TWINDNN: A TALE OF TWO DEEP NEURAL NETWORKS

BY

HYUNMIN JEONG

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois Urbana-Champaign, 2021

Urbana, Illinois

Adviser:

Professor Deming Chen

ABSTRACT

Compression technologies for deep neural networks (DNNs), such as weight quantization, have been widely investigated to reduce the model size so that they can be implemented on hardware with strict resource restrictions. However, one major disadvantage of model compression is accuracy degradation. To deal with this problem effectively, we propose a new compressed network inference scheme with a high accuracy but slower DNN coupled with its highly compressed DNN version that typically delivers much faster inference speed but with a lower accuracy. During the inference, we determine the confidence of the prediction of the compressed DNN, and infer the original neural network for the inputs that are considered not confident by the compressed DNN. The proposed design uses a balanced number of resources available on the hardware and can deliver overall accuracy close to the high accuracy model, but with the inference speed closer to the compressed DNN. We demonstrate our design on two image classification tasks: CIFAR-10 and ImageNet. Our experiments show that our design can recover up to 94% of accuracy drop caused by extreme network compression, with more than 90% increase in throughput compared to just using the original DNN. This is more than 17% extra accuracy recovery and 36% extra speedup compared to the previous work with a similar concept on VGG-16. This is the first work that considers using a highly compressed DNN along with the original DNN in parallel to achieve high accuracy and speed at the same time, while maintaining the resource balance by using two different main computation sources on the field programmable gate array (FPGA).

ACKNOWLEDGMENTS

I have received lots of support and assistance throughout my research career and writing this thesis.

I would first like to express my deepest appreciation to my adviser, Professor Deming Chen, who gave me valuable advice and suggestions throughout my entire research career. Your insightful feedback brought my works and research vision to the next level.

I would like to acknowledge my colleagues in the Circuits Laboratory for their collaboration. You provided constructive criticisms and practical suggestions that certainly enhanced my work.

I would also like to thank ECE Editorial Services for proofreading this thesis. Your revisions and suggestions surely made this thesis more refined and understandable for general readers.

In addition, I would like to thank my parents for their continuous support and wise counsel. You have always been on my side regardless of how well I do. I would also like to thank my friends for providing entertaining and meaningful relaxation after hard work. Your distractions were indeed necessary for making my college life healthier.

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION 1

CHAPTER 2 BACKGROUND AND RELATED WORK 4
2.1 Extremely Low Bit-Width Neural Networks 4
2.2 Multiple Neural Networks Architecture 5

CHAPTER 3 DESIGN FLOW AND IMPLEMENTATION 7
3.1 Model Generation 7
3.2 Accelerator Development 8
3.3 Confidence Score for Decision Making 11
3.4 Software Development 13

CHAPTER 4 EXPERIMENT 17
4.1 CIFAR-10 19
4.2 ImageNet 19

CHAPTER 5 CONCLUSION 22

REFERENCES 24

CHAPTER 1

INTRODUCTION

Machine learning is one of the most popular fields in the current era. It is used in various areas, such as speech recognition, face recognition, medical diagnosis, etc. However, the serious problem is that the neural networks for machine learning applications [1, 2] are becoming too large and slow as they get more complicated and powerful. This problem is further exacerbated when neural networks are used for edge devices with a small chip for real-time systems. As a result, researchers have proposed two major solutions to tackle this problem.

The first is to use specialized hardware for neural network inference. One popular device type is the graphics processing unit (GPU), which is widely adopted for accelerating neural network computations. In this study, on the other hand, we will focus on using field programmable gate array (FPGA) devices as specialized hardware. There are many benefits of using FPGAs for neural network computations, but the most important aspect is that it can provide a more specialized and customized hardware that is designed solely for a specific application. Developers can even design a customized hardware for the application that is difficult to optimize with GPUs. This is possible due to the fundamental design of FPGAs, where developers can allocate any resources to design any circuits they want as long as they honor the total resource limit available on the FPGA, and this feature of FPGAs often provides an efficient way to implement or evaluate new ideas and designs.

The second is to reduce the size of neural networks so that their inference latencies are low enough to handle real-time inputs [3, 4, 5, 6, 7, 8]. There are numerous methods to reduce the size of neural networks for different platforms, among which are CPUs, GPUs, and FPGAs. Of these methods, only FPGAs offer the particular benefit of full customization compared to the other two, so they have been studied extensively with a variety of methods to optimize neural networks. Quantization of networks is the most popular

and effective method to reduce the size and inference latency at the same time [9] for FPGAs, as developers can also minimize the data size to reduce the usage of both memory and computation resources required. In particular, extremely low bit-width networks on FPGAs, such as binary or ternary neural networks, have been studied recently [10, 11, 12, 13, 14, 15, 16, 17]. These networks require significantly fewer resources compared to the regular quantized networks. However, this benefit is not free, of course. One major disadvantage of these low bit-width networks is that they tend to have even more accuracy drop than regular quantized neural networks, as a result of further reduced precision. Therefore, it is more difficult to use binary or ternary neural networks as they are, especially in fields such as surveillance or medical diagnosis systems, where the cost of that accuracy drop is much larger than the inference speed improvement.

This study aims to accelerate neural network inference by using an extremely low bit-width network implementation on FPGAs, while maintaining the accuracy of the original network by using a relatively high precision network concurrently, without having to develop a single neural network accelerator that meets both accuracy and inference speed requirements. This design can also solve the resource bottleneck problem that arises when developing a neural network accelerator on FPGAs. Ideal implementation of this concept can maximize the resource utilization of all computation resources on FPGA and increase the throughput beyond the number of multipliers available.

In summary, we propose a system that consists of two distinct networks: one extremely low bit-width network that is focused on speed, and another moderately quantized network that is focused on accuracy. In this thesis, the extremely low bit-width network will be called a compressed network, and the moderately quantized network will be called an original network. These two networks work in a way that can exploit advantages in both accuracy and speed at the same time. Our main contributions are as follows:

- We design TwinDNN accelerators that are designed and optimized to exploit both low and high bit-width networks, with pipelined and parallelized computation engines that balance the utilization of both digital signal processing blocks (DSPs) and look-up tables (LUTs).

- We build a software solution that allows the two accelerators to be run

in a hierarchical fashion with a real-time parallel inference scheme that maximizes the throughput of the design.

- For ImageNet and ResNet-18, our TwinDNN solution can deliver up to 1.9× speedup with only 3% extra DSPs compared to the solution when only a single original neural network is used, and up to 95% of the accuracy loss is recovered during hierarchical inference compared to the solution when only a single compressed network is used.

In Chapter 2, some background information and previous studies related to this work will be introduced. In Chapter 3, the design flow of our implementation and experiment will be explained. In Chapter 4, the results of our experiments will be described. Chapter 5 will conclude the thesis with discussion of future explorations.

CHAPTER 2

BACKGROUND AND RELATED WORK

2.1 Extremely Low Bit-Width Neural Networks

Recent researches have succeeded in binarizing or ternarizing parts of layers in neural networks [10, 11, 12, 13, 14]. Many experiments claim that these compression methods are very effective in terms of latency reduction with some accuracy drops. As one would expect, as the number of bits used to represent either weights or feature maps decreases, the accuracy drops more significantly. Because the goal of our study is to compensate for the accuracy loss caused by compression, we can forgive moderate accuracy loss, as long as the benefit of using those networks is significant. First, we define the quantized weights with extremely low precision as follows:

$$
\begin{aligned}
w_b &= \begin{cases} -w_{scale} & \text{if } b = 0 \\ +w_{scale} & \text{if } b = 1 \end{cases} \\
w_t &= \begin{cases} -w_{scale} & \text{if } t = -1 \\ +w_{scale} & \text{if } t = 1 \\ 0 & \text{if } t = 0 \end{cases}
\end{aligned}
\tag{2.1}
$$

Equation 2.1 shows how these extremely low bit-width weights are used in computation. The term b is a 1-bit value that can be either 0 or 1, and t is a 2-bit value that can take either -1, 0, or 1. The key idea here is that w_{scale} value is the same across the weights. The bits are only used in sign representations. In binary, as an example, a single bit of 0 represents negative and 1 represents positive, and this logic can be implemented in a simple condition, or a multiplexer in FPGAs. The w_{scale} value is stored separately, and the same w_{scale} value is multiplied over all binary weights to get the actual weight values. However, we do not need to perform all of these multiplications separately. Considering $b_1 = 0$ and $b_2 = 1$ for the binary case,

where a stands for activation, or feature map, then we can express a very simple neural network computation as follows:

$$\begin{aligned} a_{next} &= w_{b_1} \times a_1 + w_{b_2} \times a_2 \\ &= -w_{scale} \times a_1 + w_{scale} \times a_2 \\ &= w_{scale} \times (-a_1 + a_2) \end{aligned} \tag{2.2}$$

This shows how binary and ternary weight computations can be handled with a single multiplication. Reducing the number of actual multiplications reduces the need of DSPs, and indeed makes the overall computation faster. For ternary, the only difference is that two bits now represent positive, negative, and zero. Therefore, the main benefit of using extremely low bit-width neural networks is more effective and balanced resource utilization, specifically on FPGAs. For a typical DNN implementation on FPGAs, DSP is the one that directly determines the performance, and so is the limiting factor of the performance. Therefore, typical DNN implementations on FPGAs utilize nearly all DSPs available, and other resources, including LUTs, are left underutilized. Extremely low bit-width network, on the other hand, only uses a minimal number of DSPs and mainly utilizes LUTs as a main computation source instead of DSPs. In this study, we instantiate both the original DNN and the extremely low bit-width DNN (compressed DNN) at the same time, in a way that the original DNN uses most of the DSPs available on the board, and the compressed DNN uses extra LUTs that were not used by the original DNN. This method allows us to utilize both DSP and LUT resources as much as possible to ultimately speed up the overall inference.

2.2 Multiple Neural Networks Architecture

There have already been researches on this concept of hierarchical neural network design, where compressed and original networks are both used in neural network inference [18, 19, 20]. They have succeeded in achieving balanced accuracy and latency results by using different techniques, such as low-power MCU [18], or FPGA [19, 20], to realize the concept. Although this work also mainly uses the concept of hierarchical neural network design, there are several major differences from previous studies.

Previous studies have focused only on the accuracy and inference speed of

the design, and they are indeed the most important factors when evaluating neural network accelerators. However, these numbers cannot really represent the best performance if the design cannot utilize all the resources that are available. For FPGAs specifically, most accelerator designs are focused on utilizing DSPs, which are the main computation units on FPGAs, but not so much on LUTs, which can also be used as computation units in special cases, such as binary and ternary networks. However, in this study, both DSP and LUT utilizations are maximized in a flexible and efficient way by implementing two different networks with different main computation units. This is an aspect that CPU-based solutions [18] or even previous FPGA-based solutions [19, 20] did not offer. Previous FPGA-based solutions either implemented only one of the networks on FPGAs [20], which resulted in LUT being a bottleneck, or did not use extremely low bit-width networks [19], which resulted in DSP being a bottleneck. Our work represents a novel direction in hardware accelerator design, which can potentially achieve the maximum throughput beyond the number of DSPs and break the traditionally thought limitation of FPGA accelerators.

Furthermore, our hardware and software design provides a true real-time parallel inference scheme, which allows its users to exploit all resources for the entire time. Such customization of accelerators and their concurrent execution are big advantages of FPGAs, and this is fundamentally different from the sequential approaches that previous studies [18, 19, 20] took, which could make one network idle when the other network is running.

Finally, our work also applies more advanced training methods [14, 13, 11, 12], which allow the network to be compressed down to binary and ternary networks with a reasonable accuracy. These methods significantly improve the overall accuracy and speed of the design.

CHAPTER 3

DESIGN FLOW AND IMPLEMENTATION

Our implementation flow consists of three parts: creating the original network and compressed network models, implementing high-level-synthesis (HLS) accelerator intellectual properties (IPs) for those networks, and creating a software system for TwinDNN inference.

3.1 Model Generation

Creating the original network starts with a typical floating-point training, which can also be completed by using a pretrained model available. For training, we used a Caffe framework [21], which was also customized to be used by other works (e.g., [14]). To enhance the accuracy, we use a variety of well-known techniques, such as learning rate decay and batch normalization. After the floating point model finishes training, network weights are quantized to designated bit-widths, which are 16-bit and 8-bit in our experiments. These moderately quantized networks are called original networks in our study, and they typically maintain the accuracy of the floating point network. Quantization scheme is determined by the accuracy drop and distribution of weights. First, we try a uniform quantization scheme, where we apply the same integer and decimal bit widths for all layer weights. We always use uniform quantization whenever possible because nonuniform quantization requires extra logic and computation required for bit shifting in hardware. If the accuracy drop is significant, we then try a nonuniform quantization scheme depending on the distribution of weights and activations. There can still exist a slight accuracy drop after non-uniform quantization, and there are a few ways presented in [22, 23] to recover this accuracy drop, which can be implemented in the future.

Compressed network model, on the other hand, cannot be generated with-

out a training scheme that is specifically designed for binary and ternary neural networks. For the binary neural network, which was used in our CIFAR-10 experiment, we used the same model in [13], which was trained using the method proposed by [11]. This model showed approximately 5% accuracy drop compared to the floating point network model.

For the ternary neural network, which was used in our ImageNet experiments, we trained the model by using the framework explained in [14]. Our trained model, however, could not reach the exact accuracy reported in [14], and this is due to the additional fine-tuning and data augmentations that they performed. Our trained model also showed approximately 5%-8% accuracy drop compared to the floating point network model, which seems valid for the purpose of this work.

3.2 Accelerator Development

Xilinx's Vivado high-level-synthesis tool was used to generate IPs for both original and compressed networks. Their tools allow developers to apply various optimizations, such as loop pipelining and array partitioning, more easily on their FPGAs. We targeted Ultra96 and ZCU102 FPGAs, which are both Arm-based Xilinx Zynq UltraScale+ MPSoC development boards. ZCU102 has more overall resources than Ultra96 and is used for MobileNetV2 experiments only.

Our design process was as follows. First, we design an accelerator for the original network without considering the compressed network, except for the few DSPs that the compressed network may use. This is the same as the development process of a normal neural network accelerator. Here, we can even use neural network IPs that are already optimized for the specific FPGA in use, as long as it has some leftover LUTs, which is typically the case because accelerator designs on an FPGA are mostly limited by the number of DSPs. Then, we design an accelerator for the compressed network with leftover LUTs. In our experiments, more than half of LUTs remained unused by the original network, which left a significant amount of resources available for the compressed network design. Thus, we were able to design a very reasonably optimized compressed network accelerator with these leftover resources, which does not require additional balancing for these two accelerators.

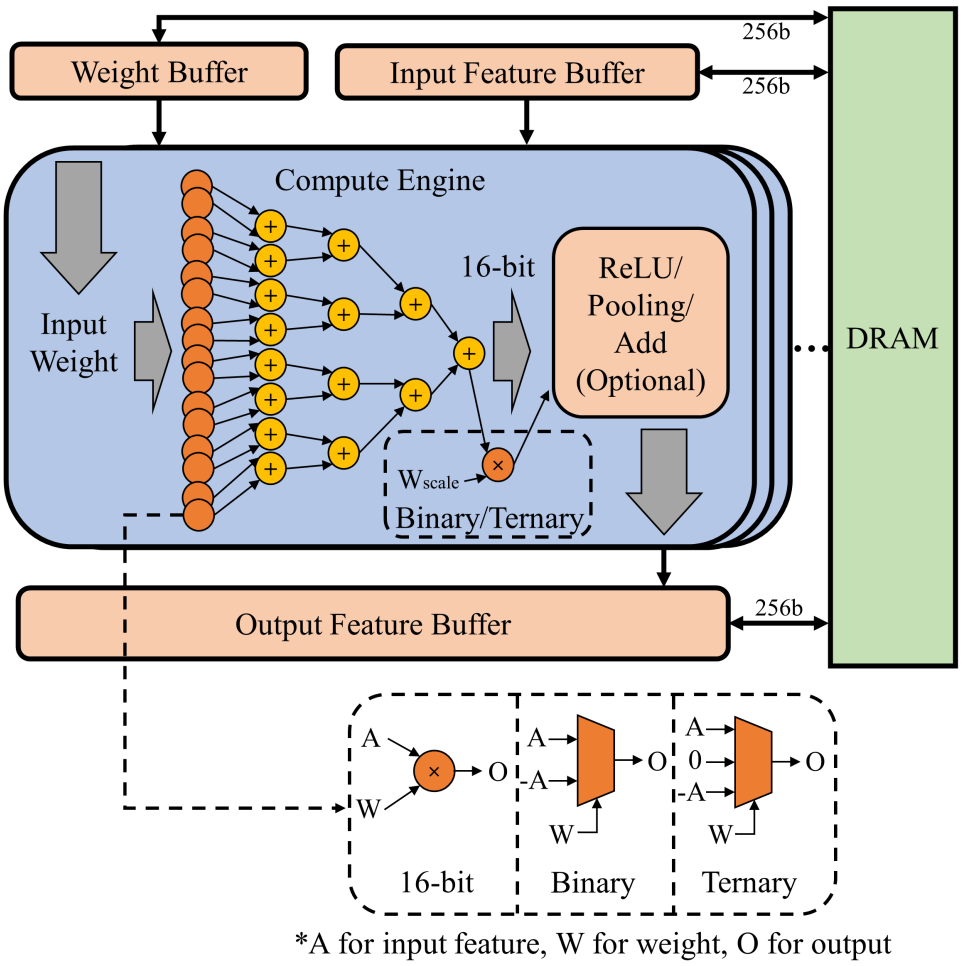

Figure 3.1: Basic accelerator architecture

Two accelerators are designed to take Caffe [21] network model definition as an input. This is to ensure that our accelerators can be used with any network configurations. Specifically, for each layer, accelerators will be aware of whether the layer is convolutional or fully connected, the convolutional layer parameters such as kernel size and stride, and whether to perform additional computations such as pooling. Therefore, the original and compressed network accelerators can work individually with different network definitions. This feature of our accelerators provides an extra flexibility in network configurations, so allows users to utilize different networks without additional design overhead. Currently, our accelerators are only generalized for neural networks with ImageNet-based image classification tasks due to resource constraints, but we plan to further generalize the input and output layer as well so that it can work with any type of neural networks.

Figure 3.1 shows the overall architecture of accelerators. For convolutional and fully connected layer computations, the main technique we used was to have multiple pipelined computation engines that compute partial multiply-accumulate (MAC) operations. It will perform element-wise multiplication of weights and input features, and then compute the sum of the products using an adder tree. These computation engines are pipelined so that they can produce a MAC of 16 weights and 16 features every single cycle. For a 16-bit network, every orange node uses a single DSP each, which takes each input feature (A) and weight (W) as operands and computes their product. As a further optimization for DSPs, for an 8-bit network, because each DSP block on Ultra96 FPGA supports up to 25×18 bit multiplication, we were able to allocate two 8×8 multiplications on one DSP, with a method proposed by [24]. Finally, for binary and ternary networks, we modified our computation engines to utilize multiplexers (MUX) for multiplication computations and a single DSP for the final scaling. For binary networks, w_b is used as a 1-bit selector to determine the output between $-A$ and $+A$. For ternary networks, w_t is used as a 2-bit selector to determine the output between $-A$, 0, and $+A$. Then, the sum of those outputs will be computed using the adder tree, same as before. At the end of all computations, we will multiply w_{scale} values from Equation 2.1.

Typical neural network models also include additional layers such as rectified linear unit (ReLU), pooling, and addition layers. These layers are appended to the computation engine outputs and receive selector bits to de-

termine whether these additional computations are needed. As these layers only take a small portion of overall inference time compared to convolutional and fully connected layers, their computations are not parallelized. Instead, they are pipelined so that we can maximize the throughput of these computations while allocating more resources to computation-heavy layers.

Another optimization method we used is to utilize on-chip memory to store partial weights, features, and intermediate results. For 16-bit networks, for example, we have a 16×16 weight buffer, which fetches only the weights needed for current computation. Convolution computation was redesigned so that every weight needs to be fetched only once. Partial input and output features are also stored in on-chip block memory. Both input and output features are divided into blocks with 16 channels, which are stored in on-chip block memory at a time. Utilization of on-chip memory allows the majority of global memory accesses to become a local memory access instead. Without this optimization, global memory access is likely to become a bottleneck of accelerator performance, as it is many times slower than on-chip block memory and flip-flops.

The last optimization to discuss is to maximize the utilization of bus width. Although most weights and features that are loaded and stored to DRAM have a bit-width of less than 16, it does not mean that the data need to be transferred at that bit-width. Most FPGAs have much larger DRAM bus bit-width than 16-bit. In order to utilize the memory bus as much as possible, we reorder parameters and features into 256-bit blocks that can be loaded or stored on DRAM as a single element. Furthermore, the parameters and features are organized in a way that would allow the accelerator to invoke burst contiguous memory reads to maximize the bandwidth. It is possible to have bigger blocks, such as 512-bit blocks, but after this point the memory interfaces use a lot more LUTs, which are supposed to be used to increase the speed of the low bit-width network.

3.3 Confidence Score for Decision Making

In neural network image classifications, output of the final layer is a list of values for each class, and the class with the highest final layer output is typically chosen as a prediction. Here, each value represents how possible is

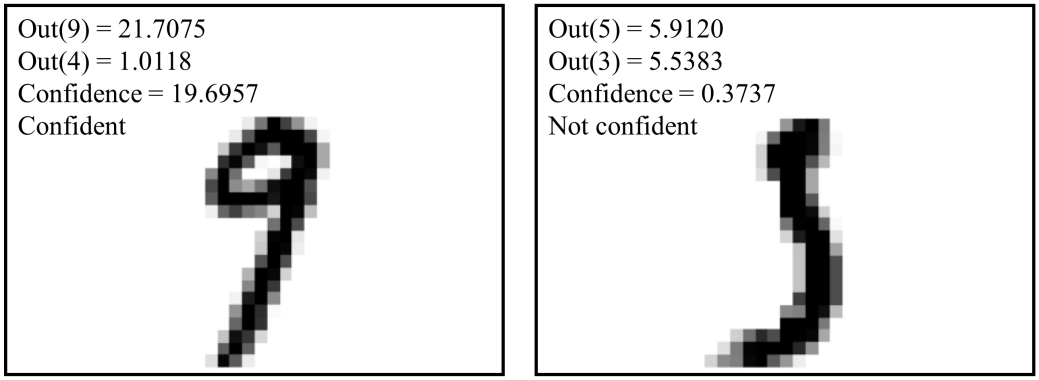

Figure 3.2: Handwritten digit recognition examples for confidence score

that image in the class, and based on these values, we will define what we call a confidence score of an inference. Confidence is defined as the difference between the two largest output values of the neural network and is used to determine if the prediction of the compressed network is reliable enough to be used as an actual output without verification from the original network. Simply speaking, the confidence score of a compressed network output will be used to determine whether to infer the original network.

Here is an explanation of the logic behind utilizing the confidence score during inference. We present Figure 3.2 as an example. Let us define $Out(x)$ as final output value for label x, and the top two output values and confidence score are shown. For the left image, where $Out(9) = 21.7075$ and $Out(4) = 1.0118$, we consider it as confident, because the network is almost sure that the digit is 9. However, for the right image, where $Out(5) = 5.9120$ and $Out(3) = 5.5383$, we consider it not confident, because even though 5 has the highest possibility, 3 seems to have a reasonably high possibility as well. Especially when we are using an extremely low bit-width network, such small difference could have resulted from the noise of computing in low precision.

Therefore, instead of just finding a label with maximum possibility, our TwinDNN system will now find two labels with the first and second maximum possibilities and compute the difference between those two possibilities. If the difference is large (i.e., beyond a threshold determined empirically), the compressed network prediction is considered confident and will be used as a final output. If the difference is small, however, the compressed network prediction is considered not confident, and in this case, the image will need additional verification from the original network that is designed to have

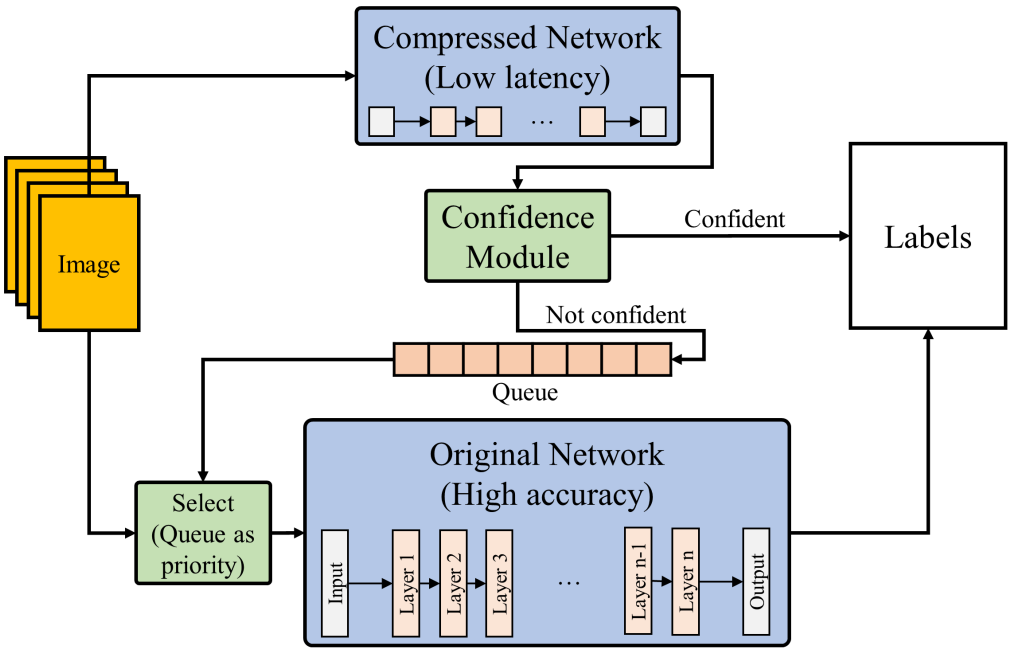

Figure 3.3: Graphical representation of hierarchical architecture

maximum accuracy.

3.4 Software Development

The accelerators are invoked from the software running inside the processing system of the FPGA. Because the two accelerators are both instantiated in a single design, they support concurrent execution. Figure 3.3 shows a graphical representation and Algorithm 1 describes a behavioral pseudocode of TwinDNN's software system, which is designed to fully utilize both networks. First, an image from the source will be processed by whichever network becomes available first. Note that for the image to be processed by the original network directly, the queue in Figure 3.3 should be empty, as it takes priority. If the original network was used for the initial inference, its prediction, or the index with the maximum output value (ArgMax in Algorithm 1), will always be used as the final prediction, because the original network has a higher accuracy. If the compressed network was used for the initial inference, the software will compute two maximums and the index of the first maximum (Max1, Max2, and ArgMax1 in Algorithm 1, respectively) of the compressed

Algorithm 1: Behavioral pseudocode of TwinDNN's software system

```
Input  : Images[], Threshold
Output: Labels[]
Configure CompressedNetwork, OriginalNetwork, and Queue
Thread Compressed Network
    while Images is not empty do
        CurrentImage ← Images.next()
        Output ← CompressedNetwork(CurrentImage)
        Max1, Max2, ArgMax1 ← GetTwoMax(Output)
        Confidence ← Max1 − Max2
        if Confidence > Threshold then
            Labels.add(ArgMax1)
        else
            Queue.push(CurrentImage)
        end
    end
Thread Original Network
    while Images is not empty do
        if Queue is not empty then
            CurrentImage ← Queue.pop()
        else
            CurrentImage ← Images.next()
        end
        Output ← OriginalNetwork(CurrentImage)
        Max, ArgMax ← GetMax(Output)
        Labels.add(ArgMax)
    end
```

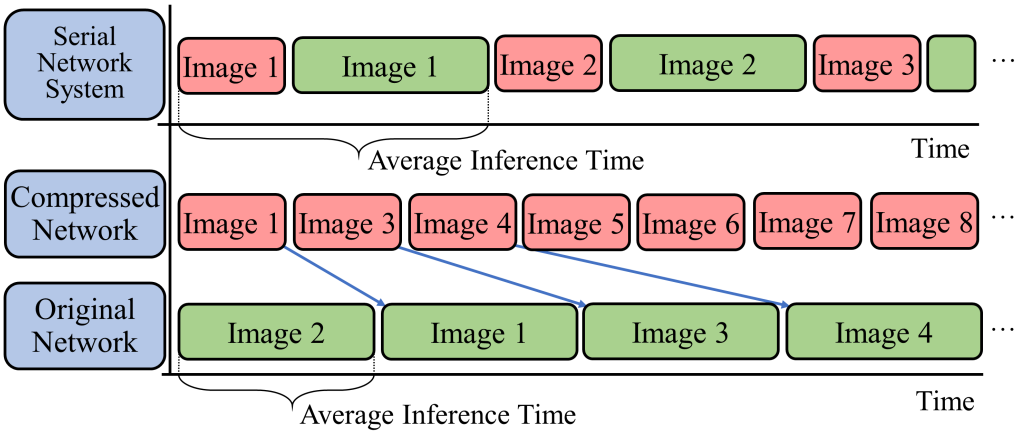

Figure 3.4: Worst case inference diagrams for serial and parallel inference schemes

network output. Then, the confidence score is calculated by subtracting those two maximums to determine whether the image needs additional inference on the original network. If confidence is above the threshold, its prediction, or the index of the first maximum, will be used as the final prediction, but if the confidence is below the threshold, the input image will go into the queue for the original network inference, which will be processed later by the original network. This way, we can ensure that both accelerators are running for the entire time until all images are processed.

This dynamic parallel inference scheme provides one of the main differences between this work and [18, 19], in terms of the worst case inference time. Figure 3.4 is presented to support this claim. In this diagram, red images represent compressed network inferences, which are all considered not confident in the worst case, green images represent original network inference, and blue arrows represent the images waiting in the queue. For a serial network inference system, the worst case total inference time for an image set is the sum of the time taken by the compressed network and the time taken by the original network, which means in the worst case it will perform slower than using the original network alone. However, our FPGA parallel inference scheme drives both original and compressed network accelerators simultaneously. Note that the original network starts with Image 2. This is because right after the compressed network starts processing Image 1, Image 2 will look for an idle accelerator, which will always be the original network as Image 1 has not completed processing yet so the queue is empty. This allows both networks to run in parallel for the entire time until the input source is depleted, so the original network will process all the images without delay because there are more and more images coming from the compressed network constantly. This ensures that the worst case total inference time for an image set is just the time taken by the original network alone, ignoring the queue managing time, which is negligible compared to the time taken for neural networks. Therefore, our parallel inference scheme ensures that the system will not perform slower than using the original network alone, even in the worst case.

Threshold value is determined from experiment. Threshold value of 0 means all compressed network predictions will be considered confident, and none of the inputs will go into the queue. This results in both networks running in parallel independently, as the original network will also get the

input from source. Higher threshold value means that more images go into the original network compared to the low threshold value case, thus it results in higher accuracy but longer inference time. Note that when the threshold value goes above a certain point, the queue will contain some images even after all images from the source are depleted. From that moment, only the original network will be running, and this reduced parallelism impacts the inference speed significantly. Therefore, it is recommended to choose a threshold value that will keep the queue small. Threshold value of infinity, in fact, is the same as just running the original network alone, because all compressed network outputs will be considered not confident and require original network inference. We test a variety of threshold values to see which one gives the most balanced result between accuracy and speed, and will use it to obtain the final result.

CHAPTER 4

EXPERIMENT

We tested our design on two different datasets—CIFAR-10 and ImageNet—and three different networks—ConvNet, ResNet-18, and MobileNetV2. ConvNet and ResNet-18 were used for CIFAR-10 dataset, and ResNet-18 and MobileNetV2 were used for ImageNet dataset. Multiple datasets and networks are used to evaluate the proposed solutions more thoroughly with different image formats and network architectures. We also tried different combinations of different bit-widths to evaluate the generalization of the method, to prove we can use flexible bit-width combinations that can match the designated amount of resources available.

Throughout our experiment, we will define the baseline network as the moderately compressed network (i.e. original network as used throughout this thesis), instead of the floating point network. This also means when we are comparing to the baseline, we are comparing to the configuration where the original network is used alone. There are three reasons behind this choice. First, moderately quantized networks typically maintain the accuracy of floating point networks, with less than 0.1% accuracy difference. Second, we want to show the accuracy recovery of our TwinDNN structure itself, independent of the base network we use. This means even if our moderately quantized network provides lower accuracy than the floating point network, resulting in relatively lower final accuracy, we do not want to conclude that our TwinDNN structure is ineffective, as long as the final accuracy is close to the moderately quantized network accuracy. Finally, if we use floating point network as a baseline, speedup would be too high and impractical because floating point networks are too slow and are rarely implemented on FPGAs.

Table 4.1: Experimental results and comparison with previous work

	[20]	**TwinDNN**	CascadeCNN[19]		**TwinDNN**		
Dataset	CIFAR-10	CIFAR-10	ImageNet		ImageNet		
Platform	ZC706	Ultra96	ZC706		Ultra96		ZCU102
Frequency (MHz)	N/A	100	150		150		200
Number of DSPs*	N/A	256/4	900/900	900/900	274/8	274/8	536/64
Total number of DSPs	N/A	260	1800	1800	282	282	600
Number of LUTs*	N/A	22110/25074	N/A	N/A	24114/25416	30970/25416	60424/27507
Total number of LUTs	N/A	63727	N/A	N/A	56922	63610	119851
Original Network‖ (Precision)	Model C (32-bit)	ResNet-18 (16-bit)	AlexNet (7-bit)	VGG-16 (7-bit)	ResNet-18 (16-bit)	ResNet-18 (8-bit)	MobileNetV2 (32-bit)
Compressed Network (Precision)	FINN[17] (Binary)	ConvNet[13] (Binary)	AlexNet (4-bit)	VGG-16 (4-bit)	ResNet-18 (Ternary)	ResNet-18 (Ternary)	MobileNetV2 (Ternary)
Threshold	N/A	1.5	N/A	N/A	0.7	1.0	0.3
Accuracy (%)†	87.0 (▽3.7)	**92.8** (▽1.3)	N/A (▽3.75)§	N/A (▽3.25)§	69.2 (▽0.3)	67.1 (▽0.8)	68.5 (▽1.3)
Accuracy Recovery (%)	69.7	**71.1**	79.9§	77.4§	**94.9**	**81.4**	**82.9**
Inference Speed (FPS)‡	11.98 (3.87×)	**12.90** (**5.0**×)	N/A (1.48×)	N/A (1.55×)	6.25 (**1.91**×)	6.54 (**1.66**×)	7.14 (**1.65**×)

* Represented as *resources used by original network/resources used by compressed network*

‖ Baseline network

† Represented as *Raw accuracy (Accuracy drop compared to baselines)*

‡ Represented as *Raw speed (Speedup compared to baselines)*

§ Based on top-5 accuracy

4.1 CIFAR-10

We first tested our design on the CIFAR-10 dataset. The experiment was performed with a 16-bit ResNet-18-based network created by us, and a binary ConvNet-based network created by [13], on Ultra96 development board, with a frequency of 100 MHz. Table 4.1 shows the experimental results. In terms of resource utilization, as expected, 16-bit network mainly uses DSPs. A total of 256 DSPs were used for 16×16 computation engines. The binary network, on the other hand, only uses 4 DSPs, which are used for w_{scale} multiplications. It uses more LUTs than the 16-bit network since it mainly performs computation on LUTs. Note that the entire design uses more LUTs than the sum of the two accelerators. This is because extra LUTs are used for interconnects and memory interfaces. Although LUT usage for individual models may seem small, we are actually utilizing more than 90% of LUT resources available for the TwinDNN solution. In terms of performance, a combination of 16-bit and binary network gives more than 71% accuracy recovery, with $5\times$ speedup compared to the baseline 16-bit network. Accuracy recovery is the fraction of the compressed network accuracy drop recovered by the TwinDNN architecture, mathematically defined as $1 - \frac{AccuracyDrop_{combined}}{AccuracyDrop_{compressed}}$ (%).

Table 4.1 also provides a comparison with a previous work on CIFAR-10 [20]. Model C is a customized neural network with the highest accuracy among the three networks that [20] presents. Here, our TwinDNN solution gives 5.8% higher final accuracy with 1.4% extra accuracy recovery compared to their highest accuracy configuration. We also have 7% higher throughput and 113% extra speedup compared to each baseline, even though [20] uses a much larger FPGA ZC706, which has 900 DSPs. This proves that our TwinDNN solution is highly optimized and much more effective than previous studies through parallel execution scheme and efficient resource utilization.

4.2 ImageNet

Next, we tested our design on a much bigger dataset, ImageNet. This time the experiment was performed with two different networks on different FPGA configurations: ResNet-18 on Ultra96 with 150 MHz frequency

and MobileNetV2 on ZCU102 with 200 MHz frequency. For this experiment, a larger FPGA, which is ZCU102, was used along with Ultra96 to show that this design works on a much more scaled environment.

We use 16-bit, 8-bit, and ternary versions of ResNet-18, and 32-bit and ternary versions of MobileNetV2. Ternary networks were used as compressed networks, and other networks were used as original networks. Table 4.1 shows that similar to our CIFAR-10 experiment, ternary networks use many fewer DSPs than other fixed-point networks. Additionally, for ResNet-18, 8-bit network uses more LUTs than 16-bit network, and this is because 8-bit network uses additional logic for bit shifting and introduces additional parallelism by using 1 DSP for 2 multiplications. In general, we can combine two differently quantized accelerators in parallel to increase the throughput with only a small number of extra DSPs compared to the original network accelerators.

In terms of performance, our result suggests that our software—with the original network and using confidence—can identify the majority of inputs that are likely to be incorrect with the compressed network. For ResNet-18 16-bit and ternary configuration, with a threshold value of 0.7, our design shows almost 95% accuracy recovery with more than 1.91× speedup compared to the baseline. For 8-bit and ternary configuration, with a threshold value of 1.0, our design shows more than 81% accuracy recovery with 1.66× speedup compared to the baseline. Finally, for MobileNetV2 32-bit and ternary configuration, with a threshold value of 0.3, our design shows 82.9% accuracy recovery with 1.65× speedup.

Table 4.1 also provides a comparison with CascadeCNN [19] on ImageNet with AlexNet and VGG-16. Exact final accuracy and inference speed are not available, but the accuracy drop and speedup compared to their 7-bit baseline are shown. Accuracy recoveries for these experiments were calculated manually based on their other results, assuming their 7-bit accuracy is the same as their 16-bit accuracy. The result shows that our TwinDNN solution on 16-bit and ternary ResNet-18 provides 17.5% extra accuracy recovery and 36% extra speedup compared to [19]. In addition, note that they used a larger FPGA ZC706, and utilized all available 900 DSPs for both configurations. A total DSP usage of 1800 represents the resources consecutively used by both accelerators. This means that DSPs are still the bottleneck of their design, even for their lowest bit-width configuration. They also did an FPGA

reconfiguration to switch between networks, instead of having both networks on one design. To minimize the time taken by FPGA reconfiguration, they had to batch the images, which may result in non-real-time inference results, as the images have to wait for FPGA reconfiguration for high accuracy network inference. However, because our design implements both networks in parallel, we can ensure that the images are always processed in real-time.

The overall result indicates that our parallel inference scheme with hierarchical network structure works well for accuracy recovery given two optimized neural network accelerators, with a fairly high speedup against the baseline. Our definition of confidence also proves to be a useful metric for verifying neural network output.

CHAPTER 5

CONCLUSION

In this thesis, we proposed a TwinDNN system with a high-accuracy network and a low-latency network using a hierarchical inference logic that will infer high-accuracy network when the prediction of low-latency network is not considered confident. This design becomes especially more effective on the FPGAs where DSP resources are limited compared to LUT resources, as the compressed network latency will mainly depend on the number of LUTs. The ultimate goal of this design would be achieving a true maximum resource utilization of FPGAs, which means utilizing all DSPs and LUTs for the entire time. There are several aspects that make this study stand out. The first aspect is its high flexibility. Although in this project we mostly used ResNet-18 and MobileNetV2, we can put any two ImageNet-based neural networks into our current TwinDNN system without any extra design effort, or we can even put other neural networks through a proper hardware accelerator design process. There are already many neural network accelerators that are built for different focuses: accuracy and speed. We only need to find two accelerators that would fit on the target FPGA, and apply the same logic presented in this thesis for experiments. The second is better concentration. Accelerator development becomes much more difficult when developers need to care about multiple aspects at the same time. However, this work can potentially allow one group of developers to solely focus on increasing accuracy, and the other group of developers to solely focus on reducing latency. It will ultimately reduce the time and effort it takes to build a high-quality accelerator that achieves both accuracy and speed goals.

There are also several aspects where this work can be enhanced further. The first aspect is specialized training. If we can train a specialized network that is trained only to classify between the top few predictions of the compressed network, we may be able to save resources and improve the confidence of the compressed network. Another specialized training scenario can be to

train the compressed network to classify or detect easy objects and train the original network to target difficult objects. This way, the two networks can complement each other better. Second is heterogeneous computing with GPUs. GPUs are usually much more efficient than FPGAs in floating-point operations, and floating-point precision indeed gives higher accuracy than low bit-width fixed-point precision. If we can make the GPU run the original network as floating-point, and make the FPGA run the compressed network, we may be able to achieve an even more efficient solution for this study.

REFERENCES

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *International Conference on Neural Information Processing Systems (NIPS)*, 2012.

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[3] X. Zhang, H. Lu, C. Hao, J. Li, B. Cheng, Y. Li, K. Rupnow, J. Xiong, T. Huang, H. Shi, W.-m. Hwu, and D. Chen, "SkyNet: A hardware-efficient method for object detection and tracking on embedded systems," *Conference on Machine Learning and Systems (MLSys)*, 2020.

[4] S. Zhou, Z. Ni, X. Zhou, H. Wen, Y. Wu, and Y. Zou, "DoReFa-Net: Training low bitwidth convolutional neural networks with low bitwidth gradients," *Computing Research Repository (CoRR)*, 2016.

[5] X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong, W.-m. Hwu, and D. Chen, "DNNBuilder: An automated tool for building high-performance DNN hardware accelerators for FPGAs," *International Conference on Computer-Aided Design (ICCAD)*, 2018.

[6] H. Dbouk, H. Sanghvi, M. Mehendale, and N. Shanbhag, "DBQ: A differentiable branch quantizer for lightweight deep neural networks," *European Conference on Computer Vision (ECCV)*, 2020.

[7] S. Huang, C. Pearson, R. Nagi, J. Xiong, D. Chen, and W.-m. Hwu, "Accelerating sparse deep neural networks on FPGAs," *High Performance Extreme Computing Conference (HPEC)*, 2019.

[8] A. Misra and V. Kindratenko, "HLS-based acceleration framework for deep convolutional neural networks," *International Symposium on Applied Reconfigurable Computing (ARC)*, 2020.

[9] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning," *International Conference on Learning Representations (ICLR)*, 2016.

[10] J. Wang, Q. Lou, X. Zhang, C. Zhu, Y. Lin, and D. Chen, "Design flow of accelerating hybrid extremely low bit-width neural network in embedded FPGA," *International Conference on Field Programmable Logic and Applications (FPL)*, 2018.

[11] M. Courbariaux and Y. Bengio, "BinaryNet: Training deep neural networks with weights and activations constrained to +1 or -1," *Computing Research Repository (CoRR)*, 2016.

[12] M. Courbariaux, Y. Bengio, and J.-P. David, "BinaryConnect: Training deep neural networks with binary weights during propagations," *Neural Information Processing Systems (NIPS)*, 2015.

[13] R. Zhao, W. Song, W. Zhang, T. Xing, J.-H. Lin, M. Srivastava, R. Gupta, and Z. Zhang, "Accelerating binarized convolutional neural networks with software-programmable FPGAs," *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2017.

[14] Y. Chen, K. Zhang, C. Gong, C. Hao, X. Zhang, T. Li, and D. Chen, "T-DLA: An open-source deep learning accelerator for ternarized DNN models on embedded FPGA," *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2019.

[15] F. Li and B. Liu, "Ternary weight networks," *Computing Research Repository (CoRR)*, 2016.

[16] M. Blott, T. B. Preußer, N. J. Fraser, G. Gambardella, K. O'Brien, Y. Umuroglu, M. Leeser, and K. Vissers, "FINN-R: An end-to-end deep-learning framework for fast exploration of quantized neural networks," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 2018.

[17] 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," *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2017.

[18] L. Mocerino and A. Calimera, "CoopNet: Cooperative convolutional neural network for low-power MCUs," *Computing Research Repository (CoRR)*, 2014.

[19] A. Kouris, S. I. Venieris, and C.-S. Bouganis, "CascadeCNN: Pushing the performance limits of quantisation in convolutional neural networks," *International Conference on Field Programmable Logic and Applications (FPL)*, 2018.

[20] S. Amiri, M. Hosseinabady, S. McIntosh-Smith, and J. Nunez-Yanez, "Multi-precision convolutional neural networks on heterogeneous hardware," *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2018.

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

[22] R. Zhao, Y. Hu, J. Dotzel, C. D. Sa, and Z. Zhang, "Improving neural network quantization without retraining using outlier channel splitting," *Computing Research Repository (CoRR)*, 2019.

[23] NVIDIA, "TensorRT." [Online]. Available: https://developer.nvidia.com/tensorrt

[24] Xilinx, "Deep learning with INT8 optimization on Xilinx devices." [Online]. Available: https://www.xilinx.com/support/documentation/white_papers/wp486-deep-learning-int8.pdf