Binary Complex Neural Network Acceleration on FPGA

(Invited Paper)

*Hongwu Peng[1], *Shanglin Zhou[1], Scott Weitze[2], Jiaxin Li[1], Sahidul Islam[3], Tong Geng[4], Ang Li[4], Wei Zhang[1], Minghu Song[1], Mimi Xie [3], Hang Liu[2], and Caiwen Ding[1]
*These authors contributed equally.
[1]University of Connecticut, Storrs, CT, USA. [2]Stevens Institute of Technology, Hoboken, NJ, USA.
[3]University of Texas at San Antonio, San Antonio, TX, USA. [4]Pacific Northwest National Laboratory, Richland, WA, USA.
[1]{hongwu.peng, shanglin.zhou, jiaxin.3.li}@uconn.edu, [1]wei.zhang.gbs@gmail.com,
[1]{minghu.song, caiwen.ding}@uconn.edu, [2]{sweitze, hliu77}@stevens.edu,
[3]{sahidul.islam, mimi.xie}@utsa.edu, [4]{tong.geng, ang.li}@pnnl.gov

***Abstract*—Being able to learn from complex data with phase information is imperative for many signal processing applications. Today's real-valued deep neural networks (DNNs) have shown efficiency in latent information analysis but fall short when applied to the complex domain. Deep complex networks (DCN) , in contrast, can learn from complex data, but have high computational costs; therefore, they cannot satisfy the instant decision-making requirements of many deployable systems dealing with short observations or short signal bursts. Recent, Binarized Complex Neural Network (BCNN), which integrates DCNs with binarized neural networks (BNN), shows great potential in classifying complex data in real-time. In this paper, we propose a structural pruning based accelerator of BCNN, which is able to provide more than 5000 frames/s inference throughput on edge devices. The high performance comes from both the algorithm and hardware sides. On the algorithm side, we conduct structural pruning to the original BCNN models and obtain 20 × pruning rates with negligible accuracy loss; on the hardware side, we propose a novel 2D convolution operation accelerator for the binary complex neural network. Experimental results show that the proposed design works with over 90% utilization and is able to achieve the inference throughput of 5882 frames/s and 4938 frames/s for complex NIN-Net and ResNet-18 using CIFAR-10 dataset and Alveo U280 Board.**

***Index Terms*—Binarized neural networks, binarized Complex Neural Network, FPGA, high level synthesis, convolutional neural network, surrogate Lagrangian relaxation**

I. Introduction

Due to the growing need for DNN performance on different tasks, today's DNN model has a relatively large model parameter size. For example, ResNet-18 for image classification has a model size of 45 MB, the YOLOv5l of the YOLOv5 family [1] for object detection has a model size as 245 MB, BERT [2] for NLP has 23M embedding weights and 317M neural network weights. For the shallow DNN model, all the weight and intermediate activations can be stored in the FPGA on-chip memory [3] (BRAM and URAM). However, large DNN models' weight is hard to store in the FPGA on-chip memory, and external memory is used to store the weight and activations. [4]. Therefore, the external memory usage for weight and activation will stall the system's performance.

Using Xilinx Alveo U200 Accelerator Cards [5], as an illustration, we often need to access off-chip memory since the on-chip memory capacity is only 35 MB. However, the off-chip memory is inherently slower to access than on-chip memory in terms of 2 aspects: ① The off-chip memory (DDR4) total bandwidth is 77 GB/s, while the total bandwidth on-chip memory is 31 TB/s (400× higher than the off-chip memory); ② The off-chip memory (DDR4) has around 100 ns memory access latency [6], while the on-chip memory can be access at 500 MHz frequency [3] with only one cycle latency (50× lower latency than off-chip memory). Furthermore, The DRAM memory usually has more than 100× higher power consumption [7] than the FPGA on-chip SRAM. The whole system performance is constrained by accessing off-chip memory.

In order to fit the model into the FPGA platforms' on-chip memory, model compression techniques can be used. The model compression techniques help reduce the model size and accelerate the inference of the DNN model with an acceptable accuracy loss. The model compression techniques can be divided into two types: weight pruning and weight quantization. We will leverage both of those two techniques in this paper to achieve a higher model compression rate.

In this paper, we focus on the acceleration of BCNN based NIN-Net and ResNet models. We further propose hardware design to achieve high parallelism and high throughput for the FPGA platform. We implement the proposed techniques on Alveo U280 hardware platforms for comparison of latency and throughput. Experimental results show that the FPGA hardware design enables 5882 frames/s and 6154 frames/s for BCNN based NIN-Net and ResNet models. Our contributions are summarized as follows:

- Basic framework and building blocks for BCNN are given. For the pooling layer, the spectral pooling, average

SLR:拉格朗日松弛法, 参见拉格朗日乘子法

pooling, and max pooling are compared in terms of model performance.

- The Surrogate Lagrangian Relaxation (SLR) weight pruning technique is adopted for BCNN weight pruning, straight-through estimator (STE) is adopted for BCNN weight quantization, the whole model compression framework achieves a high compression ratio with an acceptable accuracy loss.
- The binarized complex convolution kernel design is proposed to enable a high level of hardware parallelism and low pipeline initiation interval.
- The hardware resource scheduling for the BCNN model implementation on FPGA is discussed, and we achieve a high overall hardware throughput.

The organization of the work is as follows. Section II gives the basics of DNN model compression knowledge and the BCNN model. Section III discusses the model structure, SLR-based compression, and STE-based binary quantization. Section IV gives the hardware design for the BCNN based models. Section V gives the BCNN model's training, the hardware implementation result. Section VI gives the overall conclusion for the hardware design and experiments.

II. Related Work

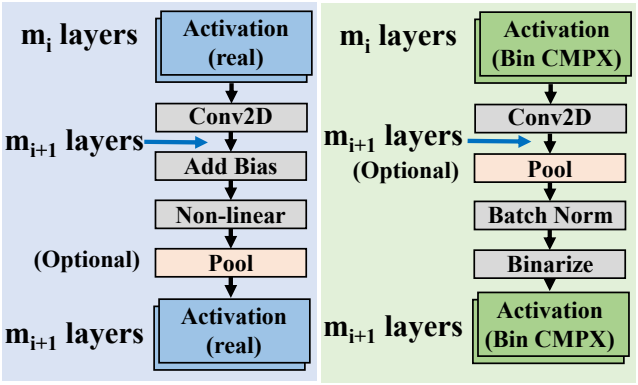
In this section, we will briefly discuss the current works on DNN model compression techniques, BNNs, and complex neural networks.

A. Model Compression

In order to reduce the DNN model size and inference latency, the model compression techniques can be adopted. The main challenge of the SOTA model compression techniques is maintaining the model's accuracy on tasks while improving the model inference speed and throughput on hardware platforms. There are two types of model compression techniques: weight pruning [4] and weight quantization [8].

Two types of pruning methods are widely used for the weight pruning technique: the unstructured pruning method and the structured pruning method [9]. The unstructured pruning technique [10]–[12] is easier to implement on software and has a high compression ratio with low accuracy loss. However, due to its irregular memory access pattern, the unstructured pruning can hardly be accelerated on most of the hardware platforms. The structured pruning technique [10]–[12] constrains the weight matrix to be pruned in a structured and hardware-friendly pattern. For example, block-circulant matrices [8], [13], [14] can be used for weight representation after pruning. The structured pruning-based hardware implementation achieves better performance due to the higher parallelism achievable by regular memory access patterns and reduced computation burden.

Another source of redundancy in the DNN model is the bit representation of the weight. The DNN baseline model usually adopts float32 bits representation for the weight value. In order to compress the bit representation of the data, various works [15], [16] have proposed different DNN model quantization techniques, including fixed bit-length, ternary, and binary weight representations. The truncated length bit representation reduces DNN model size, computation burden on the hardware platform, and memory bandwidth consumption. The fixed bit-length representation of the DNN model parameter can be further classified into equal-distance quantization and power-of-two quantization. The equal-distance quantization is similar to fixed-point number representation, and it reduces the hardware resource utilization while maintaining high accuracy. The power-of-two quantization further improves the hardware efficiency owning to the bit shift-based multiplication. However, the unequally distributed scale of power-of-two quantization leads to a non-negligible DNN model accuracy degradation. In order to improve the model accuracy and maintain hardware efficiency, mixed powers-of-two-based quantization is proposed [17]. The mixed powers-of-two-based quantization features its' combination of primary powers-of-two and a secondary powers-of-two part, and the multiplier requires a 2-bit shifter and one adder.



(a) Original CNN architecture (b) BCNN architecture

Fig. 1: Comparison between original CNN and BCNN

B. Binary Complex Neural Networks

The concept of Binary Neural Network (BNN) originated from the binary weight neural network (BWNN) [18], and the BWNN only quantizes the bit representation of the weight value into the binary value. However, for the FPGA devices with small on-chip memory, the intermediate activations of the BWNN are still too large to be stored in the on-chip SRAM, and external memory is required. The later works [19], [20] proposed BNN and researched the quantization of both activations and weights into a binary representation. Those works illustrated few key concepts to maintain the model accuracy for BNN: straight-through estimator (STE) for gradient descent, batch-normalization after binarized convolution, and keep full precision for both activations and weights at first and last layers. By quantizing both activation and weight, the multiplication is degraded into binary XOR operation and is highly hardware-friendly for FPGA and GPU platforms.

A simply popcnt(xor()) function can be used for binarized convolution layers or fully connected layers.

In order to enhance the model information representation with the same parameter size or even less parameter size, deep, complex neural networks are proposed [21]. The complex neural network has the dedicated complex version of the basic building block: convolution, batch normalization, weight initialization strategy, etc. The deep complex neural networks achieve comparable performance to ordinary DNNs with half model parameters.

The binary complex neural network (BCNN) [22] combines the benefit of both binary neural networks and complex neural networks. The activations and weights of deep complex neural networks are quantized to one bit except the first and last layer. In order to reduce the computation overhead and improve the model accuracy, Li *et al.*. [22] also proposed few new concepts: quadrant binarization for forward and backward propagation, complex Gaussian batch normalization (CGBN), and binary complex weight initialization. Those concepts will be discussed in detail in Section III.

C. Basic of Surrogate Lagrangian Relaxation (SLR)

The surrogate Lagrangian relaxation method (SLR) [23] is an optimization algorithm similar to the alternating direction method of multipliers (ADMM) [24], which breaks optimization problems into several smaller subproblems that can be solved iteratively. However, it also overcame major convergence difficulties of standard Lagrangian relaxation. As the solution of decomposed subproblems is coordinated by updating Lagrangian multipliers, convergence can be proved when the solutions satisfy the "surrogate" optimality condition with a novel step-sizing formula [23].

Gurevin *et al.* [25] was the first work implementing an SLR-based framework on DNN weight pruning. Comparing with the ADMM-based weight pruning method, when under classification task and the same compression rate, SLR can achieve almost 10% point higher accuracy on VGG-16 on CIFAR-10 dataset, and 3% point higher accuracy on ResNet-18 on ImageNet dataset. Under object detection tasks on COCO 2014 benchmark, models pruned with the SLR method can at most have 9% higher accuracy after hard-pruning than the ADMM method under all the YOLO framework. Also, when hardpruning accuracy is checked periodically during training steps, SLR can faster converge and reach the desired accuracy. Almost 3× faster can be achieved during classification tasks for SLR on CIFAR-10 and 2× faster on ImageNet. During objection detection tasks, 3× faster can be achieved on COCO 2014 benchmark. Experiments also show that the SLR-based weight-pruning optimization approach achieves high accuracy even at the hard-pruning stage. This retrain-free propriety reduces the traditional three-stage pruning pipeline to two-stage and reduces the budget of retraining epochs.

III. Training and Compression on BCNN

In this section, we will discuss the BCNN model in detail, which includes model structure, fundamental building blocks and operations for BCNN, weight pruning using SLR, and weight quantization based on quadrant binarization and STE.

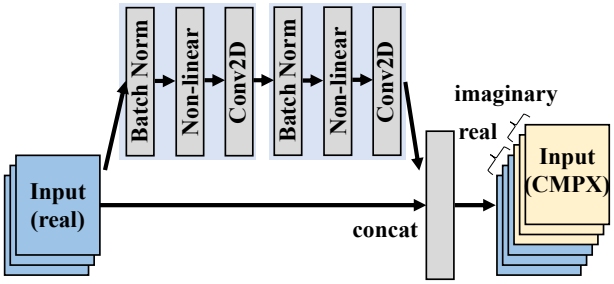


Fig. 2: Complex input generation

A. Structure of BCNN

The comparison between BCNN and the original convolution neural network (CNN) structure is demonstrated in Fig. 1. The mathematical equation for convolution and add bias operation can be found in Eq. 1. As shown in Fig. 1a, the original CNN is composed of convolution layer, add bias, non-linear layer, and pooling layer. For the BCNN, the structure is different from the original CNN, and the structure is shown in Fig. 1b. The pooling layer and batch normalization layer should come after the convolution layer, and the bias can be removed from the network to reduce the computation overhead without accuracy loss. For the BCNN, batch normalization is a mandatory operation for model convergence [26].

$$x_{m_{i+1}} = f(\sum_{c=1}^{m_i} x_c * w_{c,m_{i+1}} + b_{m_{i+1}}) \tag{1}$$

For the image with three channels (RGB) as input, the initial input only contains real part. In order the generate the complex input, a two-layers residual CNN is designed to learn the imaginary part. The network for generating the complex input is shown in Fig. 2.

B. Building Blocks and Operations

The basic building blocks of BCNN are slightly different from that of ordinary DNN. The complex version of the convolution layer, pooling layer, batch normalization, and binarize function will be discussed in this section.

1) Complex Convolution: The binary complex number can be defined as: $x_c = x_r + i \cdot x_i$, where the numbers x_r and x_i belong to $\{+1, -1\}$. A single binary complex number is represented by 2 digits, and has twice memory occupations than that of binary number. Assuming the input activation is $x_c = x_r + i \cdot x_i$, weight is $w_c = w_r + i \cdot w_i$. The dot product between the input activation x_c and weight w_c can be donated as equation format Eq. 2 or matrix format Eq. 3. The mathematically expression of complex convolution operation can be deduced from Eq. 1 and Eq. 3. The convolution can be divided into bit-wise XOR operation, popcnt operation, and fixed-point number add/subtract for the binary complex convolution operation. Comparing to full precision convolution operation on the FPGA platform, only a few LUT resources

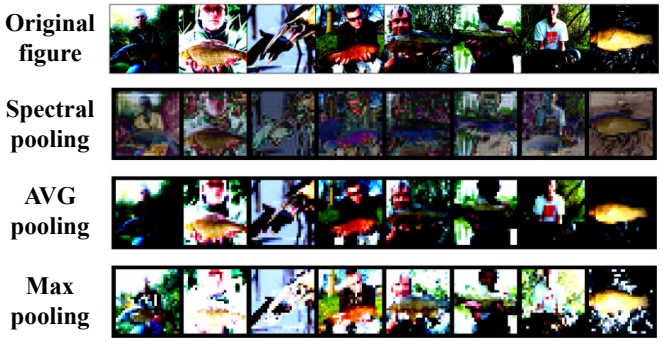


Fig. 3: Comparison of pooling function

will be used to calculate the binary complex convolution operation, and the memory bandwidth requirement is reduced by 32 $\times$.

$$y_c = y_r + i \cdot y_i = (x_r w_r - x_i w_i) + i \cdot (x_r w_i - x_i w_r) \quad (2)$$

$$\begin{bmatrix} y_r \\ y_i \end{bmatrix} = \begin{bmatrix} w_r & -w_i \\ w_i & w_r \end{bmatrix} \cdot \begin{bmatrix} x_r \\ x_i \end{bmatrix} \quad (3)$$

2) Pooling Layer: The pooling layer is optional in the convolutional layer. For a deep CNN, the pooling layer will be used at specific layers to downsample at a rate to reduce the activation sizes. There are two widely used pooling layers in the original CNN: max pooling and average pool. For BCNN, the activations are represented as complex numbers, enabling the possibility of another type of pooling method that has good preservation of information: spectral pooling [27].

The spectral pooling conducted a fast Fourier transform (FFT) over the 2D dimension of activations and truncated the high-frequency component to leave the center low frequency spectral. An inverse FFT (IFFT) is then conducted to transform the spectral information back to the spatial domain.

A comparison of different pooling methods is shown in Fig. 3. The pictures are obtained from the ImageNet dataset [28]. For spectral pooling, the pixel value is scaled back to the range of $(0, 1)$ to ensure visibility. As shown in the figure, the spectral pooling and average pooling preserves more spatial information than the max pooling. The max pooling is more like a 2D whitening function and performs poorly for images with higher brightness.

As for the computation complexity, the FFT operation of spectral pooling has a $O(MN\sqrt{MN})$ computation complexity; average pooling and max pooling both has a $O(MN)$ computation. In order to maintain a good trade-off between the model accuracy and computation complexity, the average pooling is chosen for the hardware design.

3) CGBN: The complex version of batch normalization (CGBN) is an essential operation that helps the BCNN to converge. The original form of batch normalization can be found in Eq. 4, where ${x_i}^{(k)}$ donates the input activation, ${\mu_B}^{(k)}$ donates the mean of the activations in the mini-batch, ${\delta_B}^{(k)}$ donates the variance for the activations in the mini-batch, and ϵ donates a small value to be added to avoid dividing by zero. γ and β are both scaling factors that can be learned during the training step. The activation output is represented as $\widehat{x}_i^{(k)}$.

$$\widehat{x}_i^{(k)} = \gamma \frac{{x_i}^{(k)} - {\mu_B}^{(k)}}{\sqrt{({\delta_B}^{(k)})^2 + \epsilon}} + \beta \quad (4)$$

For the complex neural network, the original form of complex batch normalization can be found in Eq. 5 [21]. V is the 2×2 covariance matrix, and $E[x_c]$ is the mean of complex activations within the mini-batch. γ is a 2×2 matrix and β is a complex number. Both γ and β can be learned during backpropagation.

$$\widehat{x}_c = \gamma \cdot V^{-1/2}(x_c - E[x_c]) + \beta \quad (5)$$

However, as discussed in Li *et al.*. [22], the original form of complex batch normalization requires too much calculation, and the CGBN concept is proposed. The mathematical equation for the CGBN can be found in Eq. 6. Both γ and β are complex numbers scaling factors and can be learned. The CGBN for BCNN has higher accuracy and lower computation complexity so that CGBN will be used for both the software and hardware implementation.

$$\widehat{x}_c = \gamma \cdot \left(\frac{x_r - \mu_r}{\sqrt{2{\delta_r}^2 + \epsilon}} - \frac{x_i - \mu_i}{\sqrt{2{\delta_i}^2 + \epsilon}}\right) + \beta \quad (6)$$

4) Binarization: There are two types of widely used binarization [18]: deterministic binarization and stochastic binarization. The equation for deterministic binarization is given in Eq. 7, activation is binarized to +1 and -1 according to their sign. The equation for stochastic binarization is given in Eq. 8. In the equation, the $\delta(x)$ is a hard clipping function and satisfies $\delta(x) = max(0, min(1, \frac{x+1}{2}))$.

$$x_b = sign(x) = \begin{cases} +1 & \text{if } x \geq 0 \\ -1 & \text{if } x < 0 \end{cases} \quad (7)$$

$$x_b = sign(x) = \begin{cases} +1 & \text{has probability } p = \delta(x) \\ -1 & \text{has probability } 1 - p \end{cases} \quad (8)$$

The stochastic binarization has a better model accuracy performance than the deterministic binarization. However, the implementation of stochastic binarization requires a stochastic number generator and is expensive for hardware design. So the deterministic binarization will be used for both software and hardware experiment.

For the complex number with real and imaginary parts, quadrant binarization is proposed in [22] to conduct the binarization and will be used in this paper. The concept of quadrant binarization is naive and straightforward: the real and imaginary part is binarized individually during the forward propagation.

C. Channel-wise Weight Pruning Using SLR

Consider an N-layer DNN indexed as $i \in 1, ..., N$. The collection of weights at each convolutional layer is denoted by $\mathbf{W}_i$ and the collection of corresponding biases is denoted by $\mathbf{b}_i$, and loss function is denoted by $f(\{\mathbf{W}\}_{i=1}^N, \{\mathbf{b}\}_{i=1}^N)$.

The objective of channel-wise weight pruning can be done by minimizing the loss function and make it subject to constraints that the number of nonzero channels of the weight in each layer is less than or equal to a predefined number. This can be formulated as Eq. 9.

$$\begin{aligned} \underset{\{\mathbf{W}_i\},\{\mathbf{b}_i\}}{\text{minimize}} \quad & f\left(\{\mathbf{W}_i\},\{\mathbf{b}_i\}\right), \\ \text{s.t.} \quad & \mathbf{W}_i \in \mathbf{C}_i, i = 1,\ldots,N, \end{aligned} \tag{9}$$

where $\mathbf{C}_i := \{\mathbf{W}_i|$ the number of channels that being zero in $\mathbf{W}_i$ is greater than $\gamma_i\}$, where $\{\gamma_i\}_{i=1}^N$ is a set of predefined hyper-parameter. This can further be equivalently rewritten in an unconstrained form as Eq. 10.

$$\underset{\{\mathbf{W}_i\},\{\mathbf{b}_i\}}{\text{minimize}} f\left(\{\mathbf{W}_i\},\{\mathbf{b}_i\}\right) + \sum_{i=1}^{N} h_i\left(\mathbf{W}_i\right) \tag{10}$$

The first term represents the nonlinear loss function, and the second represents the non-differentiable penalty term [29] that $h_i(.)$ is the indicator function of $\mathbf{C}_i$ shown in Eq. 11.

$$h_i\left(\mathbf{W}_i\right) = \begin{cases} 0 & \text{if } \mathbf{W}_i \in \mathbf{C}_i,\ i = 1,\ldots,N \\ +\infty & \text{otherwise} \end{cases} \tag{11}$$

The problem cannot be solved only analytically or only using stochastic gradient descent. In this case, duplicate variables are introduced as Eq. 12.

$$\begin{aligned} \underset{\{\mathbf{W}_i\},\{\mathbf{b}_i\}}{\text{minimize}} \quad & f\left(\{\mathbf{W}_i\},\{\mathbf{b}_i\}\right) + \textstyle\sum_{i=1}^{N} h_i\left(\mathbf{Z}_i\right), \\ \text{subject to} \quad & \mathbf{W}_i = \mathbf{Z}_i, \quad i = 1,\ldots,N \end{aligned} \tag{12}$$

To solve the problem, SLR leverages the augmented Lagrangian multipliers and penalizes their violations using quadratic penalties, which can be written as Eq. 13.

$$\begin{aligned} L_\rho\left(\mathbf{W}_i,\mathbf{b}_i,\mathbf{Z}_i,\mathbf{\Lambda}_i\right) &= f\left(\mathbf{W}_i,\mathbf{b}_i\right) + \sum_{i=1}^{N} h_i\left(\mathbf{Z}_i\right) \\ &+ \sum_{i=1}^{N} \text{tr}\left[\mathbf{\Lambda}_i^T\left(\mathbf{W}_i - \mathbf{Z}_i\right)\right] + \sum_{i=1}^{N} \frac{\rho}{2}\left\|\mathbf{W}_i - \mathbf{Z}_i\right\|_F^2 \end{aligned} \tag{13}$$

where Λ_n is the Lagrangian multipliers (dual variables) corresponding to constraints $\mathbf{W}_i = \mathbf{Z}_i$ with the same dimension as $\mathbf{W}_i$. The scalar ρ is a positive number that represents the penalty coefficient, and $\|.\|_F^2$ denotes the Frobenius norm. This can be decomposed into two subproblems and being solved iteratively until convergence.

Step 1: Solve subproblem (loss function) for $\mathbf{W}_i$ using Stochastic Gradient Decent. At iteration k, the "loss function" subproblem is minimizing the Lagrangian function while keeping $\mathbf{Z}_i$ at previously obtained values $\mathbf{Z}_i^{k-1}$ for given values of multipliers Λ_n^k: $\min_{\mathbf{W}_i} L_\rho\left(\mathbf{W}_i,\mathbf{b}_i,\mathbf{Z}_i^{k-1},\mathbf{\Lambda}_i\right)$. This can be solved by using stochastic gradient descent (SGD) [30]. "Surrogate" optimality condition (Eq. 14) needs to be satisfied to ensure multipliers update to right directions.

$$L_\rho\left(\mathbf{W}_i^k,\mathbf{b}_i^k,\mathbf{Z}_i^{k-1},\mathbf{\Lambda}_i^k\right) < L_\rho\left(\mathbf{W}_i^{k-1},\mathbf{b}_i^{k-1},\mathbf{Z}_i^{k-1},\mathbf{\Lambda}_i^k\right) \tag{14}$$

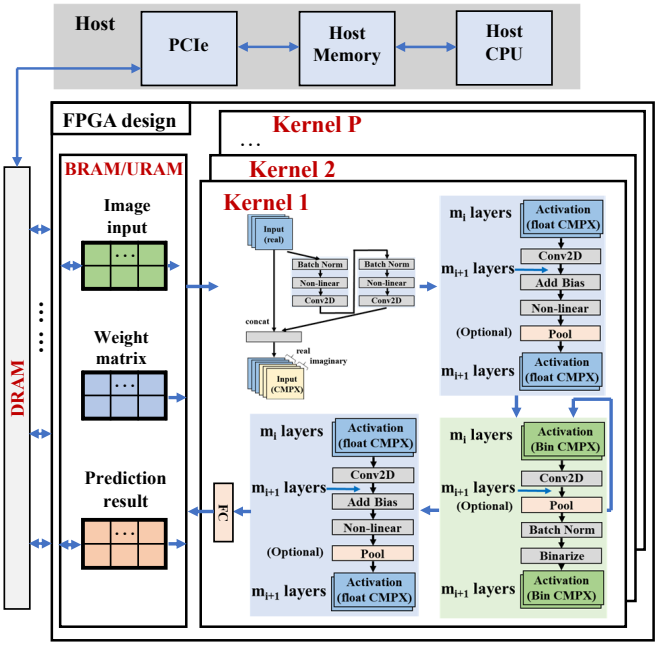


Fig. 4: Hardware design architecture

When Eq. 14 is satisfied, stepsizes and multipliers are updated as Eq. 15.

$$\begin{aligned} s'^k &= \alpha^k \frac{s^{k-1}\|\mathbf{W}^{k-1} - \mathbf{Z}^{k-1}\|}{\|\mathbf{W}^k - \mathbf{Z}^{k-1}\|} \\ \mathbf{\Lambda}_i^{k+1} &:= \mathbf{\Lambda}_i^k + s'^k\left(\mathbf{W}_i^k - \mathbf{Z}_i^{k-1}\right) \end{aligned} \tag{15}$$

Otherwise, previous stepsizes and multipliers are kept.

Step 2: Solve subproblem (Channel-wise pruning) for $\mathbf{Z}_i$ through Pruning using Projections onto Discrete Subspace. The channel-wise pruning subproblem can be written as: $\min_{\mathbf{Z}_i} L_\rho\left(\mathbf{W}_i^k,\mathbf{b}_i^k,\mathbf{Z}_i,\mathbf{\Lambda}_i^{k+1}\right)$. The global optimal of this subproblem can be solved analytically as $h_i(.)$ is the indicator function. For the weight tensor in each convolutional layer, Frobenius norm of each channel of the tensor is calculated and denoted as $\{F\}_c$, where $c = 1, 2, ..., C_i$ and C_i is the number of channels in the tensor. Channels with "larger" $\{F\}_c$ are kept and with "smaller" $\{F\}_c$ are set to zeros following that the number of nonzero channels is less than or equal to γ_i. Second "surrogate" optimality condition (Eq. 16) needs to be satisfied to ensure multipliers update to right directions.

$$L_\rho\left(\mathbf{W}_i^k,\mathbf{b}_i^k,\mathbf{Z}_i^k,\mathbf{\Lambda}_i'^{k+1}\right) < L_\rho\left(\mathbf{W}_i^k,\mathbf{b}_i^k,\mathbf{Z}_i^{k-1},\mathbf{\Lambda}_i'^{k+1}\right) \tag{16}$$

When Eq. 16 is satisfied, stepsizes and multipliers are updated again as Eq. 17.

$$\begin{aligned} s^k &= \alpha^k \frac{s'^k\|\mathbf{W}^{k-1} - \mathbf{Z}^{k-1}\|}{\|\mathbf{W}^k - \mathbf{Z}^k\|} \\ \mathbf{\Lambda}_i^{k+1} &:= \mathbf{\Lambda}_i'^{k+1} + s^k\left(\mathbf{W}_i^k - \mathbf{Z}_i^k\right) \end{aligned} \tag{17}$$

Same as the first step, previous stepsizes and multipliers are kept if the condition is not satisfied. In both steps, stepsize-

setting parameters are formalized as Eq. 18, where M and r are predefined hyper-parameters.

$$\alpha^k = 1 - \frac{1}{M \times k^{(1-\frac{1}{k^r})}}, \quad M > 1,\ 0 < r < 1 \tag{18}$$

D. Weight Quantization

The pruned model from Section III-C will be used for further weight quantization. The binarization is conducted for both activations and weights in the binarized convolution layer, and deterministic binarization function is used. The binarization function (Eq. 7) is non-differentiable at 0, so the direct back-propagation is not feasible for the weight quantizaiton training. Straight-Through-Estimator (STE) is proposed in previous literatures [18], [19] for the back-propagation. The complex version of STE is proposed in [22], and the equation can be found in Eq. 19.

$$\frac{\partial Loss}{\partial w} = \frac{\partial Loss}{\partial w_{rb}} 1_{|w_r|<C_{clip}} + \frac{\partial Loss}{\partial w_{ib}} 1_{|w_i|<C_{clip}} \tag{19}$$

IV. FPGA Hardware Architecture

FPGA is the one of the most popular hardware platform for DNN applications. FPGA platform features it's reconfigurable structure and high level of parallelism for hardware design. With the growing size of the DNN model, the weight matrices and the activations are too large to be stored in the FPGA on-chip memory. However, the aforementioned pruning and weight quantization technologies compressed both activations and weights representation, making it possible for FPGA platform to store all the intermediate results within on-chip memory. In this section, hardware design will be conducted based on Vivado HLS 2020.1, we will present our FPGA hardware structure for the BCNN model.

A. Overall FPGA structure

Our overall FPGA structure for BCNN is shown in Fig. 4. We'll have 2 BCNN model design for the this section: network in network (NIN) [31] based BCNN model and ResNet-18 based BCNN model. Both of those 2 models will be composed of 3 major layers: complex input generation layer (Fig. 2), full precision complex convolutional layer (Fig. 1a), binarized complex convolutional layer (Fig. 1b). A fully connected (FC) layer will be used at last to generate the prediction output.

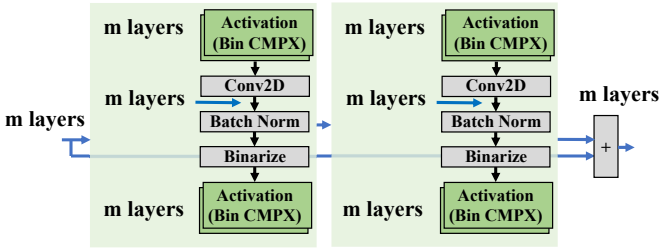


Fig. 5: Residual blocks 1

For the ResNet-18 network, there are 2 types of residual blocks, and both of those residual blocks are binarized block for the BCNN model. The residual block 1 is shown in Fig. 5,

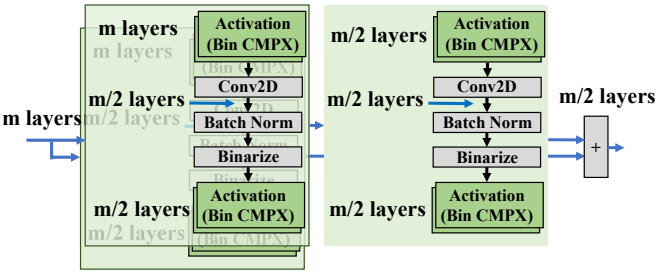


Fig. 6: Residual blocks 2

```
void BC_Conv2D(uint_256 X[32][32],
               uint_256 weight[256],
               int_16   Y[256][32][32],)
{
    Fig_Dim: for(int i = 0; i < 1024; i++)
        CH_out: for(int j = 0; j < 256; j++)
        {
            #pragma HLS PIPELINE II = 1
            #pragma HLS UNROLL factor = p
            uint_256 Mul = X[i/32][i%32] ^ weight[j];
            Y[j][i/32][i%32] = 128 – 2 * pop_cnt_256(Mul);
        }
}
```

Fig. 7: Binarized complex convolutional operation

the input is passed through 2 binarized complex convolutional layers and is added with origin input to get the final output. The residual block 2 is shown in Fig. 6, one of the path has 2 binarized complex convolutional layers and another path has only 1 binarized complex convolutional layers, then outputs of those 2 paths are added together to generate the final output.

B. Hardware Design Details

There are several major building blocks for the hardware design: full precision complex convolutional layer, batch normalization, and pooling layer; binarized complex convolutional layer, batch normalization, and pooling layer. The activation functions used in the models are simple ReLU and Hardtanh functions which have very low computation costs.

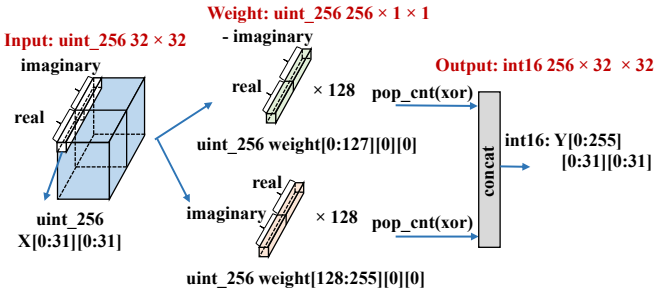


Fig. 8: Hardware mapping for binarized complex convolutional operation

The binarized complex convolution computation follows Eq. 1, Eq. 3, and Eq. 7. The weight matrix for real and imaginary part can be concatenated together to get the final weight matrix input. The concatenated input can directly serve as the general weight matrix input for the convolution. The example HLS code for the binarized convolution with is given in Fig. 7, and the hardware mapping is shown in Fig. 8. In the example design, the convolution kernel size is 1×1, stride is 1, and input and output number of complex channel are both 128. For both the input and output channel, the first 128 channels are the real part and the last 128 channels are imaginary parts. The output channel and input channel are chosen for 2 levels of parallelism. The sparse channel pruning for the convolutional layer can be conducted during the binarization operation to avoid pipeline stall during the convolution operation.

V. Experiment

A. Training of BCNN Models

In this section, we will apply SLR pruning and STE-based quantization for both BCNN based NiN-Net and ResNet-18. For the CIFAR-10 dataset, we'll present the training result for both BCNN based NiN-Net and ResNet-18. For ImageNet dataset, we'll only present the result of BCNN based ResNet-18. We conduct our experiments on Ubuntu 18.04, Python 3.7 and PyTorch v1.6.0 software version. And we are using Nvidia Quadro RTX 6000 GPU with 24 GB GPU memory for the training.

Firstly, in order to finalize the pooling layer function for the final model, spectral pooling, average pruning, and max pooling are compared on the BCNN model. BCNN based NIN-Net is used for demonstration. Three types of pooling are compared in terms of their accuracy achievable, and the result is given in Table I. The average pooling has better performance than the other two types of pooling methods with an acceptable computation complexity. So the average pooling will be used for the BCNN model.

TABLE I: Pooling layer comparison

Network	Pooling layer type	Accuracy
BCNN based NIN-Net	Spectral pooling	87.09%
	Average pruning	87.42%
	Max pooling	86.88%

TABLE II: Model accuracy for CIFAR-10 dataset

Network	Type	Accuracy
NIN-Net	Original	89.13%
	Pruned	84.92%
	Pruned & quantized	83.17%
Complex NIN-Net	Original	89.31%
	Pruned	86.13%
	Pruned & quantized	85.12%
ResNet-18	Original	92.14%
	Pruned	88.51%
	Pruned & quantized	87.67%
Complex ResNet-18	Original	92.51%
	Pruned	90.23%
	Pruned & quantized	89.34%

For the complex version of the model, the number of channel is reduced by half to ensure the same model size for the BCNN model. For the weight pruning, the pruning ratio is set as 0.5 for most of the intermediate layers. Accuracy of four models: NIN-Net, complex NIN-Net, ResNet-18, and complex ResNet-18 on CIFAR-10 dataset on can be found in Table II. For the ImageNet dataset, the BCNN based ResNetE-18 model [22] will be used for the pruning and quantization, and the result is given in Table III. As shown in the table, the complex version of the network will have better performance than the ordinary counterpart. Thus, the complex binary network-based NIN-Net and ResNet-18 will be used for the hardware design evaluation.

TABLE III: Complex ResNetE-18 model accuracy for ImageNet dataset

Network	Type	Top 5 accuracy
Complex ResNet-18	Original	83.46%
	Pruned	78.38%
	Pruned & Quantized	71.69%

B. Hardware Evaluation

The hardware evaluation is conducted on Xilinx SDSoC 2020.1 and Vivado HLS 2020.1.1. Alveo U290 board is used for the demonstration. The CIFAR-10 and dataset will be used as input, each image input has size of $3 \times 32 \times 32$. The BCNN based NIN-Net model and the BCNN based ResNet-18 model will be design to fit the CIFAR-10 dataset image input.

The resource utilization for a single BCNN based NIN-Net inference kernel can be found in Table IV. For a single kernel, the execution latency is 1.53 ms. The maximum resource utilization is bounded by the LUT resources, and nine kernels can be used simultaneously to achieve a higher level of parallelism. The maximum achievable throughput will be 5882 frames/s.

TABLE IV: Resource utilization for BCNN based NIN-Net

Resource	Utilization	Total	percentage (%)
DSP	575	9024	6.37
FF	88845	2607360	3.41
LUT	137387	1303680	10.54

For the BCNN based ResNet-18 model, the resource utilization for a single inference kernel can be found in Table V. The execution latency is 1.62 ms. The resource utilization is also bounded by the LUT resources. In this case, eight kernels can be used simultaneously during the inference step. For the BCNN based ResNet-18, the maximum throughput for the Alveo U280 platform is 4938 frames/s.

TABLE V: Resource utilization for BCNN based ResNet-18

Resource	Utilization	Total	percentage (%)
DSP	465	9024	5.15
FF	112347	2607360	4.31
LUT	161306	1303680	12.37

The cross-platform throughput comparison for BCNN model is conducted. The FPGA platform is Alveo U280, and the GPU platform is a single card Nvidia Quadro RTX 6000 GPU with 24 GB GPU memory. The throughput comparison can be found in Table VI. The proposed FPGA design achieved a 1.51 $\times$ speed up on BCNN based NIN-Net model and 1.58 $\times$ speed up on BCNN based ResNet-18 model.

TABLE VI: Cross-platform throughput comparison

Model	Platform	Throughput (frames/s)
BCNN based NIN-Net	Alveo U280	5882
	RTX 6000	3890
BCNN based ResNet-18	Alveo U280	4938
	RTX 6000	3123

VI. Conclusion

We are the first work to evaluate the BCNN for the FPGA platform. BCNN reduces the memory storage as well as memory bandwidth requirement for the DNN model while maintaining good performance in model accuracy. Further, the resource utilization for BCNN model also reduces since most of the convolution computation will be degraded into XOR and pop_cnt operations. We utilize the HLS tool to design our BCNN models, and the proposed BCNN model hardware design achieves a 5882 frames/s and 6154 frames/s for BCNN based NIN-Net and BCNN based ResNet-18 on the Alveo U280 FPGA platform, which are 1.51 $\times$ and 1.58 $\times$ speed up than the GPU platform.

References

[1] G. Jocher *et al.*, "Yolov5," https://github.com/ultralytics/yolov5, 2021.

[2] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[3] R. Shi, Y. Ding, X. Wei, H. Liu, H. So, and C. Ding, "Ftdl: An fpga-tailored architecture for deep learning systems," in *(FPGA)The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2020, pp. 320–320.

[4] B. Li, S. Pandey, H. Fang, Y. Lyv, J. Li, J. Chen, M. Xie, L. Wan, H. Liu, and C. Ding, "Ftrans: energy-efficient acceleration of transformers using fpga," in *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, 2020, pp. 175–180.

[5] *Alveo U200 and U250 Data Center Accelerator Cards Data Sheet*, Xilinx, May 2020, v1.3.1.

[6] Y.-k. Choi, Y. Chi, W. Qiao, N. Samardzic, and J. Cong, "Hbm connect: High-performance hls interconnect for fpga hbm," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021, pp. 116–126.

[7] M. Horowitz, "Energy table for 45nm process," in *Stanford VLSI wiki*, 2014.

[8] C. Ding, S. Liao, Y. Wang, Z. Li, N. Liu, Y. Zhuo, C. Wang, X. Qian, Y. Bai, G. Yuan *et al.*, "Circnn: accelerating and compressing deep neural networks using block-circulant weight matrices," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017, pp. 395–408.

[9] H. Peng, S. Huang, T. Geng, A. Li, W. Jiang, H. Liu, S. Wang, and C. Ding, "Accelerating transformer-based deep learning models on fpgas using column balanced block pruning," in *2021 22nd International Symposium on Quality Electronic Design (ISQED)*. IEEE, 2021, pp. 142–148.

[10] Y. Cai, H. Li, G. Yuan, W. Niu, Y. Li, X. Tang, B. Ren, and Y. Wang, "Yolobile: Real-time object detection on mobile devices via compression-compilation co-design," *arXiv preprint arXiv:2009.05697*, 2020.

[11] B. Li, Z. Kong, T. Zhang, J. Li, Z. Li, H. Liu, and C. Ding, "Efficient transformer-based large scale language representations using hardware-friendly block structured pruning," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings*, 2020, pp. 3187–3199.

[12] G. Yuan, X. Ma, C. Ding, S. Lin, T. Zhang, Z. S. Jalali, Y. Zhao, L. Jiang, S. Soundarajan, and Y. Wang, "An ultra-efficient memristor-based dnn framework with structured weight pruning and quantization using admm," in *2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. IEEE, 2019, pp. 1–6.

[13] L. Lu, Y. Liang, Q. Xiao, and S. Yan, "Evaluating fast algorithms for convolutional neural networks on fpgas," in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2017, pp. 101–108.

[14] S. Liao, Z. Li, X. Lin, Q. Qiu, Y. Wang, and B. Yuan, "Energy-efficient, high-performance, highly-compressed deep neural network design using block-circulant matrices," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2017, pp. 458–465.

[15] Z. Lin, M. Courbariaux, R. Memisevic, and Y. Bengio, "Neural networks with few multiplications," *arXiv preprint arXiv:1510.03009*, 2015.

[16] J. Wu, C. Leng, Y. Wang, Q. Hu, and J. Cheng, "Quantized convolutional neural networks for mobile devices," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 4820–4828.

[17] C. Ding, S. Wang, N. Liu, K. Xu, Y. Wang, and Y. Liang, "Req-yolo: A resource-aware, efficient quantization framework for object detection on fpgas," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2019, pp. 33–42.

[18] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," *arXiv preprint arXiv:1511.00363*, 2015.

[19] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks," in *Proceedings of the 30th International Conference on Neural Information Processing Systems*, 2016, pp. 4114–4122.

[20] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1," *arXiv preprint arXiv:1602.02830*, 2016.

[21] C. Trabelsi, O. Bilaniuk, Y. Zhang, D. Serdyuk, S. Subramanian, J. Santos, S. Mehri, N. Rostamzadeh, Y. Bengio, and C. Pal, "Deep complex networks. arxiv 2018," *arXiv preprint arXiv:1705.09792*.

[22] Y. Li, T. Geng, A. Li, and H. Yu, "Bcnn: Binary complex neural network," *arXiv preprint arXiv:2104.10044*, 2021.

[23] M. A. Bragin, P. B. Luh, J. H. Yan, N. Yu, and G. A. Stern, "Convergence of the surrogate lagrangian relaxation method," *Journal of Optimization Theory and applications*, vol. 164, no. 1, pp. 173–201, 2015.

[24] S. Boyd, N. Parikh, and E. Chu, *Distributed optimization and statistical learning via the alternating direction method of multipliers*. Now Publishers Inc, 2011.

[25] D. Gurevin, S. Zhou, L. Pepin, B. Li, M. Bragin, C. Ding, and F. Miao, "Enabling retrain-free deep neural network pruning using surrogate lagrangian relaxation," *arXiv preprint arXiv:2012.10079*, 2020.

[26] A. G. Anderson and C. P. Berg, "The high-dimensional geometry of binary neural networks," *arXiv preprint arXiv:1705.07199*, 2017.

[27] O. Rippel, J. Snoek, and R. P. Adams, "Spectral representations for convolutional neural networks," *arXiv preprint arXiv:1506.03767*, 2015.

[28] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.

[29] T. Zhang, S. Ye, K. Zhang, J. Tang, W. Wen, M. Fardad, and Y. Wang, "A systematic dnn weight pruning framework using alternating direction method of multipliers," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 184–199.

[30] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proceedings of COMPSTAT'2010*. Springer, 2010, pp. 177–186.

[31] M. Lin, Q. Chen, and S. Yan, "Network in network," *arXiv preprint arXiv:1312.4400*, 2013.