An FPGA-based Accelerator for State-of-art Computation-efficient CNN

Yuhong Li
leeyh@illinois.edu

Ziyan Feng
fziyan2@illinois.edu

Abstract

Convolutional neural networks have been widely deployed in the fields of computer vision and pattern recognition beacuse of their high accuracy. However, the large convolution operations are computing intensive and requires a powerful computing platform such as GPU. That makes it difficult to apply CNN to portable devices. Recently, Deep Neural Networks(DNN) for mobile devices are widely discussed. Computation-efficient architectures like ShuffleNet and MobileNet can achieve state-of-the-art performance with limited computation resources. By adopting depthwise separable convolution to replace the standard convolution for embedded platforms, they reduce the number of parameters and also require a relatively small number of arithmetic operations with only limited accuarcy loss. But they are usually implemented at the software level. The use of these mobile networks without any optimization may not provide sufficient performance when high processing speed is required. Considering FPGA is a promising candidate for the acceleration of Deep Neural Networks, in this project, we designed an FPGA-based CNN accelerator of ShuffleNetV2, a most recent novel DNN which is suitable for edge devices. We firstly trained ShuffleNetV2 on Cifar10 dataset. By comparing different quantization methods, we converted the floating-point model to 8-bit fixed-point without significant accuracy loss. By using high level synthesis(HLS)-besed design, we need to thouroughly consider on-chip computation, buffering resources and off-chip memory accesses to minimize the total latency. Our goal is to accelerate the network as much as possible without hurting the overall accuracy within the resource limitation of Zedboard.

1. Introduction

Nowadays, convolutional neural networks have attracted most interest, due to their superior performance in some tasks such as image classification, semantic segmentation, and object detection and tracking. Also, this technique has also been widely used in the industry such as autonomous driving, video surveillance, speech recognition and so on.

CNN is a computing intensive model, and it consumes huge amount of computing power in training and inference. Graphics Processing Units(GPU) are often selected as the platform for CNN because of its powerful computing ability. However, GPU's natural of high power consumption constraints its application in embedded scenario such as portable devices and wearable systems. Therefore, Field-Programmable Gate Arrays (FPGAs) and Application Specific Integrated Circuits (ASICs) are adopted in the neural network applications as the replacement of GPU. Moreover, computation-efficient networks are proposed to meet the resources requirement of edge devices because real world tasks often aim at obtaining the trade off between accuracy and a limited computational budget, given by target platform (e.g., FPGA) and application scenarios (e.g., auto driving requires low latency). This motivates a series of works towards light-weight architecture design and better speed-accuracy tradeoff, including Xception [1], MobileNet [2], MobileNetV2 [3], ShuffleNet [4], and ShuffleNetV2 [5], to name a few.

During the exploration, we saw the several gaps between CNN architectures and accelerator designs in the following areas:

Inefficient CNN models: Many FPGA accelerators still focus on obsolete, inefficient models such as AlexNet and VGG16, which require greater storage and computational resources than newer, efficient models that achieve the same or higher accuracy. With an inefficient model, an accelerator with high throughput in terms of GOPs can actually have low inference speed in terms of FPS, where FPS is the more essential metric of efficiency. To achieve AlexNet-level accuracy, SqueezeNet is 50x smaller than AlexNet; SqueezeNext is 112x smaller. Even more, MobileNetV2 is 60x smaller than Alexnet with 28% higher accuraccy on ImageNet dataset. However, not many designs target those efficient models. Additionally, techniques for accelerating older models may not generalize to newer ConvNets.

CNN operators: ConvNet models contain many different types of operators. Commonly used operators include 1×1, 3×3, 5×5 conventional convolutions, 3×3 max pooling, *etc*. More recent models also contain depth-wise, group, dilated, and factorized convolutions. Not all of these operators can be efficiently implemented on FPGAs. If a ConvNet contains many different types of operators, one must either allocate more dedicated compute units or make the compute unit more general. Either solution can potentially lead to high resource requirement, limited parallelism, and more complicated control flow. Also, hardware development will require more engineering effort.

Quantization: ConvNet quantization has been widely used to convert weights and activations from floating point to low-precision numbers to reduce the computational cost. However, many of the previous methods are not practically useful for FPGAs due to the following problems: 1) Quantization can lead to serious accuracy loss, especially if the network is quantized to ultra low precision numbers (less than 16 bits). 2) Many of the previously presented quantization methods are only effective on large ConvNet models such as VGG16, AlexNet, ResNet, etc. Since those models are known to be redundant, quantizing those to low-precisions is much easier. We are not aware of any previous work tested on efficient models such as MobileNet or ShuffleNet. 3) Many methods do not quantize weights and activations directly to fixed point numbers. Usually, quantized weights and activations are represented by fixed-point numbers multiplied by some shared floating point coefficients. Such representation requires more complicated computation than purely fixed-point operations, and are therefore more expensive.

Recent years have seen rapid development of DNNs for implementation on FPGAs. DNNs are composed of layers of regular computations such as convolution and pooling. High level synthesis (HLS) is well suited to optimize the regular computations of network layers. However, there are significant challenges in managing computational complexity, on-chip memory limitation, and external memory bottlenecks. Each layer in a DNN features different computational and memory bandwidth demand; effective design of a network demands both different optimization strategies based on layer type as well as different optimization parameters between different instances of the same layer. To produce optimal network implementations under resource constraints, we must determine best on-chip memory usage and external memory access patterns, explore layer implementation options and determine how to best allocate limited FPGA resources among the layers in order to minimize overall latency.

2. Contribution

- We will transplant ShuffleNetV2, a most recent computation-efficient neural network from software implementation to the FPGA board. We will modify and train this hardware-friendly efficient network on Cifar-10 dataset.

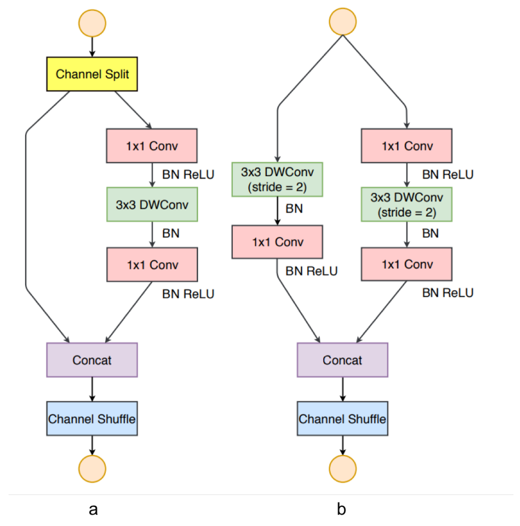


Figure 1: (a) The basic block of ShuffleNetV2 (b) The downsampling block of ShuffleNetV2

- We will design an FPGA-based accelerator for ShuffleNetV2. In order to do this, we convert the model to 8-bit fixed point without significant accuracy loss. Also, we convert the Pytorch model to the HLS model and implement the model on ZedBoard.

- We will try to explore a solution to build a high performance hardware accelerator for such computation-efficient neural networks other than ShuffleNetV2. And we will try to explore a methodology for scalable design in order to implement on different types of FPGA by balancing different on-chip resource and perforamance speedup.

3. Description

3.1. ShuffleNetV2

The main modules of ShuffleNetV2 is shown in fig 1. The network is stacked by repeating these modules except the first and the last convolutional layers. The main idea of designing these modules is to try the best to reuse the feature maps. In the basic block, the input feature maps are divided into two parts, only one part go through series of convolution operations. In the downsampling block, the feature maps go through two flows seperately and then are concatenated together so that the output channels are expanded.

3.2. Training on Cifar-10 dataset

The CIFAR-10 dataset consists of 60000 32×32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images. The dataset

Layer	Output size	Stride	Output Channels
Image	32×32	1	3
Conv1	32×32	1	24
Stage2 Down	16×16	2	48
Stage2 Basic	16×16	1	48
Stage3 Down	8×8	2	96
Stage3 Basic	8×8	1	96
Stage4 Down	4×4	2	192
Stage4 Basic	4×4	1	192
Conv5	4×4	1	512
GlobalPool	1×1	1	512
FC			10

Table 1: Configuration of modified ShuffleNetV2.

is divided into five training batches and one test batch, each with 10000 images. Samples are shown in figure 2.

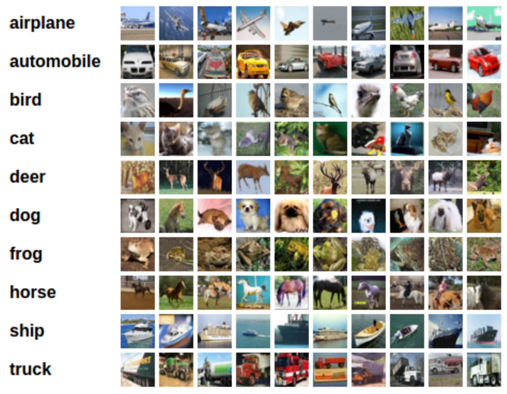


Figure 2: Samples in Cifar-10 which contains 10 classes.

To make the network suitable for the Cifar-10 dataset, we modified the ShuffleNetV2 0.5X. Firstly, we remove the maxpooling layer before the first convolutional layer. Secondly, we change the first convolutional layer's stride to 1 in order to make the network suitable for the input image size. The structure of the network is shown in the table 1.

3.3. Quantization of the CNN

In the quantization process, we firstly absorbed the batch normalization layer into the convolutional layer. And retrained the network while gradually squeeze the bitwidth of the weights.

3.3.1 Fusing batch normalization and convolution in runtime

Batch Normalization(BN) is a popular method used in modern neural network architecture. Most of today's state-of-the-art convulutional neural networks are incorporate with batch normalization layer. Batch normalization could Improves gradient flow which can be used on very deep models. (Resnet *etc*.)

3.3.2 Batch normalization

Let x be a signal (activation) within the network that we want to normalize. Given a set of such signals $x_1, x_2, \ldots, x_n$ coming from processing different samples within a batch, each is normalized as follows:

$$\hat{x}_i = \gamma \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta$$

The values μ and σ^2 are the mean and variance computed over a batch, ϵ is a small constant included for numerical stability, γ is the scaling factor and β the shift factor.

During training, μ and σ are recomputed for each batch:

$$\mu = \frac{1}{n} \sum x_i$$

$$\sigma^2 = \frac{1}{n} \sum (x_i - \mu)^2$$

The parameters γ and β are slowly learned with gradient descent together with the other parameters of the network. During test time, we usually do not run the network on a batch of images. Thus, the previously mentioned formulae for μ and σ cannot be used.

Instead, we use their estimates computed during training by exponential moving average. Let us denote these approximations as $\hat{\mu}$ nad $\hat{\sigma}^2$.

3.3.3 Fusing batch normalization with a convolutional layer

Let $\mathbf{W}_{BN} \in \mathbb{R}^{C \times C}$ and $\mathbf{b}_{BN} \in \mathbb{R}^{C}$ denote the matrix and bias from the above equation, and $\mathbf{W}_{conv} \in \mathbb{R}^{C \times (C_{prev} \cdot k^2)}$ and $\mathbf{b}_{conv} \in \mathbb{R}^{C}$ the parameters of the convolutional layer that precedes batch normalization, where C_{prev} is the number of channels of the feature map F_{prev} input to the convolutional layer and $k \times k$ is the filter size.

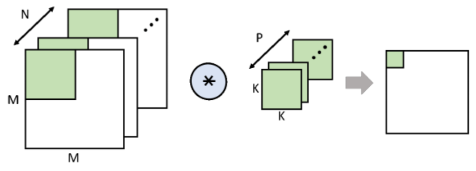
Given a $k \times k$ neighbourhood of F_{prev} unwrapped into a $k^2 \cdot C_{prev}$ vector $\mathbf{f}_{i,j}$, we can write the whole computational process as:

$$\hat{\mathbf{f}}_{i,j} = \mathbf{W}_{BN} \cdot (\mathbf{W}_{conv} \cdot \mathbf{f}_{i,j} + \mathbf{b}_{conv}) + \mathbf{b}_{BN}$$

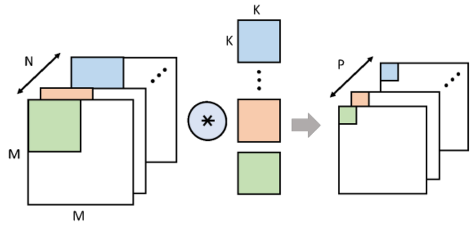
Thus, we can replace these two layers by a single convolutional layer with the following parameters:

filter weights: $\mathbf{W} = \mathbf{W}_{BN} \cdot \mathbf{W}_{conv}$;

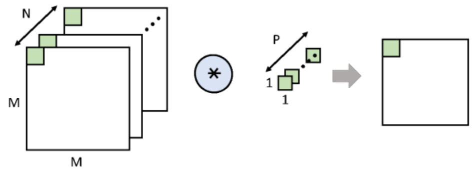
bias: $\mathbf{b} = \mathbf{W}_{BN} \cdot \mathbf{b}_{conv} + \mathbf{b}_{BN}$.



(a) standard convolution



(b) depthwise convolution



(c) pointwise convolution

Figure 3: The Depthwise Convolution Compared with Other Convolutoins

3.3.4 Depthwise Separable Convolution

Depthwise separable convolution was first introduced in [1]. It is a kind of factorized convolution, it factorizes the standard convolution into a depthwise convolution plus a pointwise convolution. Fig 3 shows how the standard convolution, depthwise convolution and pointwise convolution works. In standard convolution, each input channel has to do a convolution with one specific kernel, and then the result is the sum of the convolution results from all channels. While in depthwise separable convolution case, depthwise convolution is the first step, performing the convolution for each input channel individually. The next step is to do convolution in pointwise, which is actually a standard convolution with kernel size 1x1. Comparing to standard convolution, using depthwise separable convolution considerably reduces the number of mathematical operations and the number of parameters.

As shown in 3, the input feature map is $M \times M \times N$, and the kernel size is $K \times K \times N \times P$. Then the number of weights needed for standard convolution is [2]

$$W_{regular} = K \times K \times N \times P$$

And the corresponding number of operations is

$$O_{regular} = M \times M \times K \times K \times N \times P$$

However, for depthwise convoluton, the total number of weights is

$$W_{depthwise} = K \times K \times N + N \times P$$

and the number of operations is

$$O_{depthwise} = M \times M \times K \times K \times N + M \times M \times N \times P$$

Thus, the reduction of weights and bias could be calculated

$$F_O = \frac{O_{depthwise}}{O_{regular}} = \frac{1}{P} + \frac{1}{K^2}$$

$$F_w = \frac{W_{depthwise}}{W_{regular}} = \frac{1}{P} + \frac{1}{K^2}$$

3.4. Hardware Designs

3.4.1 System Overview

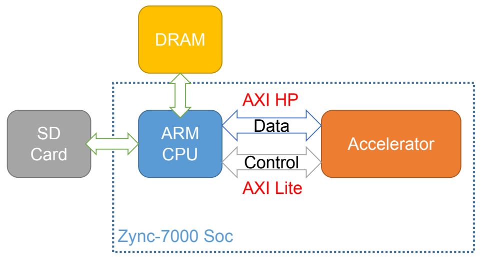


Figure 4: System Overveiw of the accelerator system

The block diagram in Fig 4 is the system overview of our acclearator system. The accelerator designed by us is on the PL side. It is responsible for the computation of the whole network. All the parameters of image, weights and bias are stored in an SD card of Zedboard. An ARM CPU on board is responsible for reading the parameters into the off-chip DRAM and communicate with the accelerator to send the control signal and data. The communication is done through AXI interface. The data is transferred through AXI high performance interface and the control signal is sent through AXI lite interface.

3.4.2 Accelerator Architecture

Fig 5 shows the overview of the acceleartor. The computation engine is responsible for the CNN operations, including the depthwise convolution, pointwise convolution and regular convolution. We also integrate ReLu into the CNN computation engine. Before the computation of each convolution layer, we load the weights and bias of this layer into the the weights and bias buffer which are on-chip BRAM. We use two feature map buffers as double buffers to store the input feature and output feature of each layer to save the usage of BRAM.

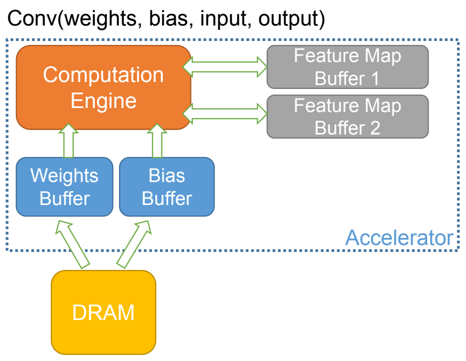
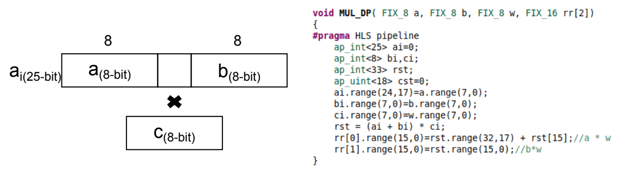


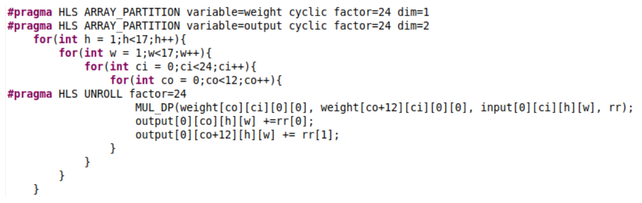
Figure 5: The block diagram of the accelerator

3.4.3 Optimization methods using Vivado HLS



```
void MUL_DP( FIX_8 a, FIX_8 b, FIX_8 w, FIX_16 rr[2])
{
#pragma HLS pipeline
    ap_int<25> ai=0;
    ap_int<8> bi,ci;
    ap_int<33> rst;
    ap_uint<18> cst=0;
    ai.range(24,17)=a.range(7,0);
    bi.range(7,0)=b.range(7,0);
    ci.range(7,0)=w.range(7,0);
    rst = (ai + bi) * ci;
    rr[0].range(15,0)=rst.range(32,17) + rst[15];//a * w
    rr[1].range(15,0)=rst.range(15,0);//b*w
}
```

Figure 6: Double Pump to use the DSP48E to compute the 8-bit fix point numbers



```
#pragma HLS ARRAY_PARTITION variable=weight cyclic factor=24 dim=1
#pragma HLS ARRAY_PARTITION variable=output cyclic factor=24 dim=2
    for(int h = 1;h<17;h++){
        for(int w = 1;w<17;w++){
            for(int ci = 0;ci<24;ci++){
                for(int co = 0;co<12;co++){
#pragma HLS UNROLL factor=24
                    MUL_DP(weight[co][ci][0][0], weight[co+12][ci][0][0], input[0][ci][h][w], rr);
                    output[0][co][h][w] +=rr[0];
                    output[0][co+12][h][w] += rr[1];
                }
            }
        }
    }
```

Figure 7: Use loop unrolling and array partition to compute in parallel.

1. **Use double pump to fully utilize the DSP48E resource on board.**
   Since we are using 8-bit fixed point, the HLS tool doesn't use DSP48E to do the computation. It would use FF and LUT instead, which results in the increased latency and overuse of LUT and FF. The reason is that DSP48E is usually used for doing 25x18 computation, but our fix point number has only 8 bits, which can not fully utilize the computing power of DSP48E. We use double pump to solve the problem. The explanation of this optimization is shown in Fig 8. This method is using DSP48E to compute the output of two multiplying operations in one time. In the example of Fig 8, we put an 8-bit fix point number a into a 25-bit number ai, and use ai to add b and multiply c. Note that when we extract the final result $a*c$, we must consider the sign bit of $b*c$ (in the code above, it is result[15]), because its sign bit would be extended and add to $a*c$. In this way, we could utilize the DSP48E to do the calculation of 8-bit fix point to save the usage of LUT and improve the performance.

2. **Use loop unrolling and array partition to exploit the parallelism**
   For the convolution layer, we use loop unrolling and array partition to exploit the computation parallelism to accelerate the computation. First, we change the structure of each convolution layer to make the loop a perfect loop, which means that only in the most nested loop, there is statement. Then we could change the order of every nested loop. We make the loop of output channel i.e., co loop to be the most nested loop and unroll this loop. According to the unroll factor, we also partition the weights array and the output array in the corresponding dimension of co. We decide the unroll factor of each layer according to the times of function calls of it and its latency. If a certain convolution function is called for many times and consume a lot of time, we would increase the unroll factor in order to use more resource to reduce its latency. For some convolution layers which have small amount of computation, we don't distribute much resource on it. In this way, we could use our limited on-board resource more efficiently and gain the best performance within the limitation.

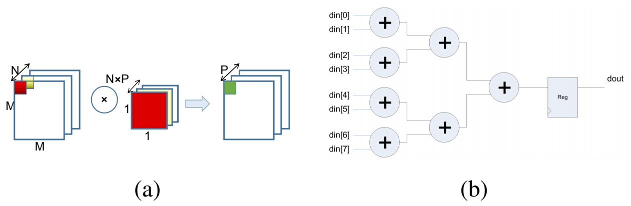


Figure 8: Use adder tree to compute the pointwise convolution

3. **Use adder tree for pointwise convolution**
   The pointwise convolution is the regular convolution whose convolution kernel is 1x1. Fig 8a shows the computation pattern of pointwise convolution. To calculate each element in the output feature(i.e., $dout$ in Fig 8b), it needs to go through every input channel to find the corresponding element to multiply the weight and calculate the sum of these products. For this accumulation process, we could use adder tree to do calculate each element in the output feature. It reduce the time complexity from $O(n)$ to $O(log(n))$. By using adder tree to calculate the result of pointwise convo-

weights type	feature maps type	if retrained	accuracy
floating32	floating32		90.37
fixed $< 1, 6 >$	fixed $< 1, 6 >$	no	86.2
fixed $< 1, 6 >$	fixed $< 1, 6 >$	yes	88.5

Table 2: The results of the quantization. $< 1, 6 >$ stands for the 1 bit for integer and 6 bits for the floating part.

lution we could improve our performance while using limited amount of resource.

4. Experiments

4.1. ShuffleNetV2 on Cifar10

In the experiments, we set the batch size as 128 and the learning rate as 0.05 with the learning rate to the initial learning rate decayed by 2 every 30 epochs. We choose the stochastic gradient descent as the optimizer. The test accuracy curve is shown in the figure 9. We used modified ShuffleNetV2 0.5x to reach the 90.37% accuracy which is better than the AlexNet with 100x less FLOPs.

4.2. Quantization results

We replaced the original ReLU operation with the ReLU2 to constraint the output of the feature maps ranges from 0 to 2. Then we absorbed the batch normalization into convolutional layers. The simulation results are shown in the table 2.

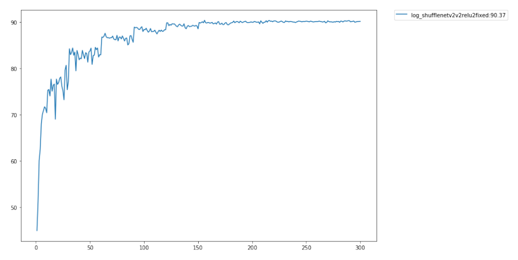
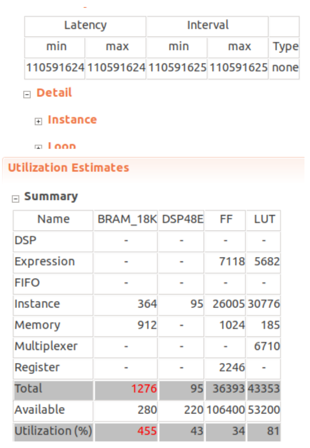


Figure 9: Testing accuracy curve on Cifar-10.

4.3. HLS Synthesis

To implement the whole network on board, we first use the ARM CPU on Zedboard to test the number of cycles of CPU as our baseline. Then we use Vivado HLS 2017.2 to optimize our C code, we also choose the Zedboard as our platform of simulation in the HLS tool. We use the AXI Timer IP provided by Xilinx to measure the time of baseline and our accelerator. First, we implement the whole network into the HLS tool without any pragma or any optimization. The synthesis result is shown in Fig 10a. From the report of HLS, we could see that the latency number is very large



Latency		Interval		
min	max	min	max	Type
110591624	110591624	110591625	110591625	none

Detail

Instance

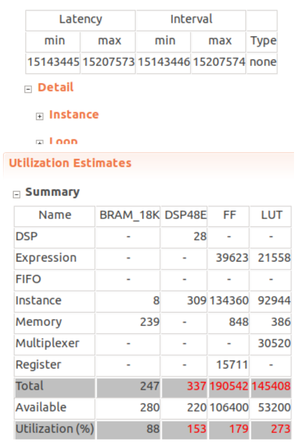
Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	7118	5682
FIFO	-	-	-	-
Instance	364	95	26005	30776
Memory	912	-	1024	185
Multiplexer	-	-	-	6710
Register	-	-	2246	-
Total	1276	95	36393	43353
Available	280	220	106400	53200
Utilization (%)	455	43	34	81

(a) Without optimization



Latency		Interval		
min	max	min	max	Type
15143445	15207573	15143446	15207574	none

Detail

Instance

Loop

Utilization Estimates

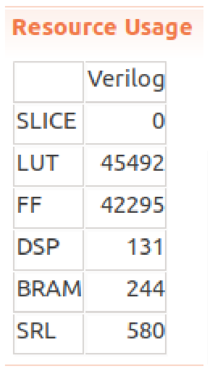
Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	28	-	-
Expression	-	-	39623	21558
FIFO	-	-	-	-
Instance	8	309	134360	92944
Memory	239	-	848	386
Multiplexer	-	-	-	30520
Register	-	-	15711	-
Total	247	337	190542	145408
Available	280	220	106400	53200
Utilization (%)	88	153	179	273

(b) After optimization

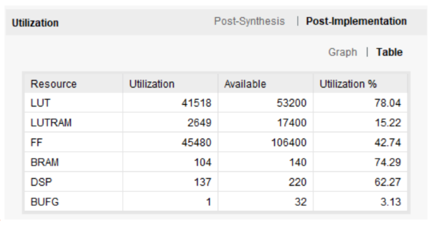
Figure 10: Latency and resource usage reported in Vivado HLS tool

before optimization. Also, the BRAM usage is over 400 %, because we haven't use double buffer for the weights, bias, and features. After our optimization, the synthesis result is shown in Fig 10b. We could see that the latency has been reduced and the BRAM usage is limited to fit to the Zedborad. However, we found that the DSP, FF and LUT usage are beyond the resource limitation of Zedboard. But later we found that after we export the RTL from HLS toll, our resource would be reduced a lot and could be implemented on Zedboard. The reason of this is that when exporting to RTL, the HLS would do some optimization. The result of exporting RTL is shown in 11a. Compared with the HLS synthesis report, we found that the DSP, FF, and LUT usage are reduced a lot so that we could implement it on Zedboard.



Resource Usage

	Verilog
SLICE	0
LUT	45492
FF	42295
DSP	131
BRAM	244
SRL	580



Utilization Post-Synthesis | Post-Implementation

Graph | Table

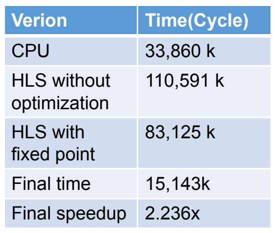
Resource	Utilization	Available	Utilization %
LUT	41518	53200	78.04
LUTRAM	2649	17400	15.22
FF	45480	106400	42.74
BRAM	104	140	74.29
DSP	137	220	62.27
BUFG	1	32	3.13

(a) export RTL (b) Post Implementation usage

Figure 11: Latency and resource usage reported after exporting RTL

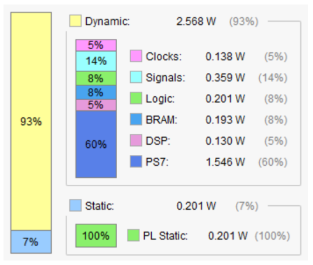
4.4. Vivado Synthesis

The final result of resource usage is the result of synthesis and implementation in Vivado. The final resource usage is shown in Fig 11b. We also present the speed up to our baseline in Fig 12a. The power usage of our system is shown in Fig 12b. Because we have limited resource on the Zedboard, so we could only do some limited optimization. There are some functions that we don't implement any optimization in order to save the usage of LUT and BRAM. But we have spared no effort to improve the performance as much as possible and ensure that the resource is used to the most time-consuming part. Finally we reach a speedup of 2.236× against the CPU baseline, and we successfully implement the whole network on board within the resource limitation.



Verion	Time(Cycle)
CPU	33,860 k
HLS without optimization	110,591 k
HLS with fixed point	83,125 k
Final time	15,143k
Final speedup	2.236x

(a) Final speedup



(b) On-chip Power Usage

Figure 12: Final speedup and power usage

5. Conclusion

In this project, we adopt an algorithm-hardware co-design approach to develop a CNN accelerator based on ShuffleNetV2. We optimize the networks operators by modifying the downsampling process and train the network on Cifar-10 dataset. We quantize both the networks weights and the activations to 8-bit fixed-point numbers with less than 2% accuracy loss. These quantizations very well exploit the nature of FPGA hardware. We implemented the network on ZedBoard Soc systems.

6. Future work and scope

For the future works, we will focus on further optimization. For example, we can add more layers in the dataflow architecture to improve the compute-to-communication ratio. We can also design more general IPs to accelerate the network. Correspondingly, we will need to adjust the network such that the computation subgraphs are more symmetric.

References

[1] François Chollet. Xception: Deep learning with depthwise separable convolutions. *arXiv preprint*, pages 1610–02357, 2017.

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

[3] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4510–4520. IEEE, 2018.

[4] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. *CoRR*, abs/1707.01083, 2017.

[5] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. Shufflenet v2: Practical guidelines for efficient cnn architecture design. *arXiv preprint arXiv:1807.11164*, 1, 2018.