Acceleration of Residual Binarized Neural Network

Yan Chen
School of Information Science
Japan Advanced Institute of Science and Technology
Nomi, Ishikawa, 923-1292, Japan
chenyan@jaist.ac.jp

Kiyofumi Tanaka
School of Information Science
Japan Advanced Institute of Science and Technology
Nomi, Ishikawa, 923-1292, Japan
kiyofumi@jaist.ac.jp

Abstract—In this paper, we discuss a state-of-the-art method of accelerating the Residual Binarized Neural Network (ReBNet) [1] by replacing fixed-point number multiplication with logical shift operation. We designed an end-to-end framework for training binary neural networks, on which the conversion to logical-shift-based multiplication on software and hardware accelerators implemented on FPGA is performed. Compared to ReBNet [1], we got similar accuracy in several datasets, and our hardware resource usage in the same degree of parallelism as ReBNet was much lower. It is concluded that our design can be implemented on a smaller device or with larger degree of parallelism in the same device.

I. Introduction

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 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 [2], and ResNet-50 has 25.5 million of parameters and needs 3.8 GFlops to inference one picture. So many parameters and floating-point operations make it difficult for even high-end GPU to run the modern CNNs in real-time, and the batch-based acceleration of GPUs results in high average latency.

CNNs' models should be trained with 32-bit floating-point precision or 32-bit/16-bit mixed floating-point precision [3], while the inference process does not require such high precision. 8-bit integer quantization [4] is popular for inference in CNNs on embedded or mobile devices. Even 1-bit binarization can achieve high accuracy.

In BinaryNet [5], 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. 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 binarized neural networks (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. In Xnor-Net [6], Rastegari et al. added scaling factor to weight, which makes their design get higher accuracy in the ImageNet dataset.

FINN [7] 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 [8].

BNNs with 1-bit activation and 1-bit weights lead to very limited accuracy. On the other hand, Residual Binarized Neural Network (ReBNet) [1] which has multiple-bit activation and 1-bit weights gets much better accuracy, and is comparable with floating-point precision. ReBNet [1] 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.

In this paper, we resolve the problems mentioned above. We achieve similar accuracy to ReBNet [1], while our hardware accelerator takes only $\frac{1}{M+2}$ DSP48s of ReBNet in each processing element (PE). As a result, our design does not run out of DSP48s and saves many LUTs so that we can apply throughput optimization and increase the degree of parallelism. In addition, we apply optimization to the Pooling Unit to decrease LUT and BRAM usage. For the same resource-limited FPGA device, our design achieves 8× higher throughput on average, lower power usage per FPS, and higher running frequency than ReBNet.

In section II, we describe existing techniques in BNNs and hardware design. Section III proposes our accelerator which improves ReBNet. In section IV, our design is compared with ReBNet in different settings. Finally, we conclude the paper in Section V.

II. Preliminaries

In this section, we show the techniques used in BNN and explain residual binarization in ReBNet [1] in particular. In addition, we present how the FINN framework [7] implements BNN in parallel on FPGA efficiently. For more details, refer to the original papers.

A. 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 1 shows how to convert a fixed-point input x to an approximate binary value e_i and encode them 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 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, $e = \sum_i^M \gamma_{e_i} \times sign(r_i)$ is the approximate binary value for input x, and the relationship between input x and e is illustrated in Figure 2. γ_e is learned during the training phase.

B. 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

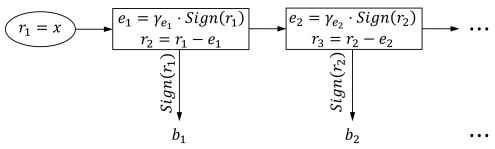


Fig. 1. Encoding multiple-level residual-binarized bits.

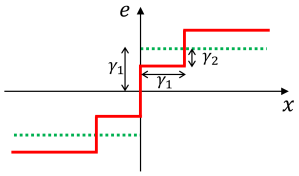


Fig. 2. Relationship between input x and approximate binary value e when M=2.

feature map vector $\vec{x}$ and kernel vector $\vec{w}$. These dot products operations are as follows.

$$dot(\vec{x}, \vec{w}) = \sum \vec{x}_i \times \vec{w}_i \tag{1}$$

In BNNs, $\{\vec{x}, \vec{w}\}$ are restricted to binary values which are $\{\pm\gamma_x, \pm\gamma_w\}$. According to [5], dot products of binarlized $\{\vec{b}_x, \vec{b}_w\}$ 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 (Figure3). Let $\vec{x} = \gamma_x \vec{s}_x$ and $\vec{w} = \gamma_w \vec{s}_w$, while $\{\gamma_x, \gamma_w\}$ are scalar values and $\{\vec{s}_x, \vec{s}_w\}$ are sign vectors which only contains ± 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:

$$\begin{aligned} dot(\vec{x}, \vec{w}) &= \gamma_x \gamma_w dot(\vec{s}_x, \vec{s}_w) \\ &= \gamma_x \gamma_w XnorPopcount(\vec{b}_x, \vec{b}_w). \end{aligned} \tag{2}$$

In ReBNet [1], 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:

$$\begin{aligned} dot(\vec{e}, \vec{w}) &= \sum_i^M \gamma_{e_i} \gamma_w dot(\vec{s}_{e_i}, \vec{s}_w) \\ &= \sum_i^M \gamma_{e_i} \gamma_w XnorPopcount(\vec{b}_{e_i}, \vec{b}_w), \end{aligned} \tag{3}$$

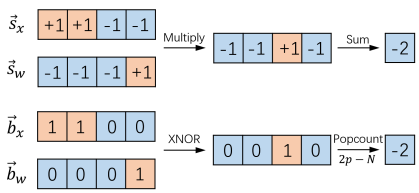


Fig. 3. Dot-Product and XnorPopcount, where p is the number of positive bits and N is the bit width of input.

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

C. Threshold-based Batch Normalization

CNNs generally include a batch normalization layer between fully-connected layer or convolution layer and activation function. 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. γ and β are updated by back propagation during learning. Processing in batch normalization is:

$$output = \frac{x-\mu}{\sqrt{\sigma^2+\epsilon}} \cdot \gamma + \beta, \tag{4}$$

where ϵ is a fixed value which is close to 0, and x is the input value. 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 $\tau = \mu - \frac{(\beta \times \sqrt{\sigma^2+\epsilon})}{\gamma}$. RebNet uses τ to get a difference $\mathcal{D}$ with input x, and then uses this difference $\mathcal{D}$ to multiply a scaling factor $\alpha = \frac{\gamma}{\sqrt{\sigma^2+\epsilon}}$, and put the product to residual-binarization activation function.

D. MVTU and SWU

Since ReBNet inherited the design of FINN, a fully-connected layer uses a Matrix Vector Threshold Unit (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 (PE) in an MVTU, and 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. The PE in ReBNet is illustrated in Figure 4. Each PE can process multiple 1-bit data in parallel, and the number of data pairs is "SIMD width". The number of PE and the SIMD width decide the degree of parallelism. The processing in each PE before encoding is:

$$\sum(\sum_{i}^{M}(\gamma_{e_i} \cdot \gamma_w \cdot XnorPopcount(\vec{b}_{e_i}, \vec{b}_w))) \cdot \alpha - \tau \cdot \alpha, \tag{5}$$

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.

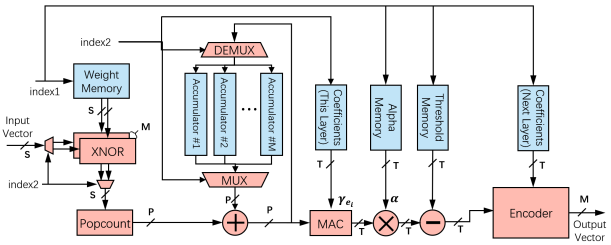


Fig. 4. Processing Element in ReBNet [1]. While index1 is PE index, index2 is layer index. S is data width of input (SIMD width), P is data width of popcount accumulator, T is data width of all of the fixed-point values, and M is the number of residual-binarization levels.

III. 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.

A. 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 $\frac{1}{2}$. Therefore, we redesigned the Residual-Binarization activation function, reduced the elements of binarize factor γ_e form 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 $e = \sum_i^M \frac{1}{2^{i-1}}\gamma_e \times sign(r_i)$. Assumme that $\mathcal{L}$ is loss function, that it uses full-precision in backpropagation, and that sign function causes vanishing gradient. According to the literature of BinaryNet [5], 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:

$$clip'(x,-1,+1) = 1_{|x|\leq 1} = \begin{cases} 1 & |x| \leq 1 \\ 0 & otherwise \end{cases} \tag{6}$$

and $e = \sum_i^M \frac{1}{2^{i-1}}\gamma_e \times clip(r_i,-1,+1)$ in backpropagation. The derivatives of cost function $\mathcal{L}$ with respect to γ is computed by chain rule as:

$$\frac{\partial \mathcal{L}}{\partial \gamma_e} = \frac{\partial \mathcal{L}}{\partial e}\frac{\partial e}{\partial \gamma_e} = \sum_i^M \frac{1}{2^{i-1}} clip(r_i,-1,+1), \tag{7}$$

and for input x:

$$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial e}\frac{\partial e}{\partial x} = \sum_i^M \frac{1}{2^{i-1}}\gamma_e \times 1_{|r_i|\leq 1}. \tag{8}$$

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.

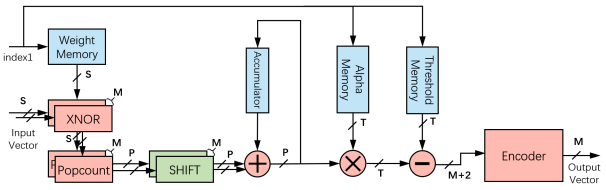


Fig. 5. Processing Element in our design. While index1 is PE index, S is data width of input (SIMD width), P is data width of popcount accumulator, T is data width of all of the fixed-point values, and M is the number of residual-binarization levels.

B. 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 don't 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:

$$\begin{aligned} \gamma' &= \gamma \cdot \gamma_{r_{cur}} \\ \beta' &= \beta \cdot \gamma_{r_{cur}} \\ \mu' &= \mu \cdot \gamma_{r_{prev}} \\ \sigma' &= \sqrt{(\sigma^2 + \epsilon) \cdot \gamma^2_{r_{prev}}}, \end{aligned} \tag{9}$$

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.

C. Processing Element

As mentioned above, we reduced the components in PE in Figure 5. 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:

$$\sum(\sum_{i}^{M}(XnorPopcount(\vec{b}_{e_i}, \vec{w}) << (M - i))) \cdot \alpha - \tau \cdot \alpha. \tag{10}$$

Then we deliver the result of Formula (10) 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 decimal part has a constant value of 1, which is 0.5 in decimal since this can avoid comparison between the same values.

In addition, we apply throughput optimization in the processing element so that the initiation interval 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.

IV. Experiments

We use Keras [9] with TensorFlow [10] 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 [11] to perform high-level-synthesis for the IP core of the accelerator we designed. Then, logic synthesis, implementation, and bit-stream generation are done on Vivado.

We implemented different accelerators for several data sets: MNIST, CIFAR-10, and SVHN, and compare them with ReBNet for the same data sets. We target several

TABLE I
TARGET FPGAS.

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

TABLE II
NEURAL NETWORK ARCHITECTURES FOR EACH DATASET.

Name	Architecture
Arch 1 (MNIST)	input(768)-FC(256)-BN-RB-FC(256)-BN-RB-FC(256)-BN-RB-FC(10)-Softmax
Arch 2 (CIFAR-10 & SVHN)	input($3 \times 224 \times 224$)-Conv(64)-BN-RB-Conv(64)-BN-RB-MP-Conv(128)-BN-RB-Conv(128)-BN-RB-MP-Conv(256)-BN-RB-Conv(256)-BN-RB-FC(512)-BN-RB-FC(512)-BN-RB-FC(10)-Softmax

FPGA devices with different sizes. The resources of the target devices are presented in TABLE I where 7z020 is a resource-limited device and 7z100 is a large device which provides a large amount of DSP48s. The highest degree of parallelism described in FINN can be implemented on the large device 7z100. However, the same degree is impossible on the 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.

A. Accuracy

We adopt neural network architectures similar to ReBNet [1] as those in TABLE II, where RB represents Residual-Binarization. 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 convolution 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 full-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.

The accuracy of a neural network is dependent on architecture and training epochs. We fix the training epochs to 200 for all of the datasets during model training, and TABLE III summarizes the highest accuracy for each dataset in different residual-binarize levels. The accuracy of our design is from the simulation of the post-convert model which is based on threshold subtraction run on Keras, and the precision of parameters is limited to that on our hardware. It is found that our method of isometric residual-binarization achieves accuracy similar to ReBNet. We carefully considered the range of each parameter and made sure the hardware produces the same intermediate data and output as the simulation on Keras in testbench, while ReBNet on hardware may have some errors with models run on Keras.

TABLE III
ACCURACY COMPARISON ON EACH METHOD.

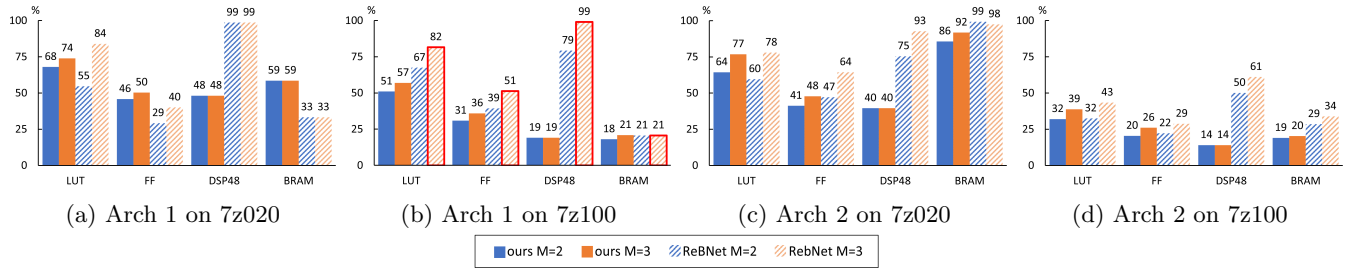
	FP32	FINN	M	ReBNet	This Work
MNIST	0.9822	0.9583	2	0.9799	0.9803
			3	0.9799	0.9793
CIFAR-10	0.8903	0.8010	2	0.8469	0.8497
			3	0.8618	0.8632
SVHN	0.9765	0.9490	2	0.9677	0.9645
			3	0.9690	0.9670

B. Hardware Implementation

According to FINN [7], we use $Fold$ to describe the degree of parallelism, where $Fold$ means iterations of an MVTU process for one picture. The $Fold$ can be calculated by $\frac{\text{Synapses} \times \text{Neurons}}{\text{PECount} \times \text{SIMDWidth}}$ for the MVTUs of fully-connected layers, and it should be multiplied by sampling cycles of SWU for convolutional layers 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 $Flod$ can be the bottleneck of the dataflow. TABLE IV shows the maximum $Fold$ with which our design and ReBNet can be implemented on 7z020 and 7z100 for each architecture. As for 7z100, we implemented the designs with the FINN [7] framework's maximum degree of parallelism. 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 bottleneck in implementations on small devices.

Figure 6 shows the resource usages on each degree of parallelism and TABLE IV includes maximum frequency, maximum throughout calculated via the maximum frequency according to FINN [7], and power usage of chip. All of the results are from reports of Vivado HLS or Vivado with target frequency of 200MHz.

Our design reach much higher degrees of parallelism in resource-limited device, 7z020, and each PE consumes fewer resources on average because of our efficient design and optimization. Since our design does not run out 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 higher degree of parallelism and partitioned



(a) Arch 1 on 7z020 (b) Arch 1 on 7z100 (c) Arch 2 on 7z020 (d) Arch 2 on 7z100

Fig. 6. Normalized hardware utilization compared to ReBNet. Where results with frame are post-placement results due to routing failed.

TABLE IV
FLOD, FREQUENCY, THROUGHPUT AND POWER USAGE.

		Max *Fold*		M	Freq (MHz)	Thr.put (kFPS)	P_{chip} (W)
Arch1	7z020	96	ours	2	145.31	1513.61	1.353
				3	143.74	1497.29	1.529
		208	ReBNet	2	106.85	256.85	1.052
				3	99.52	159.49	1.994
	7z100	16	ours	2	200.92	12577.77	5.083
				3	193.91	12119.45	4.905
			ReBNet	2	124.30	3884.40	8.121
				3	-	-	-
Arch2	7z020	32768	ours	2	134.28	4.10	1.41
				3	129.03	3.94	1.473
		65536	ReBNet	2	136.44	1.04	1.188
				3	53.70	0.27	1.61
	7z100	8192	ours	2	200.64	24.49	3.868
				3	200.40	24.46	3.961
			ReBNet	2	200.44	12.23	4.774
				3	192.31	7.83	5.916

weights, our design uses more BRAMs in Arch 1. In contrast, in Arch 2, our design uses less 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.

In implementations of 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 6. 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 2,002 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 levels. 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.

V. Conclusion and Future Work

This paper showed that, while our method, isometric residual-binarization, obtains similar accuracy to the baseline work, ReBNet, our hardware design reaches higher degree of parallelism in resource-limited devices and gets higher through-put. In the future, we will apply our method to complex CNN architectures for larger datasets, and implement them on FPGAs. In addition, we will try to use our method in difficult tasks such as object detection and semantic segmentation.

References

[1] 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)*, pages 57–64, 2018.

[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[3] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory F. Diamos, Erich Elsen, David García, et al. Mixed precision training. *CoRR*, abs/1710.03740, 2017.

[4] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew G. Howard, et al. Quantization and training of neural networks for efficient integer-arithmetic-only inference. *CoRR*, abs/1712.05877, 2017.

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

[6] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. *CoRR*, abs/1603.05279, 2016.

[7] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, et al. Finn: A framework for fast, scalable binarized neural network inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, pages 65–74. ACM, 2017.

[8] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[9] François Chollet et al. Keras. `https://keras.io`, 2015.

[10] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[11] Xilinx. Vivado design suite. `https://www.xilinx.com/products/design-tools/vivado.html`.