} \frac{1}{2^{i-1}}\gamma_e \times clip(r_i, -1, +1). \tag{3.2}$$

The derivatives of cost function $\mathcal{L}$ with respect to γ_e is computed by chain rule as:

$$\frac{\partial \mathcal{L}}{\partial \gamma_e} = \frac{\partial \mathcal{L}}{\partial e}\frac{\partial e}{\partial \gamma_e} = \sum_{i=1}^{M} \frac{1}{2^{i-1}} clip(r_i, -1, +1), \tag{3.3}$$

and for input x:

$$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial e}\frac{\partial e}{\partial x} = \sum_{i=1}^{M} \frac{1}{2^{i-1}}\gamma_e \times 1_{|r_i|\leq 1}. \tag{3.4}$$

We can update the parameters in the Isometric Residual-Binarization activation function and pass the gradient to the previous layer on a full-precision system.

3.2 Integer scaling of binarize factor

Since we use single trainable γ_e, we need to scale almost all of the parameters and make it easier for hardware to run it. In ReBNet or FINN, there are two ways to input data: 1) For simple datasets in monochrome like MNIST, data is inputted as binary bits($sign(2 \times x - 255)$). 2) For datasets like CIFAR-10 or SVHN, data is inputted as 8 bits fixed-point number($\frac{2\times x-255}{255}$), where the first bit is sign, and the remaining 7 bits are decimal part. For case 1), we do not change the input format, while, for case 2), we use RAW RGB values as input and process $2 \times x - 255$ in the first layer, so as to project the RGB

value $x \in [0, 255]$ to $x \in [-255, 255]$. This makes software simulation and hardware output the same values, so that we can predict the behavior of the hardware easily. We need a scaling factor $\gamma_r = \frac{2^{M-1}}{\gamma_e}$. In case 1), $\gamma_{r_{prev}}$ in the previous layer is initialized with 1, and in case 2), it is initialized with 255.

First, we have to correct γ which has a negative value in the batch normalization, which makes residual-binarization in inversely proportional. The weights in the previous layers, moving-means μ, and γ are element-wise-multiplied with the sign vectors of the γ. Then, after calculating the current $\gamma_{r_{cur}}$, the parameters in batch normalization are updated as:

$$\begin{aligned} \gamma' &= \gamma \cdot \gamma_{r_{cur}} \\ \beta' &= \beta \cdot \gamma_{r_{cur}} \\ \mu' &= \mu \cdot \gamma_{r_{prev}} \\ \sigma' &= \sqrt{(\sigma^2 + \epsilon) \cdot \gamma^2_{r_{prev}}}, \end{aligned} \tag{3.5}$$

where $\{\gamma, \beta\}$ are multiplied by the current scaling factor, $\{\sigma, \mu\}$ are dependent on the previous layer's output, and thus they are multiplied by the previous scaling factor. Finally, τ and α are calculated by the scaled parameters, and we can use the integer $\gamma_e (= 2^{M-1})$ in hardware implementation.

3.3 Processing Element

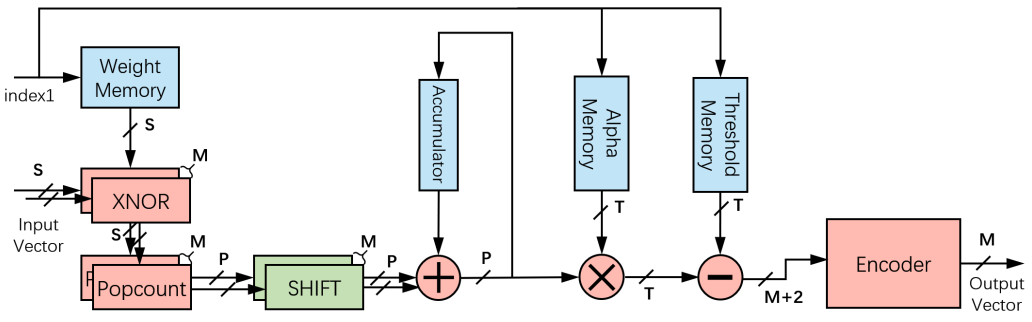


Figure 3.2: Processing Element in our design.

As mentioned above, we reduced the components in PE in Figure 3.2. While index1 is PE index, S is data width of input (SIMD width), P is data width of

popcount accumulator, T is data width of all of the fixed-point values, and M is the number of residual-binarization levels. Before adding the popcounted value to the accumulator, we process the logical left shift simply by wire connections and use adder-tree to sum different levels' shifted values. We not only remove the $M-1$ DSP48s for γ_e and popcounted values, but also reduce the amount of DSP48s for α. ReBNet needs two DSP48s to multiply T-bit fixed-point value which is the output of the MAC module and T-bit fixed-point α. Our design, on the other hand, only requires a DSP48 to multiply P-bit integer accumulated value and T-bit fixed-point α, where T is supposed to be 24, and P is the width of the integer part of T, i.e., 16. In most cases, since α is a small value, it is desirable to allocate more bits to the decimal part, e.g., 12 bits for the integer part and 12 bits for the decimal part. Then, the calculated, temporary value is subtracted by $\tau \cdot \alpha$ which can be pre-calculated. The computation so far is shown as:

$$\sum(\sum_{i=1}^{M}(XnorPopcount(\vec{b}_{e_i}, \vec{w}) << (M-i))) \cdot \alpha - \tau \cdot \alpha. \tag{3.6}$$

Then we deliver the result of Formula (3.6) to the encoder. Different from ReBNet which needs to input all of the bits for getting the correct result from comparing the very close values, we only need $M+2$ bits, which are a sign bit, the least significant M bits in the integer part, and 1-bit decimal part, to encode correctly. If there is valid information in not-selected (or higher) bits in the integer part, we need to set the maximum or minimum value to the least significant M bits: 1) When the sign is positive, the least significant M bits are to be the maximum if there is at least 1 in the unselected (higher) bits in the integer part. We can check it by element-wised OR gate; 2) When the sign is negative, the least significant M bits are the minimum if there is at least 0 in the unselected integer part. We can check it by element-wised AND gate. The fractional part has a constant value of 1, which is 0.5 in decimal since this is always "1" in the fractional part after threshold subtraction. We use constant 0.5 to represent all of the fractional part and it can avoid comparison between the same values with γ_e.(Figure 3.3)

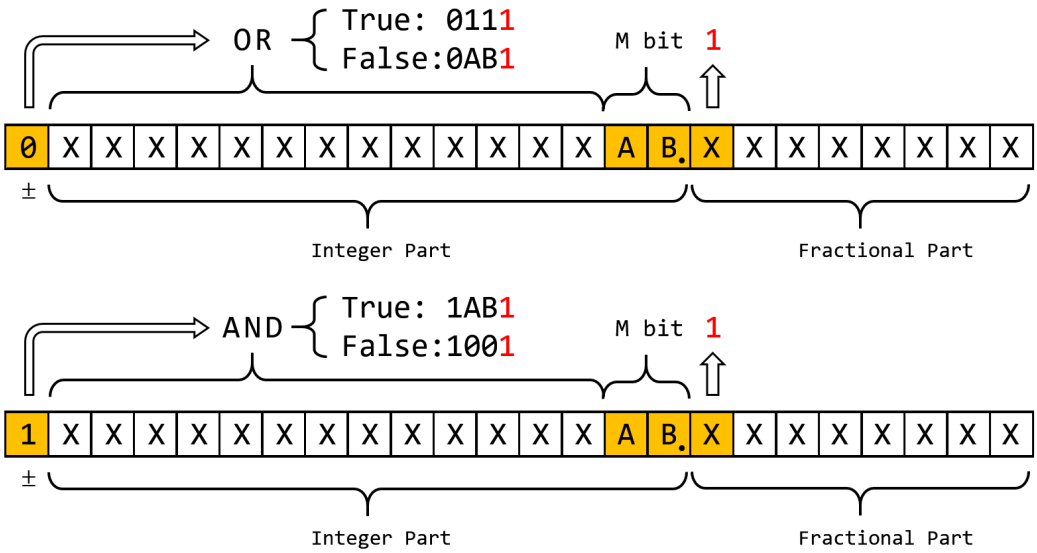


Figure 3.3: Valid bits for encoder

In addition, we apply throughput optimization in the processing element so that the Initiation Interval (II) of each PE decreases from M to 1. This means that the throughput of the design increases by M. In ReBNet, they input the data in the interleaved manner, while, in our design, data are inputted in parallel. This brings overhead of $M-1$ more popcount modules and larger dataflow control logic between layers.

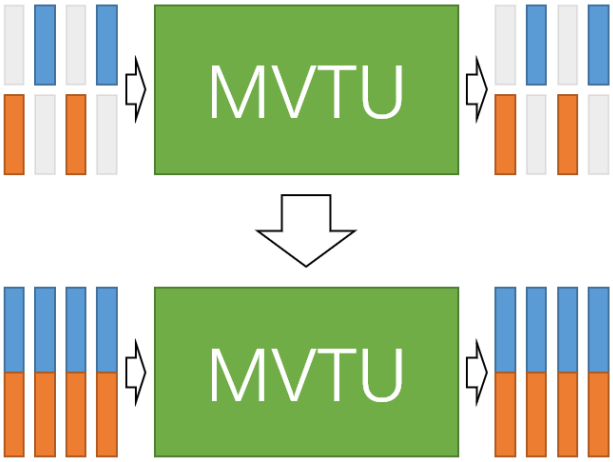


Figure 3.4: Throughput optimization

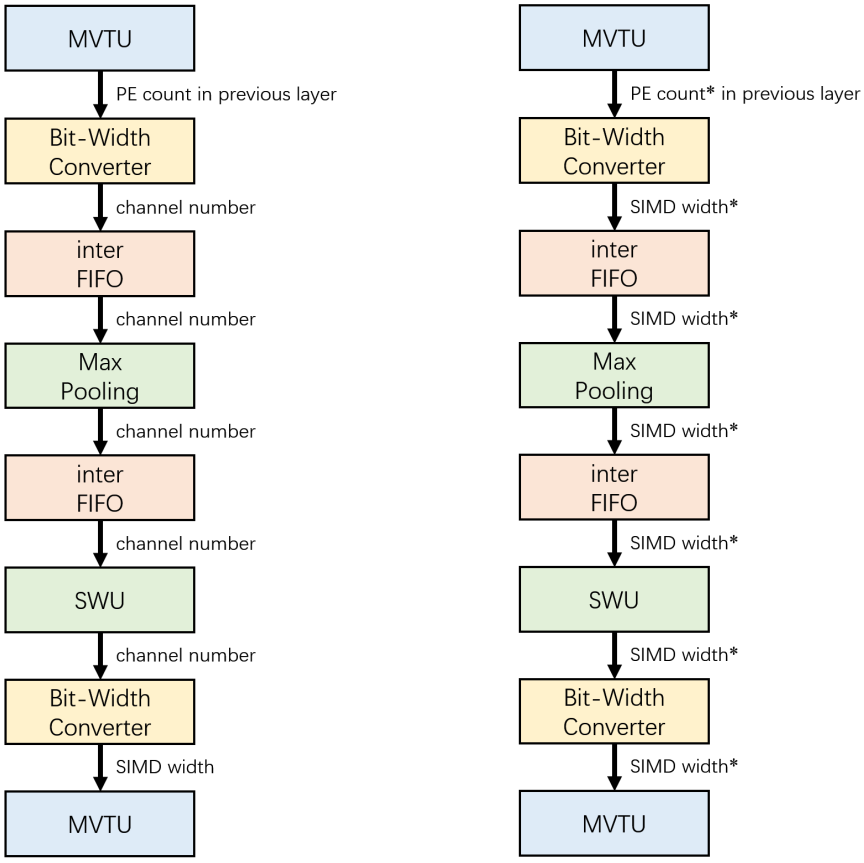
Figure 3.4 is a simple example when $M = 2$. The top is the original

ReBNet and the bottom is our design. We can see that, our design processes 4 SIMDs in 4 clock cycles, and ReBNet only processed 2 SIMDs. However, our design needs 2× bit width, it will require more Block RAM to provide so wide port, and each Block RAM will use less capacity, especially for the input buffer in SWU. We propose how to mitigate this problem in the next section.

3.4 Data Stream with Adaptive Bit Width

In SWU, it needs to buffer at least $K+1$ lines of input feature map(K is kernel size of Convolutional Layers). K lines are for sampling and the extra 1 line is for inputting new data in parallel. The total capacity required is fixed, but the wider stream makes data be divided into more Block RAMs. For example, when the size of input feature map is $20 \times 20 \times 256$, K is 3, and M is 2, the required buffer capacity is $(3+1) \times 20 \times 256 \times 2 = 20480$. The Block RAM of Xilinx modern device is RAMB36E1/RAMB36E2, which has a capacity of 32+4Kbit, and 2 ports with the width of 32+4 bit(the extra 4-bit width and 4Kbit capacity can be used in 9/18/36/72 bits word mode only). ReBNet partitions SWU buffer into $(K+1) \times M$ smaller buffers. We can calculate the required Block RAM amount of ReBNet as $256 \div 64 \times (3+1) \times 2 = 32$, where each Block RAM only uses the capacity of 640bit, 1.74% of all. In our throughput optimized design, there is no need to multiply M but the required port width will be M times. So, the required Block RAM amount is $512 \div 64 \times (3+1) = 32$. It is the same as ReBNet, but there are more hardware recourse required in interFIFOs and Bit-Width Converters.

We propose Data Stream with Adaptive Bit Width to overcome it. Figure 3.5 illustrates the data stream from MVTU of the previous layer to MVTU of the next layer, and the MaxPooling is optional. We can see that ReBNet resumes the width of the data stream to the number of channels, and divides it to SIMD width of the next MVTU at last. So the MaxPooling and SWU are required to process the wide data. Our optimized design is shown in Figure 3.5b, where we change the width of the data stream to the SIMD width of the next layer very early. All of the dataflow control logic will be smaller and the second Bit-Width Converter only needs to bypass the input. Because the bit width will be changed when SIMD width changed(the degree of parallelism changed), this optimization will not slow down the



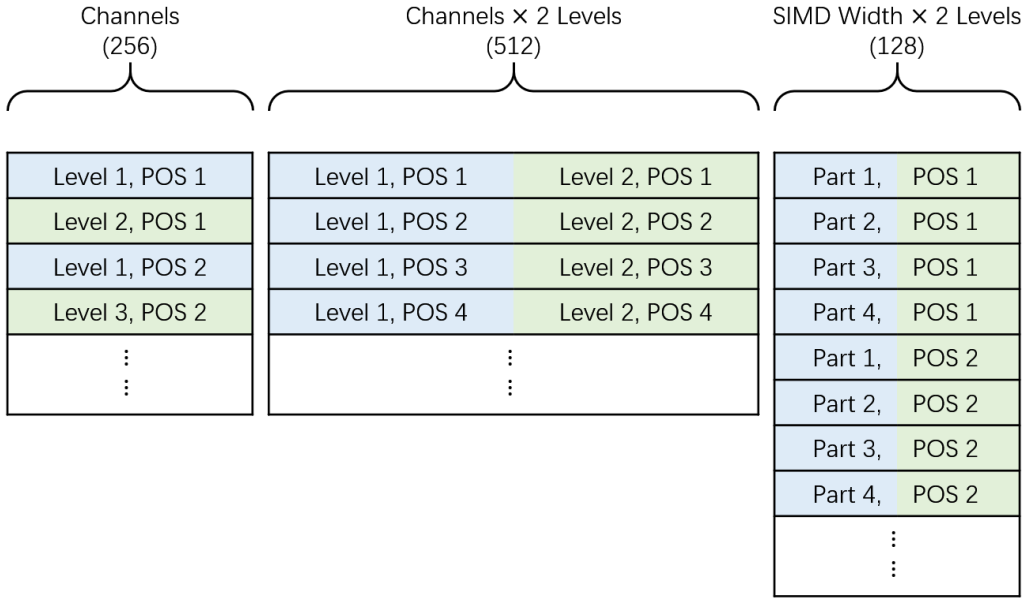
(a) ReBNet (b) Our optimized design

* should be multiplied by level number M.

Figure 3.5: Data Stream

whole system. Figure 3.6 shows the data in inter FIFO when the number of channels is 256 and the SIMD width of the next layer is 64, where POS means the position of the feature map. The bit width decreased to 25% of the unoptimized case and less than ReBNet.

The SWU does not load or write all of the lines at the same time, so the throughput will not decrease when disabling the buffer partition. In the example above, it only needs 2 Block RAMs after optimization when the SIMD width of the next layer is 64. If the SIMD width of the next layer is 32, the required Block RAM will decrease to 1.



(a) ReBNet (b) Before Opt. (c) Post Opt.

Figure 3.6: Data in inter FIFO

Chapter 4

Experiments

We use Keras [27] with TensorFlow [28] backend to train our models with methods in BinaryNet. We apply our method to the open-sourced library of ReBNet and use Vivado HLS in Vivado Design Suite [29] to perform high-level-synthesis for the IP core of the accelerator we designed. Then, logic synthesis, implementation, and bitstream generation are done on Vivado.

We implemented different accelerators for several small data sets: MNIST [5], CIFAR-10 [6] and SVHN [7], and a large data set: ImageNet (ILSVRC2012) [8], and compare them with ReBNet for the same data sets. There are typical 2 parts in a dataset: train dataset and test dataset, where train dataset is for training the model, and the test dataset is for evaluating the accuracy and data elements are different from train dataset.

Figure 4.1: Example of MNIST dataset [5]

MNIST [5] is a database of handwritten digits, and there are 60,000 training images and 10,000 test images. The images have a size of 28×28. The example of the MNIST dataset is shown in Figure 4.1.

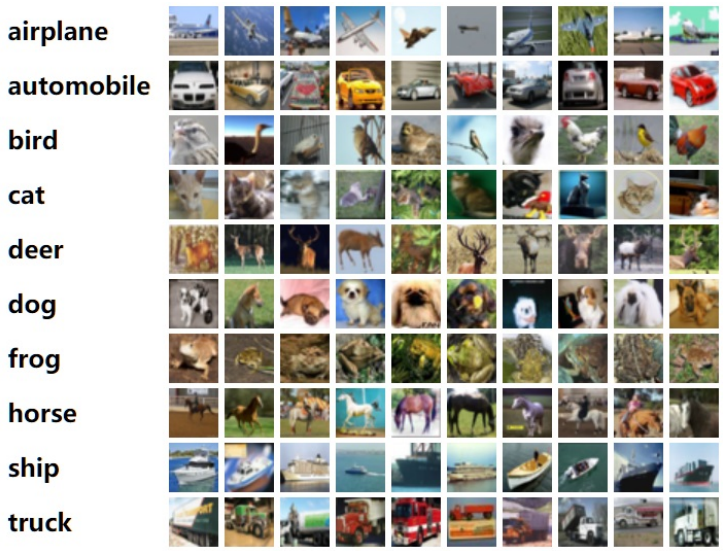


Figure 4.2: Example of CIFAR-10 dataset [6]

CIFAR-10 [6] is a database of objects in 10 classes, and there are 50,000 training images and 10,000 test images. The size of images is 32×32, and each class contains 6,000 images. The example of the CIFAR-10 dataset is shown in Figure 4.2.

Figure 4.3: Example of SVHN dataset [7]

The Street View House Numbers (SVHN) [7] is a database of digits images, and there are 73,257 training images, 26,032 test images and 531,131 extra images for training. The size of images is 32×32. The example of the SVHN dataset is shown in Figure 4.3.

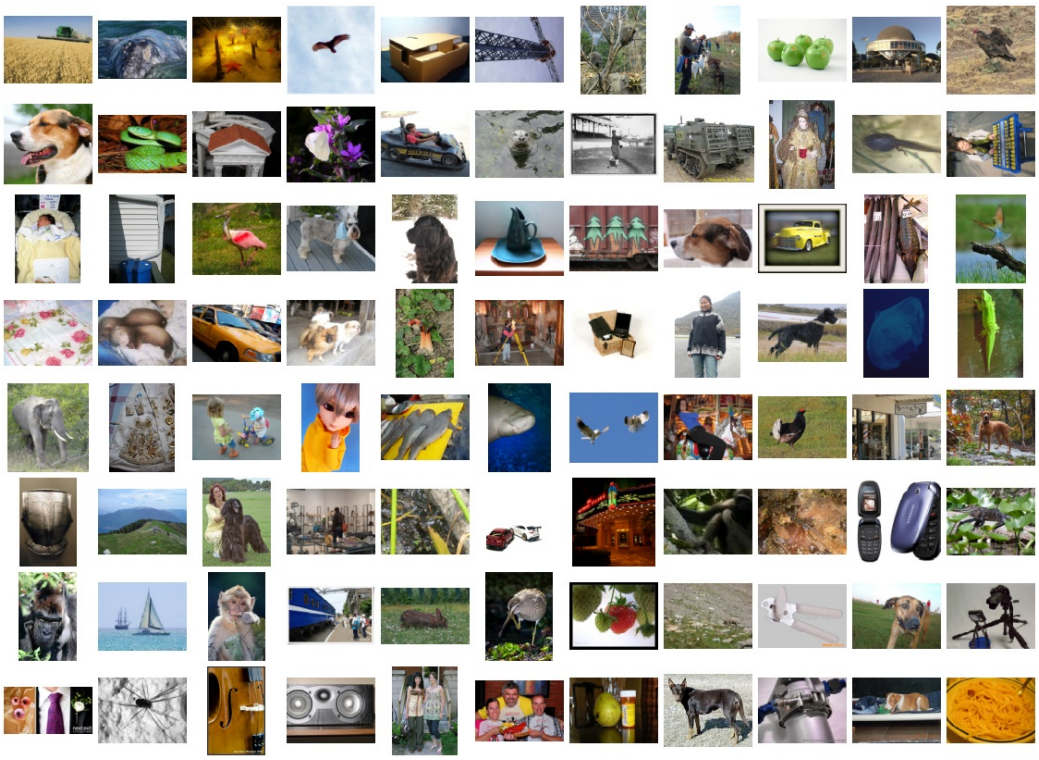


Figure 4.4: Example of ImageNet dataset [8]

ILSVRC 2012 [8], also known as "ImageNet", is a database of images which illustrate "synonym net" in 1,000 classes. There are 1,281,167 training images and 100,000 test images. Images are in different sizes. The example of the ImageNet dataset is shown in Figure 4.4.

We target several FPGA devices with different sizes in implementation comparison. The resources of the target devices are presented in Table 4.1 where 7z020 is a resource-limited device and 7z100 is a large device which provides a large amount of DSP48s. vu095 is large which provides a large amount of Block RAMs to implement ImageNet. The highest degree of parallelism described in FINN for CNNs architecture of small data sets can be implemented on the large device 7z100. However, the same degree is impossible on the resource-limited device 7z020. Instead, we tried to find the

possibly highest degree of parallelism for $M = 2$ and $M = 3$ on the resource-limited device 7z020. On the other hand, the large data set, ImageNet, can only be implemented with a fixed degree of parallelism on the vu095.

To check the effect of each optimization, we also design several scenarios to compare with the original ReBNet and unoptimized design.

At last, we show the throughput on the real development kit and compare it with the theoretical value.

Table 4.1: Target FPGA devices.

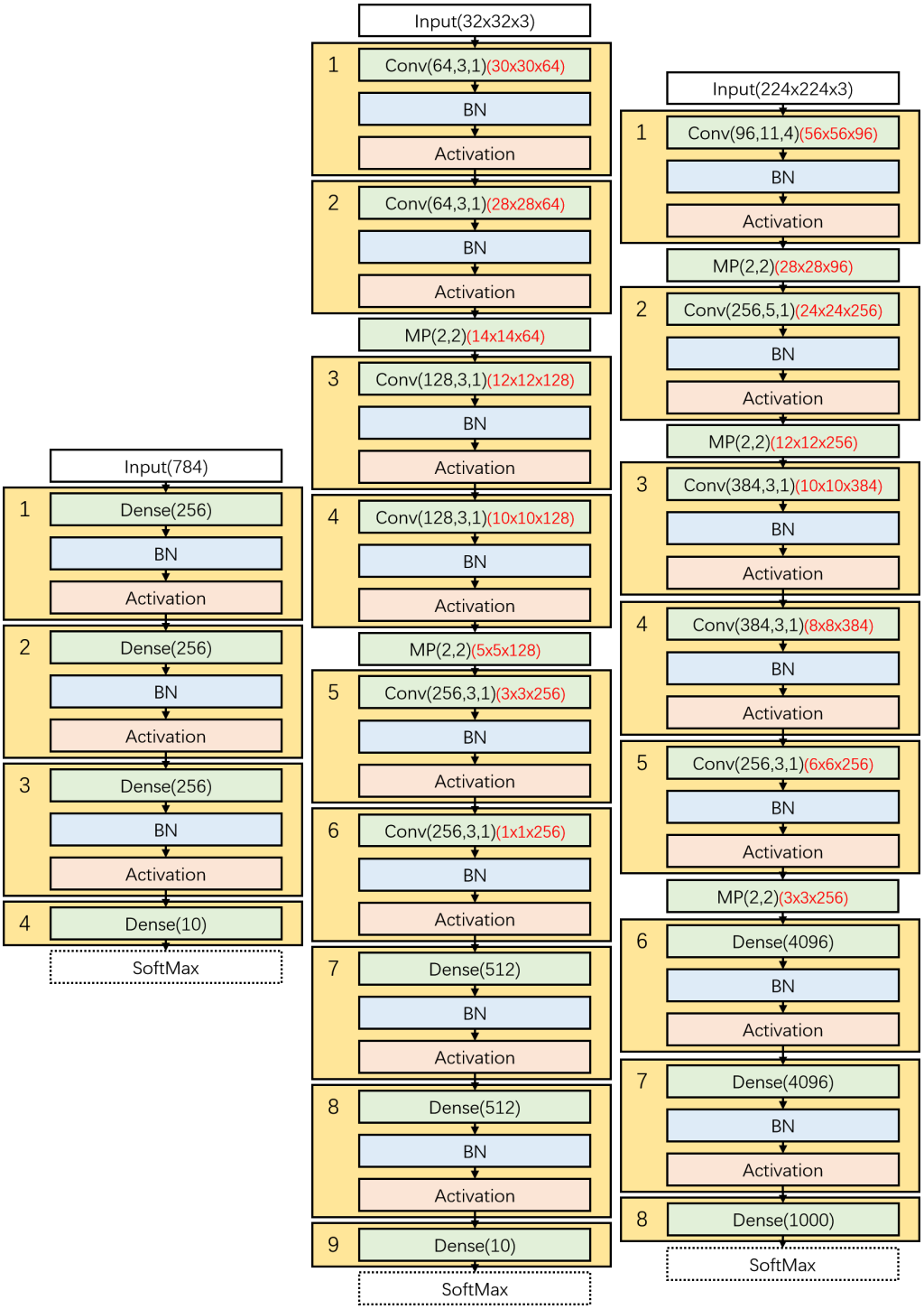
FPGA device	LUT	FF	DSP48	BRAM
xc7z020-clg484-1	53,200	106,400	220	140
xc7z100-ffg1156-2	277,400	554,800	2,020	755
xcvu095-ffva2104-2	537,600	1,075,200	768	1,728

4.1 Training models

We adopt neural network architectures similar to ReBNet [3] as those in Figure 4.5. There are 3 neural network architectures; Arch1 is for MNIST, Arch2 is for CIFAR-10 and SVHN, and Arch3 is for ImageNet. BN represents batch normalization, $Dense(A)$ represents Fully connected layer with A neurons (nodes), $Conv(C, K, S)$ represents Convolutional Layer with C output channels, $K \times K$ kernel size and Stride is $S \times S$, and $MP(K, S)$ represents MaxPooling with $K \times K$ sampling window and $S \times S$ Stride. Activation Functions depend on scenarios; in our optimized case, it will be Isometric Residual-Binarization, in the original case of ReBNet, it will be Residual-Binarization, and in floating-point case, it will be *tanh*. The number on each block of a convolutional layer or fully connected layer will be used at hardware implementation to describe the degree of parallelism. Dropout is added after some of the Activation Functions with a low probability to reduce overfitting during training. The gradient descent optimization algorithm used during training is Adam [30].

4.1.1 Small Datasets

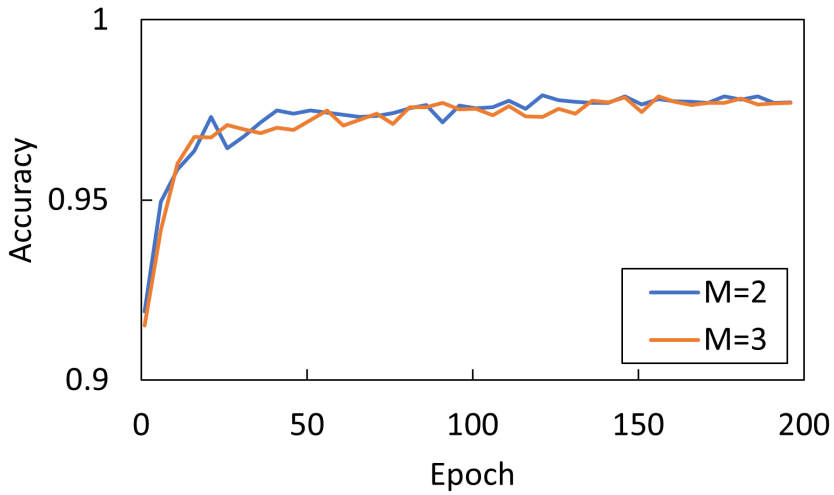
MNIST was trained on Arch 1 which only contains fully-connected layers. CIFAR-10 and SVHN are trained on Arch 2, in which all of the convolutional



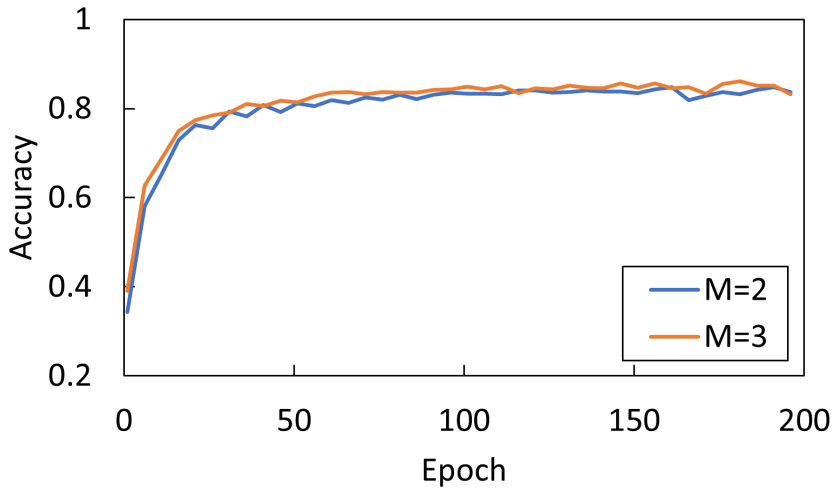
(a) Arch1 (b) Arch2 (c) Arch3

Figure 4.5: Neural Network Architectures

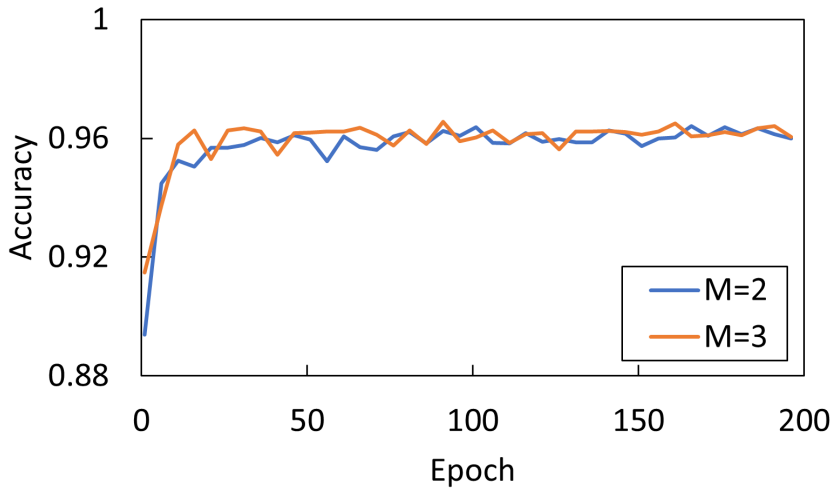
layers have a kernel size of 3×3 and stride of 1, and maxpooling layers have a kernel size of 2×2 and stride of 2. While the training in ReBNet performs batch normalization after the last fully-connected layer, our training method cuts the batch normalization, since after this batch normalization, there is no binarize activation function and it cannot be implemented as a threshold subtraction. In fact, this batch normalization does not help the model to get higher accuracy but has effects of speeding up the accuracy improvement in the early training stages. As for inference by hardware, the hardware implementation of ReBNet skips this batch normalization, causing the accuracy to be undulated. Considering this problem, we use the architectures without this batch normalization in both training and inference. Before feeding the dataset to a neural network model, the images should be preprocessed. The images of MNIST are monochrome, We change them to 2 colors: lower than 127.5 to black and upper than 127.5 to white. The images of CIFAR-10 are in color. We do not change the color values, but randomly apply shift, horizontal flip to them for Data Augmentation. For SVHN, the images contain numbers. We only apply random shift.



(a) MNIST



(b) CIFAR-10



(c) SVHN

Figure 4.6: Learning Curves

The accuracy of a neural network is dependent on architecture and training epochs. We fix the training epochs to 200 for all of the small datasets. During model training, we evaluate the accuracy of the test dataset after each epoch and show it as learning curves in Figure.4.6.

The average accuracy of $M = 3$ is higher than $M = 2$ in the CIFAR-10 and SVNH dataset, but there is no significant advance in the MNIST dataset. $M = 2$ is enough to get high enough accuracy in MNIST. Table 4.2 summarizes the highest accuracy for each dataset in different residual-binarize levels. The accuracy of FINN is quoted from the paper of ReBNet. It is found that our method of isometric residual-binarization achieves accuracy similar to ReBNet, and much higher than FINN, but there is still a gap with the floating-point case.

Table 4.2: Accuracy Comparison of small datqasets

	FP32	FINN	M	ReBNet	This Work
MNIST	0.9822	0.9583	2	0.9799	0.9801
			3	0.9799	0.9792
CIFAR-10	0.8903	0.801	2	0.8469	0.8501
			3	0.8618	0.8616
SVHN	0.9765	0.949	2	0.9677	0.9654
			3	0.969	0.9665

4.1.2 Large Dataset

Large Dataset, ImageNet, was trained on Arch3. As same as small datasets, we also cut the output batch normalization to make the accuracy of hardware implementation predictable. The preprocessing we used in the training of ImageNet is a fast mode of preprocessing from inception [10] which is widely used in training models for complex datasets. The training epochs are fixed to 200, and Table 4.3 summarizes the highest accuracy for each method, where residual-binarize levels are fixed to 2 in our design and ReBNet. Top-1 means the rate of the class with the highest output value was equal to the ground true, and sorting all output and getting the highest 5 classes, Top-5 accuracy means the rate of ground true in those 5 classes. Learning curves are shown in Figure.4.7. Our design gets higher accuracy than ReBNet, but compared with the floating-point case, the binarized method still gets much lower accuracy.

Table 4.3: Accuracy Comparison of ImageNet

		FP32	FINN	ReBNet	This Work
ImageNet	Top-1	0.5151	0.279	0.3771	0.3913
	Top-5	0.7483	-	0.6194	0.6374

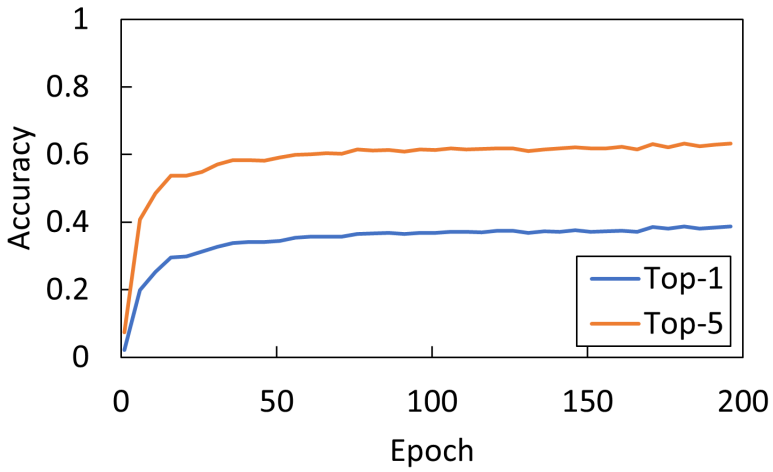


Figure 4.7: Learning Curves of ImageNet

4.2 Convert Models to Binary Weight

The format of output models of Keras is HDF5. They only can be loaded by software neural network frameworks. The parameters in them are floating-point numbers, and we need to convert them to binary values and calculate the threshold for the binary batch normalization. Then, it is necessary to generate binary weight files from them, and binary weight files can be loaded by hardware. The process flow is shown in Figure 4.8.

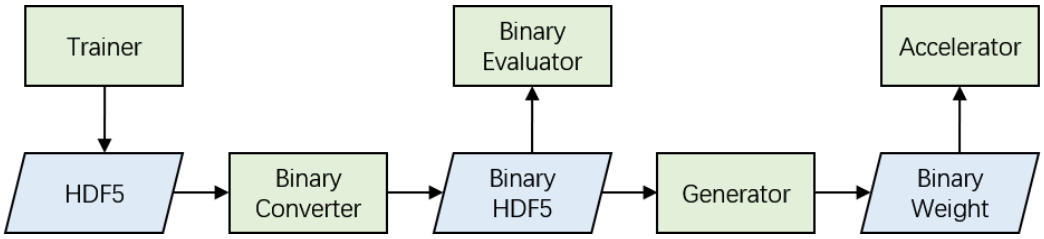


Figure 4.8: Conversion of Weights

The "Binary Converter" converts the weights of fully connected layers and convolutional layers to binary values, and fixes the problem when γ in batch normalization is negative by inverting the moving-average and weight in that block. Then it calculates the threshold τ and α from γ_e integer scaled parameters, and stores $\tau \cdot \alpha$ and α with limited precision in HDF5 files, where

$\tau \cdot \alpha$ has 24-bit precision with 8 bit fractional part and α has 24-bit precision with 12 bit fractional part. The "Binary HDF5" files can be loaded by Binary Evaluator with "mirror models" which have the same architectures as in Figure 4.5, but Batch Normalization is implemented as threshold subtraction and γ_e is integer number. The behavior of those mirror models is completely the same as the hardware accelerator we proposed. Because limiting the precision will slightly change the result of classification. we can use those mirror models to evaluate the accuracy after binary conversion. Table 4.4 shows the accuracy change from Table 4.2. Some of them become higher but some of them become lower. We will use these detailed classification results to check if hardware accelerators run correctly.

Table 4.4: Accuracy changing

dataset	M	before	after	Δ
MNIST	2	0.9801	0.9803	0.0002
	3	0.9792	0.9793	0.0001
CIFAR	2	0.8501	0.8497	-0.0004
	3	0.8616	0.8632	0.0016
SVNH	2	0.9654	0.9645	-0.0009
	3	0.9665	0.9670	0.0005

The "Generator" builds binary weight files for hardware accelerators. We modified an open-sourced generating script from the FINN framework to generate the binary weight files we need. The "Generator" splits neurons of weights to PEs and synapses of weights to SIMD lanes. So the number of neurons in the previous layer must be an integer multiple of SIMD width, and the number of neurons in the current layer must be an integer multiple of PE count. To relax this limitation, we make "Generator" insert "bubble data", and change threshold values to process correctly. However, it works well with fully connected layers only at the present time. With this modification, we can implement hardware accelerators with high flexibility of the degree of parallelism.

4.3 Simulation

There are many types of simulation in hardware accelerator design: behavioral simulation, pre-synthesis simulation, post-synthesis simulation, pre-implementation simulation, and post-implementation. But in HLS, only "C simulation" and "C/RTL cosimulation" are required, where "C simulation" is a special type of simulation only in HLS project to test and verify the correctness of the design, and "C/RTL cosimulation" is composed of C simulation with data port record and behavioral simulation. It is used to make sure RTL codes generated by HLS have the same behavior as that with C codes, and check timing, data path, etc.

In the simulation, we used binary weights generated through the conversion process described in the previous section. Figure 4.9 shows the output of simulation and mirror model Binary Evaluator, where the parameter of MNIST(M=3) for Arch1 and parameter of CIFAR-10(M=3) for Arch2. The output index larger than 10 will be ignored in the next stage of processing.

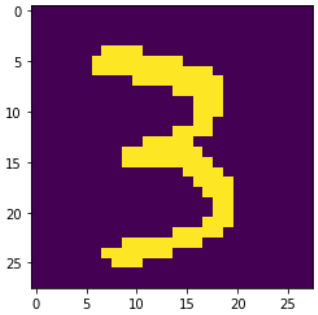
In the simulation of Arch1, the input image is "3", and the 4^{th} output (from 0 to 9) has the maximum value, so the classification result is 3. However, the derailed result of Arch1 simulation is different from Binary Evaluator because we insert bubble data to get 0.5× speed up. It results in a bias valued 8 from the output of the Binary Evaluator.

In the simulation of Arch2, the input image is a deer, and the 5^{th} output has the maximum value. According to Figure 4.2, the 5^{th} class is deer. Therefore, the result is correct. Because we did not insert bubble data into Arch2 simulation, the result is as same as Binary Evaluator.

Figure 4.10 is waveform from "C/RTL cosimulation" of Arch2. We did not initialize weight memory because it will make the waveform too large. We input 10 images in this simulation. The waveform of single-bit signal in the upper part is full signal of interFIFOs(active low), and we can see that the interFIFO in the red frame is full after 1 image is processed. The waveform of bus in the lower part is the value of accumulator in the first PE in each MVTU. When the value is hold, it means that this MVTU is in waiting state. In the blue frame, the waiting time becomes longer obviously when processing the last 3 images. It makes the accelerator unable to reach theoretical throughput. We should adjust FIFO depth to make the accelerator close to theoretical throughput.

```
[[-526. -132.  206. 1318. -254.   38. -392.   94.  168. -100.]]
```

output of Binary Evalutor



input image

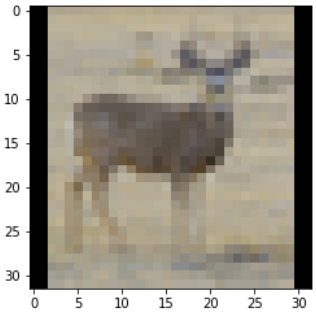
```
acc -518
acc -124
acc 214
acc 1326
acc -246
acc 46
acc -384
acc 102
acc 176
acc -92
acc -98
acc -98
acc -98
acc -98
acc -98
acc -98
```

output of C simulation

(a) Arch 1

```
[[  74. -470.  510.  436. 2698.  456.  298.  604. -276. -454.]]
```

output of Binary Evalutor



input image

```
acc 74
acc -470
acc 510
acc 436
acc 2698
acc 456
acc 298
acc 604
acc -276
acc -454
acc 244
acc 244
acc 244
acc 244
acc 244
acc 244
```

output of C simulation

(b) Arch 2

Figure 4.9: Simulation

Figure 4.10: Waveform from "C/RTL cosimulation"

4.4 Implementation

We implement the hardware accelerator module for different FPGA devices, where there is no Processing System, I/O constraints or AXI interconnect. The "repeat" argument that allows the hardware accelerator to process multiple images in the pipeline is disabled, which makes it easy and clear to analyze throughput and iteration. Enabling the "repeat" parameter will slightly increase the use of hardware resources to implement additional logic to count data frames and calculate loop cycles.

4.4.1 Resource-limited Device

In this subsection, we try to find the possibly highest degree of parallelism for $M = 2$ and $M = 3$ on the resource-limited device 7z020 for Arch1 and Arch2, and compare the frequency, throughput, and power efficiency. According to FINN [20], we use $Fold$ to describe the degree of parallelism, where $Fold$ means the number of iterations of an MVTU process for one picture. $Fold$ is determined by SIMD width and PE count. Larger SIMD width makes synapse iteration smaller, and larger PE count makes neuron iteration smaller. On the other hand, in ReBNet, Initiation Interval(II) of PE is M, which takes $Fold \times M$ iterations to process one picture. The layer with the largest $Fold$ can be the bottleneck of the dataflow. Table 4.5 shows the $Fold$, SIMD width and PE count configuration with which our design and ReBNet can be implemented on 7z020 for each architecture.

As mentioned before, the MaxPooling in ReBNet runs out of Block RAMs (or Distributed RAMs) in 7z020, and cannot be implemented in any parallelism setting. Thus, we disabled the parallel processing in MaxPooling for ReBNet on 7z020. However, this change does not become the bottleneck in implementations on small devices.

Figure 4.11 shows the resource usages on each degree of parallelism and Table 4.6 includes maximum frequency, power usage of chip, and maximum throughout calculated via the maximum frequency according to FINN [20]. The maximum throughput can be calculated by $\frac{\text{frequency}}{\text{MAX}(Fold)\cdot\text{InitiationInterval}}$. All of the results are from reports of Vivado HLS or Vivado with a target frequency of 200MHz.

Our design reaches much higher degrees of parallelism in resource-limited device, 7z020, and each PE consumes fewer resources on average because of

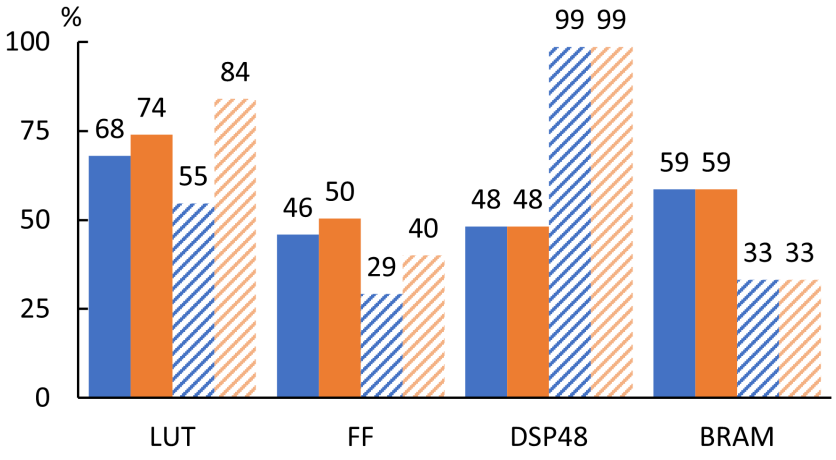
Table 4.5: PE court, SIMD widht and *Fold* for 7z020

(a) Arch 1

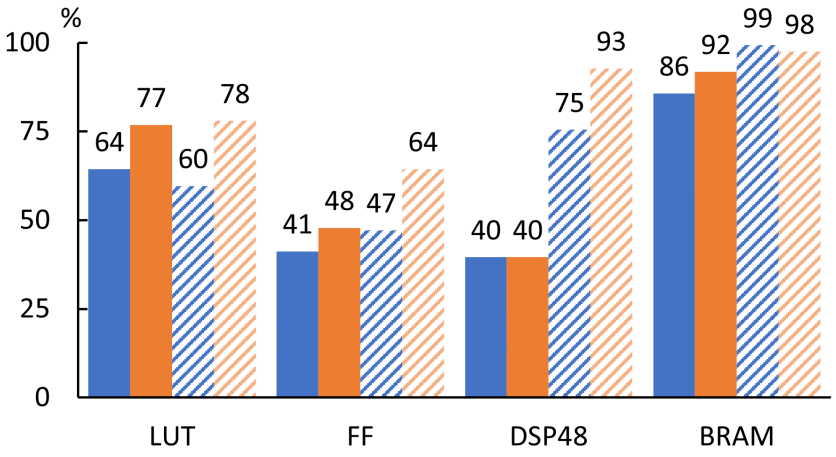
Layer	ours			ReBNet		
	PE	SIMD	*Fold*	PE	SIMD	*Fold*
1	48	64	78	20	64	169
2	32	24	96	16	20	208
3	24	32	88	20	16	208
4	4	12	88	4	20	208

(b) Arch 2

Layer	ours			ReBNet		
	PE	SIMD	*Fold*	PE	SIMD	*Fold*
1	16	3	32400	8	3	64800
2	32	32	28224	16	32	56448
3	16	32	20736	8	32	41472
4	16	32	28800	8	32	57600
5	4	32	20736	2	32	41472
6	1	32	18432	1	16	36864
7	1	4	32768	1	2	65536
8	1	8	32768	1	4	65536
9	1	1	8192	1	1	8192



(a) Arch 1



(b) Arch 2

Figure 4.11: Normalized hardware utilization on 7z020

our efficient design and optimization. Since our design does not run out of the DSP48, extra LUTs are not necessary to implement multipliers, which leads to higher maximum frequency, especially compared with RebNet in Arch 2 when $M = 3$. Because of the higher degree of parallelism and partitioned weights, our design uses more BRAMs in Arch 1. In contrast, in Arch 2, our design uses fewer BRAMs as a result of optimization on the buffering method in SWUs and MaxPooling, even though the design is implemented with two-fold degree of parallelism. Our design achieves 8 times higher throughput than ReBNet on average with 7z020.

Table 4.6: Frequency, throughput and power usage for 7z020

(a) Arch 1

	This Work		ReBNet	
M	2	3	2	3
LUT	36172	39333	29097	44670
FF	48780	53584	31052	42613
DSP48	106	106	217	217
BRAM36	82	82	46.5	46.5
power(W)	1.353	1.529	1.052	1.994
freq(MHz)	145.31	143.74	106.85	99.52
thr.put(kFPS)	1513.61	1497.29	256.85	159.49

(b) Arch 2

	This Work		ReBNet	
M	2	3	2	3
LUT	34261	40883	31727	41471
FF	43865	50875	50164	68509
DSP48	87	87	166	204
BRAM36	120	128.5	139	136.5
power(W)	1.41	1.473	1.188	1.61
freq(MHz)	134.28	129.03	136.44	53.70
thr.put(kFPS)	4.10	3.94	1.04	0.27

4.4.2 Maximum parallelism on 7z100

We implement the maximum degree of parallelism described in FINN on 7z100, and the settings are shown in Table 4.7. Large SIMD width is considered to use less hardware resource than large PE count, but the maximum SIMD width of FINN framework is 64 due to 64-bit AXI data bus and no 128-bit integer type in ARM CPU environment and python numpy module which is used to generate binary weight file.

Table 4.7: PE court, SIMD widht and $Fold$ for 7z100

(a) Arch 1

Layer	Maximum		
	PE	SIMD	$Fold$
1	256	64	13
2	64	64	16
3	64	64	16
4	4	64	16

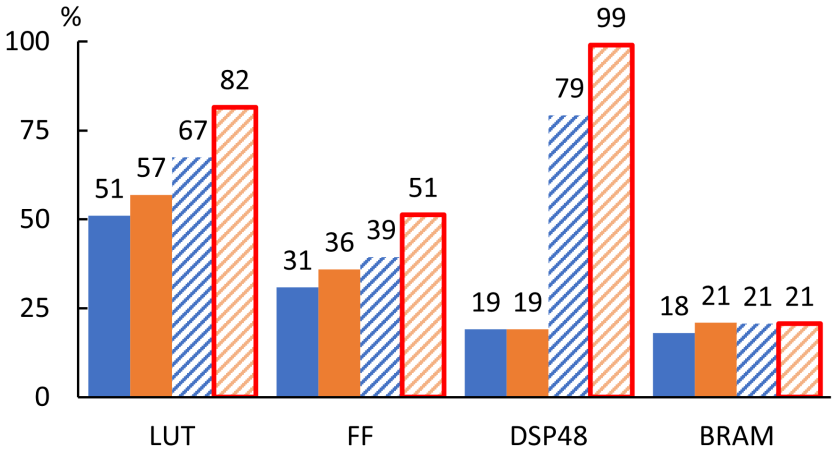
(b) Arch 2

Layer	Maximum		
	PE	SIMD	$Fold$
1	64	3	8100
2	64	64	7056
3	32	64	5184
4	32	64	7200
5	8	64	5184
6	2	64	4608
7	1	16	8192
8	1	32	8192
9	1	1	8192

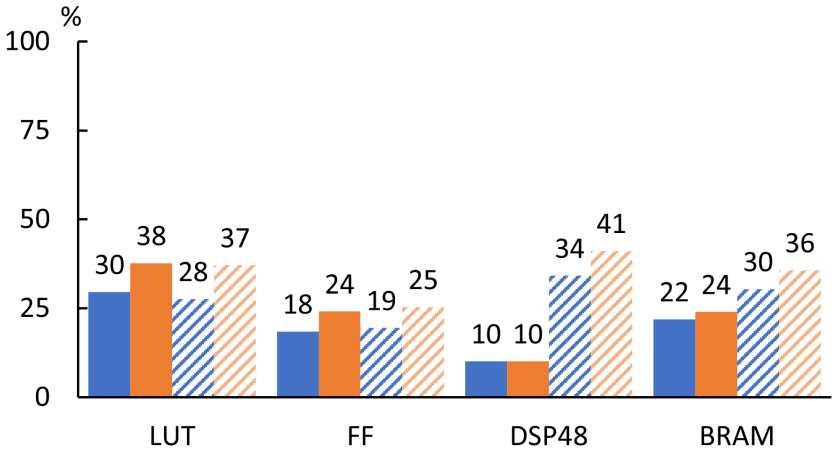
Figure 4.12 shows the resource usages and Table 4.8 includes maximum frequency, maximum throughout calculated via the maximum frequency, and power usage of the chip, where a red results with frame are post-placement results due to routing failure. All of the results are from reports of Vivado

HLS or Vivado with a target frequency of 200MHz.

In the usage of Arch2, our design uses more LUTs than ReBNet in both M=2 and M=3. The reason for this situation is the advantages of our design will become smaller with larger SIMD width since more LUTs are used for adder trees. We will analyze this in the following sections. In the usage of Arch1, ReBNet uses much more hardware resources because it does not include a special MVTU for the first layer which inputs 1 level data and outputs M levels data. And the PE count of the first layer is quite large to make $Fold$ to be minimum, which causes a large overhead for ReBNet.



(a) Arch 1 on 7z100



(b) Arch 2 on 7z100

Figure 4.12: Normalized hardware utilization on 7z100

In implementations on the large device, 7z100, Arch 1 of ReBNet with $M = 3$ fails to complete Initial Routing due to high congestion. Instead, we use post-placement utilization information, which is depicted with a red frame in Figure 4.12. No frequency or throughput information is obtained for this scenario. Although ReBNet does not exhaust DSP48s in 7z100, the scenario with the highest usage of 99% requires 2,001 out of the total 2,020 DSP48s in 7z100. We can see that, as mentioned before, our design uses around $\frac{1}{M+2}$ DSP48s of ReBNet, and usage of the other components is also

lower. The throughput of our design is M times higher than ReBNet in each residual-binarization level. In addition, the usages between resource types are well balanced in our method and the power usage in our design is much lower than ReBNet.

Table 4.8: Frequency, throughput and power usage for 7z100

(a) Arch 1

	This Work		ReBNet	
M	2	3	2	3
LUT	141509	157939	187041	226320
FF	171064	199415	218616	284579
DSP48	385	385	1601	2001
BRAM36	136	158	156	156
power(W)	5.083	4.905	8.121	-
freq(MHz)	200.92	193.91	124.30	-
thr.put(kFPS)	12557.77	12119.45	3884.40	-

(b) Arch 2

	This Work		ReBNet	
M	2	3	2	3
LUT	82126	104627	76562	102914
FF	102480	133779	107922	140622
DSP48	204	204	690	831
BRAM36	165.5	181	229.5	269.5
power(W)	2.912	3.88	3.493	4.247
freq(MHz)	200.24	202.10	200.44	201.01
thr.put(kFPS)	24.44	24.67	12.23	8.18

4.4.3 Implement Arch3 on vu095

We implemented Arch3 which is for ImageNet on vu095. Model of Arch3 contains 34,056,224 weights, and vu095 has a large amount of Block RAMs so that all the weights can be stored on-chip. The PE count and SIMD width settings are shown in Table 4.9, and the settings provide the minimum $Fold$

of the FINN framework. The loop cycle of SWU in the first layer is 379456, which limited the minimum $Fold$.

Table 4.9: PE court, SIMD widht and $Flod$ of Arch3

Layer	PE	SIMD	$Fold$
1	16	33	206976
2	32	32	345600
3	8	32	345600
4	8	32	331776
5	4	32	248832
6	1	32	294912
7	2	32	262144
8	1	16	256000

We only implemented Arch3 with $M = 2$. The utilization is illustrated in Figure 4.13 and the detailed results are shown in Table 4.10. Over 1000 Block RAMs are used to store weights, and our design uses less than half of the hardware resources than ReBNet overall except Block RAM. The throughput is calculated by 379456(loop cycle of SWU in the first layer) instead of the maximum $Fold$.

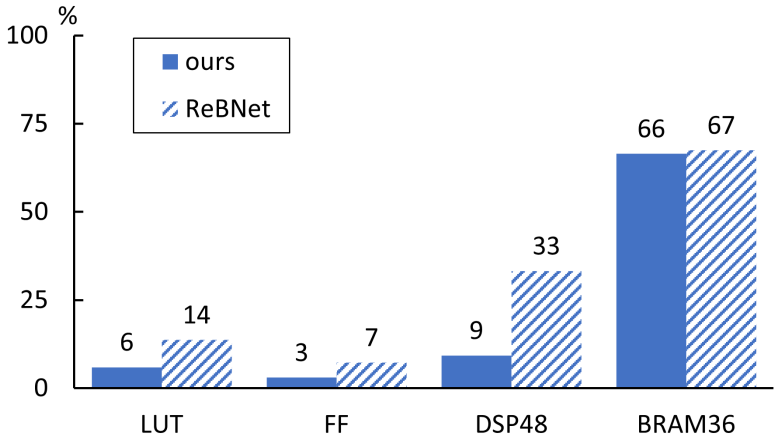


Figure 4.13: Normalized hardware utilization of Arch3

Table 4.10: Frequency, throughput and power usage of Arch3

	This Work	ReBNet
LUT	31938	73403
FF	31975	77785
DSP48	71	255
BRAM36	1148	1165
power(W)	2.872	4.144
freq(MHz)	162.1	155.84
thr.put(FPS)	427.19	205.34

4.5 Measurement on Development Kit

In this section, we run Arch1 and Arch2 on ZedBoard(Figure 4.14). ZedBoard is a development kit based on Xilinx xc7z020clg484-1 with 512MB 32bit DDR3 1066MHz SDRAM.

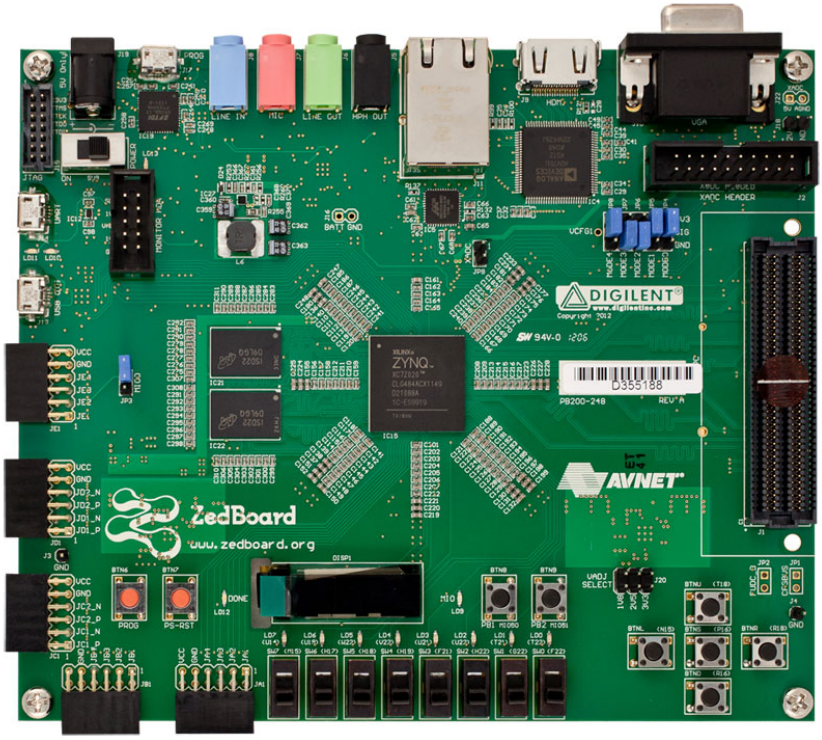


Figure 4.14: ZedBoard

To run the hardware accelerator, we need to add ZYNQ Processing System, AXI Interconnect and Processor System Reset to our design so that we can control it via ARM Cortex-A9 CPU in ZYNQ Processing System. There are two types of buses that connect ZYNQ Processing System with hardware accelerator, AXI 4 and AXI 4 Lite. AXI 4 Lite is for initializing weight memory, setting base address, sending start command and getting current state, and ARM CPU is the master device. AXI 4 is for feeding the input images and receiving the classification results, and the hardware accelerator is the master device, so the hardware accelerator can access the DDR memory controller directly via AMBA Switches. We use the degree of parallelism describe in Section 4.4.1, and to make the timing constraints pass easily, we used a safe frequency of 100MHz during implementation.

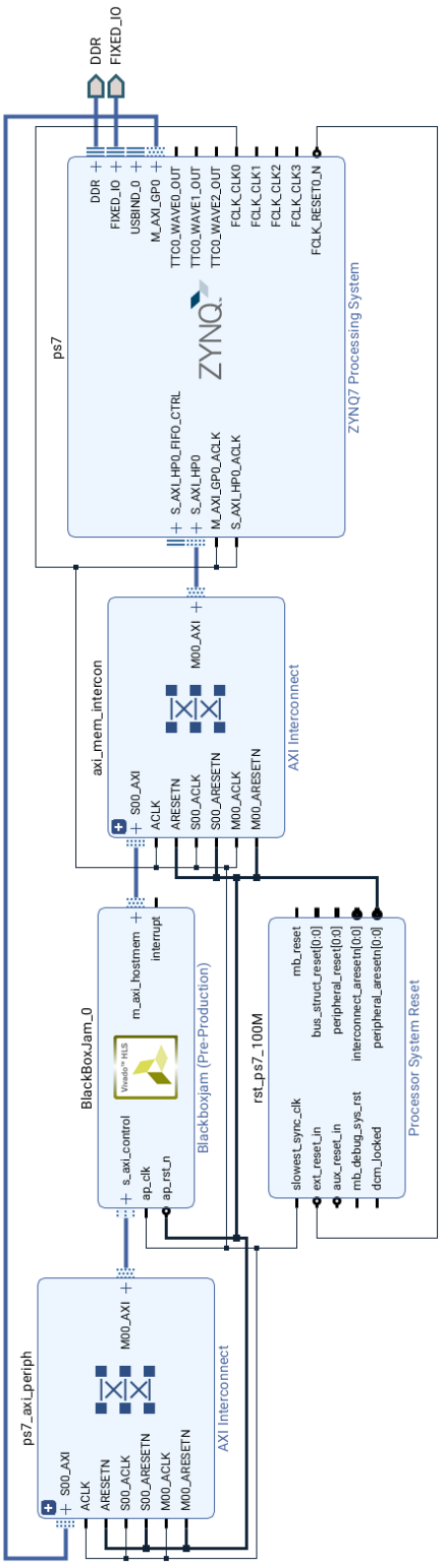


Figure 4.15: Processing System

We wrote a simple control program running on ARM CPU to make hardware accelerator run, get the running time, and check the result if it is the same as Binary Evaluator. Since CIFAR-10 and SVNH share the neural network architecture, we only packed the test datasets of MNIST and CIFAR-10, and the ground true, and Binary Evaluator results of them. MNIST and CIFAR-10 each have 10,000 images in the test dataset. The ARM CPU will output log via uart during running. The log is shown in Code 4.1.

Code 4.1: Log from ARM CPU

```
1  //Arch1 MNIST M=2
2  start!
3  Output took 9614.34 us.
4  done!
5  OK: 9803  Failed: 197 Same with SW: 10000
6
7  //Arch2 CIFAR-10 M=2
8  start!
9  Output took 3281570.75 us.
10 done!
11 OK: 8497  Failed: 1503  Same with SW: 10000
12
13 //Arch1 MNIST M=3
14 start!
15 Output took 9614.39 us.
16 done!
17 OK: 9793  Failed: 207 Same with SW: 10000
18
19 //Arch2 CIFAR-10 M=3
20 start!
21 Output took 3281570.69 us.
22 done!
23 OK: 8632  Failed: 1368  Same with SW: 10000
```

All of the scenarios got completely the same result with Binary Evaluator, and the running time, throughput, theoretical achievement rate and power usage are shown in Table 4.11. The power usage was measured by a meter, and it is the power usage of the whole ZedBoard when the hardware accelerator is running, while the idle power usage is 3.68W. To measure the power usage of MNIST which only runs for about 9ms, we duplicated the train dataset of MNIST 40 times, from 60,000 to 2400,000, and it took about 2.3 seconds to infer them.

Table 4.11: Hardware accelerators run on ZedBoard

	M	time(us)	thr.put(FPS)	achi.rate	power(W)
Arch1	2	9614.34	1040113	99.85%	4.48
(MNIST)	3	9614.39	1040108	99.85%	4.80
Arch2	2	3281570.75	3047	99.85%	4.31
(CIFAR-10)	3	3281570.69	3047	99.85%	4.62

We compare this result with NVIDIA Tesla P100 PCIe 16 GB which runs floating-point models with the same architectures in Figure 4.5. GPU runs fast in batch mode, but the latency of batch mode is quite large, which consists of time of inputting batch-size pictures plus the time of computing a batch. In our hardware accelerator, the latency is nearly around the time of inputting one picture multiplied by layer numbers. Table 4.12 shows the throughput and power usage of Tesla P100 on each architecture. We warmed up GPU before measuring the throughput and power usage was reported by the NVIDIA driver with a sampling rate of 1ms. The power usage when idle is about 25W, and we run each scenario twice after warming up, and the result is the average value.

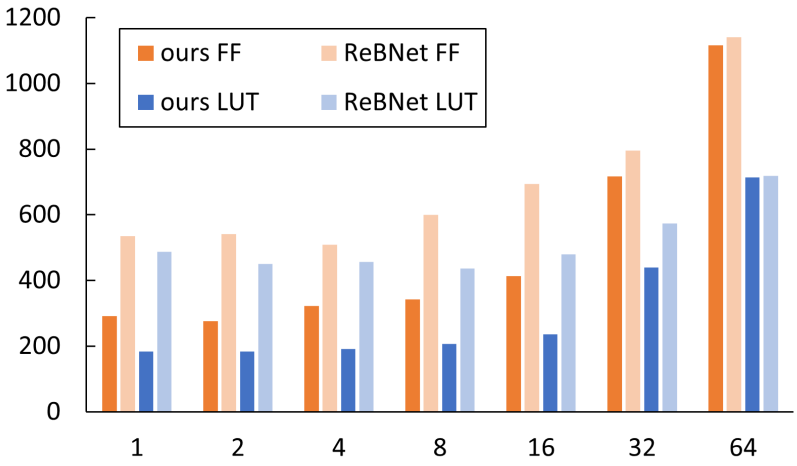
Table 4.12: NVIDIA Tesla P100

	batch	thr.put(FPS)	power(W)
Arch1	100	84151	32.93
(MNIST)	1	1224	32.86
Arch2	100	17361	95.06
(CIFAR-10)	1	664	42.79
Arch3	100	1844	96.69
(ImageNet)	1	307	93.00

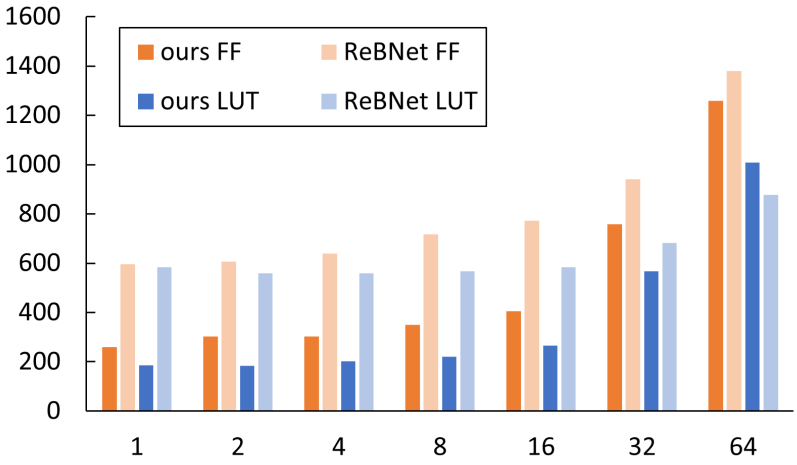
The high-end GPU in unbatch mode which has a low latency was slower than low-cost FPGA, and the power usage is quite higher than FPGA.

4.6 Analysing on usage of MVTU

In Section 4.4.2, we found our design uses more hardware resources in Arch2 under the same *Fold* settings. We discuss it in this section. To find the impact of SIMD width on the growth of utilization, we design a small accelerator that only contains one MVTU, and there is only one PE in that MVTU. The input synapses and output neurons are fixed to 1024, and these parameters only take effects on loop cycles and the size of weight memory, but we only measure the utilization of the MVTU, so it will not significantly change results. Figure 4.16 shows the utilization of that MVTU implemented with different SIMD Width, and the target device in use is 7z020.



(a) $M = 2$



(b) $M = 3$

Figure 4.16: Utilization of one MVTU with one PE

We can see that our designs use much fewer LUTs and Flip-Flops than ReBNet per PE when the SIMD width is lower than 16, but when it is 32, the advantages are reduced. When it is 64, our design uses almost the same level of resource with ReBNet in $M = 2$ and significantly uses more LUTs in $M = 3$. Since there are common parts for PEs such as I/O buffer and loop counter in an MVTU, we implement 16 PEs in one MVTU to check if this conclusion can be generalized, and the usage is shown in Figure 4.17. The proportional relationship did not change from Figure 4.16.

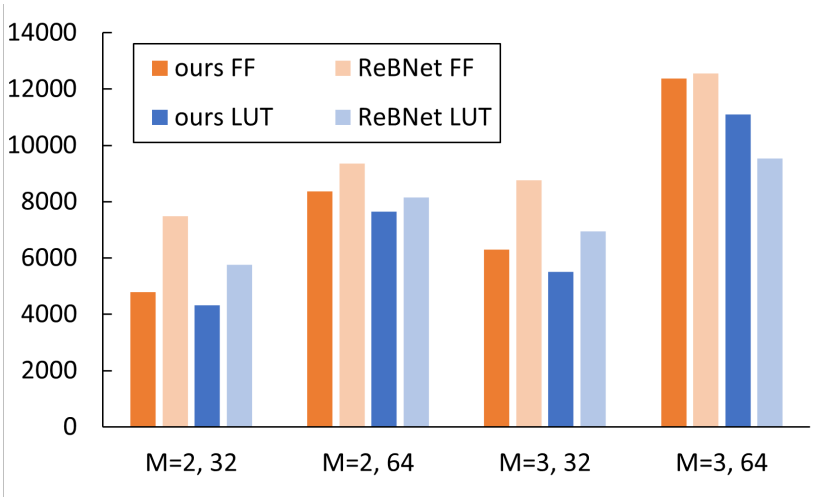


Figure 4.17: Utilization of one MVTU contains 16 PEs

Since more Popcount modules are required in our design, and there are shift-registers in Popcount that only can be implemented by SLICEM, this may cause spending large area in some FPGAs which have less SLICEM.

4.7 Analysing on optimization of SWU

We described our optimization, Data Stream with Adaptive Bit Width, in Section 3.4. This optimization directly affects the utilization of SWU. In this section, we make a comparison of SWU with this optimization, SWU without this optimization, and SWU in ReBNet. First, we implemented *Fold* settings from Table 4.5(Arch2, This Work, M=2) onto 7z100, and the results are shown in Table 4.13.

Table 4.13: Comparison on Data Stream Optimization

	ReBNet	without opt.	with opt.
LUT	34324	33459	29265
FF	49586	50322	41155
DSP	318	87	87
BRAM	387	418	241

The scenario without optimization spends more Block RAMs and Flip-

Flops than ReBNet, and the usage of LUT is close to ReBNet. After optimizing, our design uses much fewer hardware resources overall.

Then, we implemented SWU only to determine the effect of optimization in detail as we did in Section 4.6. The kernel size is set to 3×3, and the width and height of the input feature map are set to 64. These values only affect loop cycles but not significantly change utilization. The number of input feature map channels is 128 and 256. The utilization of SWU module when $M = 2$ is shown in Table 4.14, where SIMD width only affects our optimized design, and DSP48 will not be used in SWU module.

Table 4.14: Utilization of one SWU

(a) Channels = 128

SIMD	ReBNet	without opt.	64	32
LUT	518	519	322	433
FF	590	373	828	969
BRAM	16	16	2	2

(b) Channels = 256

SIMD	ReBNet	without opt.	64	32
LUT	820	787	440	513
FF	589	549	1033	946
BRAM	32	30	4	4

The require buffer capacity of SWU is

$$\begin{aligned} &\text{Channels} \times (\text{KernelSize} + 1) \times \text{FeaturemapWidth} \times M \\ &= 128 \times 4 \times 64 \times 2 = 65536 \end{aligned} \tag{4.1}$$

when the number of channels is 128. The necessary amount of Block RAMs is 2. It will be doubled when the number of channels is 256. From Table 4.14, our optimized designs save up to $\frac{7}{8}$ Block RAMs, and the capacity in use is 32Kbit for each Block RAM. It is very efficient.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

This thesis described our method, isometric residual-binarization, and corresponding hardware design. Thanks to our efficient design, other optimization which makes accelerator M times faster can be applied. Less Initiation Interval requires wider dataflow which causes overhead, and we showed our method to resolve this problem. We also proposed a smaller encoder in PE to save hardware recourse.

Our isometric residual-binarization obtains similar accuracy to the baseline work, ReBNet, in small datasets, and a bit higher accuracy in ImageNet, but the obtained accuracy is still lower than full-precision models. We evaluated the accuracy change after binary conversion by a software "Binary Evaluator".

Our hardware design reaches higher degree of parallelism in resource-limited devices and gets higher through-put than ReBNet. In large device 7z100, our design gets much lower hardware resource usage in Arch1, but similar level of hardware resource usage in Arch2. In both Arch1 and Arch2, our designs only use about $\frac{1}{M+2}$ DSP48s. This makes usages between resource types are well balanced in our method. We also implemented Arch3 which is for large dataset ImageNet on vu095.

Then we measured Arch1 and Arch2 on development kit, Zedboard. The classification results were completely the same as the output of software "Binary Evaluator" and the throughput on the development kit was close to theoretical value where the achievement rate was up to 99.85%. Then we compared it with floating-point device.

In addition, we analysed the impact of SIMD width on the growth of hardware resource usage. When SIMD width is less than or equal to 32, our design uses less hardware resource per PE than baseline work.

Finally, we analysed the effect of our optimizations which reduce the hardware usage, especially on Block RAMs. our optimization makes SWU in our design save up to $\frac{7}{8}$ Block RAMs.

5.2 Future Work

In the future, we will try to improve the accuracy on ImageNet with more efficient backbone, and implement it on FPGAs. In addition, we will try to use our method in difficult tasks such as object detection and semantic segmentation.

Bibliography

[1] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *CoRR*, vol. abs/1704.04861, 2017. [Online]. Available: http://arxiv.org/abs/1704.04861

[2] X. Zhang, X. Zhou, M. Lin, and J. Sun, "Shufflenet: An extremely efficient convolutional neural network for mobile devices," *CoRR*, vol. abs/1707.01083, 2017. [Online]. Available: http://arxiv.org/abs/1707.01083

[3] M. Ghasemzadeh, M. Samragh, and F. Koushanfar, "Rebnet: Residual binarized neural network," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2018, pp. 57–64.

[4] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, no. 1, p. 1929–1958, Jan. 2014.

[5] Y. LeCun, C. Cortes, and C. Burges, "Mnist handwritten digit database," *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, vol. 2, 2010.

[6] A. Krizhevsky, "Learning multiple layers of features from tiny images," Tech. Rep., 2009.

[7] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng, "Reading digits in natural images with unsupervised feature learning," 2011.

[8] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and

L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[9] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015. [Online]. Available: http://arxiv.org/abs/1512.03385

[10] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," *CoRR*, vol. abs/1409.4842, 2014. [Online]. Available: http://arxiv.org/abs/1409.4842

[11] M. Courbariaux and Y. Bengio, "Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1," *CoRR*, vol. abs/1602.02830, 2016. [Online]. Available: http://arxiv.org/abs/1602.02830

[12] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," *CoRR*, vol. abs/1603.05279, 2016. [Online]. Available: http://arxiv.org/abs/1603.05279

[13] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. G. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," *CoRR*, vol. abs/1712.05877, 2017. [Online]. Available: http://arxiv.org/abs/1712.05877

[14] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *CoRR*, vol. abs/1609.07061, 2016. [Online]. Available: http://arxiv.org/abs/1609.07061

[15] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," *CoRR*, vol. abs/1506.02626, 2015. [Online]. Available: http://arxiv.org/abs/1506.02626

[16] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets," *CoRR*, vol. abs/1608.08710, 2016. [Online]. Available: http://arxiv.org/abs/1608.08710

[17] Baoyuan Liu, Min Wang, H. Foroosh, M. Tappen, and M. Penksy, "Sparse convolutional neural networks," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015, pp. 806–814.

[18] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," in *Advances in Neural Information Processing Systems*, D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, Eds., vol. 29. Curran Associates, Inc., 2016, pp. 2074–2082. [Online]. Available: https://proceedings.neurips.cc/paper/2016/file/41bfd20a38bb1b0bec75acf0845530a7-Paper.pdf

[19] P. Micikevicius, S. Narang, J. Alben, G. F. Diamos, E. Elsen, D. García, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu, "Mixed precision training," *CoRR*, vol. abs/1710.03740, 2017. [Online]. Available: http://arxiv.org/abs/1710.03740

[20] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. 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*, ser. FPGA '17. ACM, 2017, pp. 65–74.

[21] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *CoRR*, vol. abs/1502.03167, 2015. [Online]. Available: http://arxiv.org/abs/1502.03167

[22] D. Hubel and T. Wiesel, "Receptive fields, binocular interaction, and functional architecture in the cat's visual cortex," *Journal of Physiology*, vol. 160, pp. 106–154, 1962.

[23] K. Fukushima, "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position," *Biological Cybernetics*, vol. 36, pp. 193–202, 1980.

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

[25] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'12. Red Hook, NY, USA: Curran Associates Inc., 2012, p. 1097–1105.

[26] J. Weng, N. Ahuja, and T. S. Huang, "Cresceptron: a self-organizing neural network which grows adaptively," in *International Joint Conference on Neural Networks (IJCNN)*, vol. 1. IEEE, 1992, pp. 576–581.

[27] F. Chollet *et al.*, "Keras," https://keras.io, 2015.

[28] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[29] Xilinx, "Vivado design suite," https://www.xilinx.com/products/design-tools/vivado.html.

[30] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014. [Online]. Available: http://arxiv.org/abs/1412.6980