CQNN: a CGRA-based QNN Framework

Tong Geng	Chunshu Wu	Cheng Tan	Bo Fang	Ang Li	Martin Herbordt
Boston University	*Boston University*	*PNNL*	*PNNL*	*PNNL*	*Boston University*
Boston, USA	Boston, USA	Richland, USA	Richland, USA	Richland, USA	Boston, USA
tgeng@bu.edu	happycwu@bu.edu	cheng.tan@pnnl.gov	bo.fang@pnnl.gov	ang.li@pnnl.gov	herbordt@bu.edu

***Abstract*—Quantized Neural Networks (QNNs) have drawn tremendous attention since – when compared with Convolution Neural Networks (CNNs) – they often dramatically reduce computation, communication, and storage demands with negligible loss in accuracy. To find an optimal balance between performance and accuracy, developers use different data-widths for different layers and channels. Given this large parameter space, it is challenging to design a QNN accelerator which is generally efficient for various and flexible model configurations.**

In this paper we propose CQNN, a novel Coarse-Grained Reconfigurable Architecture-based (CGRA) QNN acceleration framework. CQNN has a large number of basic components for binary functions. By programming CQNN at runtime according to the target QNN model, these basic components are integrated to support QNN functions with any data-width and hyper-parameter requirements. The result is an optimal QNN for the target model. The framework includes compiler, hardware design, simulator, and RTL generator. Experimental results show CQNNs can complete the inference of AlexNet and VGG-16 within 0.13ms and 2.63ms, respectively. We demonstrate the design on an FPGA platform; however, this is only for showcasing the method: the approach does not rely on any FPGA-specific features and can thus be implemented as ASIC as well.

***Index Terms*—QNN, CGRA, Accelerator, Machine Learning**

I. Introduction

Deep Neural Networks (DNNs) have been widely adopted due to their ability to analyze latent information from structured data and to achieve high accuracy through learning [4], [5], [10], [11]. However, for many applications that are high-volume but power-, resource-, or latency-restricted, achieving maximal accuracy may not be crucial [9], [22]. This is especially true for IoT and smart-edge applications. These often require low-latency, real-time inference, as well as low cost, but reaching a certain well-defined level of accuracy is often sufficient.

DNN applications are both computation and communication intensive making it challenging to use them and achieve these requirements. One approach is to squeeze the model by using fewer bits to represent features and parameters. The extreme case is Binary Neural Networks (BNNs), which use only a single bit to represent a feature or parameter [20]. BNNs have been demonstrated to have great potential in cost- and power-restricted domains, but have not been widely adopted in real-world applications due to their significant loss in accuracy. A promising compromise, called Quantized Neural Networks (QNNs), is to use mixed precision, e.g., from 1-bit to 8-bits, and to vary precision across layers and channels. This is done in such a way so as to find the optimal balance between performance and accuracy [9], [16]. QNNs have been found to dramatically reduce computation and communication requirements with negligible loss in accuracy.

To meet requirements of different applications, models are trained with various hyper-parameters including the number of layers, number of channels per layer, and number of bits used at each layer and channel. It is challenging to design an accelerator that can work efficiently with any combination of these hyper-parameters, especially when various numbers of bits used. Most existing architectures are designed for networks with specific configurations. When the model configurations change, the architecture needs to be re-implemented offline. This kind of accelerator guarantees high efficiency, but has poor flexibility. Some other accelerators are designed in more general ways, e.g., to support programming by users for different QNNs. These designs provide good flexibility, but often lose efficiency due to their generalized architectures.

In this paper, we design a novel CGRA-based QNN acceleration framework, CQNN. Taking advantage of the CGRA architecture, CQNN provides both high performance and good flexibility in the processing of mixed-precision QNNs. By programming CQNN at runtime, the architectures of the processing elements of CQNN can be dynamically reconfigured as the best match to the QNN models under processing. The proposed framework includes (1) compiler which generates the instructions to reconfigure the CGRA Network on Chip (NOC) at runtime, (2) binary-component-based CGRA architecture which can be configured to support QNN functions with different hyper-parameters, (3) a cycle-accurate simulator for quick performance evaluation, and (4) an RTL generator for fast implementation. CQNN supports mixed-precision QNNs. In CQNN, all basic units are designed to support binary operations. Multiple binary units are integrated at runtime to support the processing of QNN operations with various data-widths with negligible hardware overheads.

The contributions of this paper are summarized as follows:

- We propose CQNN, a CGRA-based QNN inference acceleration framework. The proposed framework is efficient for QNNs with any model configuration.
- We propose a binary-based CGRA architecture that can be dynamically reconfigured at runtime to support the kernel execution with various data widths efficiently.
- We evaluate our design on Xilinx Ultra-scale+ VCU118 FPGA development board. With the proposed design,

the inference of AlexNet and VGG-16 can be completed within 0.13ms and 2.63ms, respectively.

II. Background

We introduce BNN and QNN models and then discuss existing work on QNN acceleration.

A. BNN and QNN Models

BNNs are an extreme case of QNNs and evolved from conventional CNNs through Binarized Weight Networks (BWN) [2] with the observation that if the weights were binarized to 1 and -1, then expensive floating-point multiplications could be replaced with additions and subtractions. It was next observed that if both weights and inputs were binarized, then even the 32-bit additions and subtractions could be demoted to logical bit operations. With this observation, XNOR-Net was proposed and has become one of the most influential BNNs [3], [20].

The basic structure of a BNN has four essential functions in each CONV/FC layer: XNOR, Population Count (POPCOUNT), Batch Normalization (BN), and Binarization (BIN). Since the weights, inputs, and outputs are binary, multiply-accumulate in traditional DNNs become XNOR and POPCOUNT in BNNs. The output of POPCOUNT is normalized in BN, which is compulsory for high accuracy in BNNs. Batch Normalization (BN) incorporates full-precision floating-point (FP) operations, i.e., two MUL/DIVs and three ADD/SUBs:

$$y_{i,j} = \left(\frac{x_{i,j} - \mathbb{E}[x_{*,j}]}{\sqrt{Var[x_{*,j}] + \epsilon}} \right) \cdot \gamma_j + \beta_j \quad (1)$$

The normalized outputs from BN (i.e., $y_{i,j}$), which are floating point, are binarized in BIN by comparing with 0. Here, BIN acts as the non-linear activation function. Max pooling is sometimes required. Traditionally, pooling is between BN and BIN. It can be shown, however, that this is equivalent to placing pooling after BIN; thus the FP operations in pooling become bit-OR operations, significantly reducing computation complexity.

Researchers have observed various opportunities to further optimize the basic BNN structure. FINN [1], [23] stands out by merging BN and BIN. The original FP-based BN function in Equation 1 and BIN function are integrated as a threshold:

$$Threshold_{i,j,k} = \frac{\mathbb{E}_{*,j,k} + \mathbb{L}_{j,k}}{2} - \frac{\beta_{j,k} \cdot \sqrt{Var[x_{*,j,k}] + \epsilon}}{2 \cdot \gamma_{j,k}} \quad (2)$$

where $\mathbb{L}$ is the length of the vector $K \times \mathrm{K} \times \mathrm{IC}$, K is the filter size, and IC is the number of input channels. Note that $\gamma_{j,k}$ and $\beta_{j,k}$ are learned in training and are fixed in inference. The threshold can, therefore, be obtained after training and kept constant in inference. In this way, the expensive FP operations in BN now become a simple threshold comparison.

The structure of BNNs can be easily extended to QNNs. As QNNs use multiple bits to represent features and parameters, the XNOR and POPCOUNT functions in BNNs become low-precision Multiplication (QMUL) and Accumulation (QACC); the BIN function in BNNs becomes Quantization (QUANT). The functions of QNNs are all based on operations with a limited number of bits, e.g. 1-8. Figure 1 illustrates QNN structure. Similar to BNNs, the QUANT and BN functions of QNNs can also be merged into threshold-based BN (QT-BN). Different from BNNs, T-BN in a q-bit QNN layer contains at least $(2^q - 1)$ thresholds per channel and requires at least q times comparisons to quantize an output feature. All these thresholds can be decided during training. Therefore, they can be treated as constants in the acceleration of inference [1], [13]. As QNNs and BNNs share similar structures, in CQNN, we decompose QNN operations into multiple BNN operations and try to realize QNN hardware modules by integrating binary components.

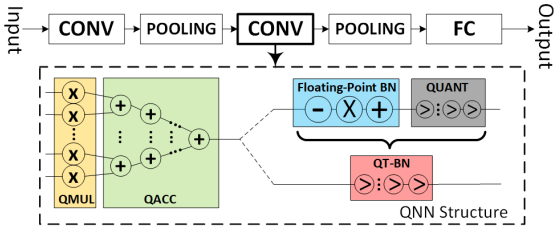

Fig. 1. A simple 3-CONV-1-FC BNN Network structure. It is similar to DNN, except that Activation acts as QUANT. QNN model structure can be further optimized. Floating-point BN and QUANT functions can be merged into QT-BN with multiple thresholds.

B. Existing Work

Quantization of DNNs has been well studied. Besides the already mentioned XNOR-Net [20] and BWN [2], Zhou et al. proposed the DoReFa network which clips activations to improve the utilization of quantization levels [25]. The top-1 accuracy loss of AlexNet is only 6% using 1 bit and 2 bits to represent parameters and features respectively. Miyashita et al. proposed a logarithm-based quantization and demonstrated that using 4-bit parameters and 5-bit features only incurs 1.7% loss in top-5 accuracy for AlexNet [17]. Park et al. proposed a more advanced quantization method demonstrating that ResNet-101 with 5-bit parameters and 6-bit activations has comparable accuracy to the full-precision network [19].

There has also been work on accelerators for QNNs. Wang et al. proposed a hardware-aware automated quantization framework, HAQ [24]. Park et al. proposed an energy-efficient QNN accelerator based on outlier-aware low-precision computation [18]. Umuroglu et al. proposed a flexible heterogeneous streaming architecture for a fast, scalable, and flexible FPGA accelerator for BNNs [23]. Tong et al. proposed an out-of-order architecture, O3BNN, which prunes redundant operations at runtime during inference [7]. These designs provide high performance, but require re-implementation to efficiently support QNNs with various hyper-parameters. Another study that has ideas related to CQNN is Bit Fusion. It does not focus on QNNs, however, and the proposed techniques are not applied to pooling, activation, or BN kernels.

III. BNN Module Integration for QNN

In this section, we discuss how to build QNN modules with multiple binary components. Three QNN modules are introduced: Quantized CONV (Q-CONV), QT-BN, and Quantized POOLING (Q-POOL).

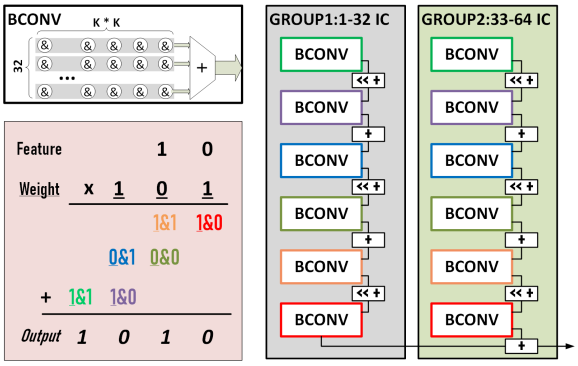

Fig. 2. QCONV for a QNN layer with 2-bit features and 3-bit parameters and the transformation from a 2-bit×3-bit multiplication into 6 bit-add operations.

A. Q-CONV

The Q-CONV modules perform the QMUL and QACC functions. At each cycle, a Q-CONV module performs $K \times K \times NIC$ multiplication operations and accumulates their results in a pipelined manner. The accumulation result is then forwarded to a QT-BN module for quantization and then a complete output feature is calculated. K is the CONV window size and NIC the number of input channels.

In CQNN, each Q-CONV module is composed of multiple bit-level CONV (BCONV) components. Each BCONV is in charge of conducting binary CONV operations of 32 input channels and their POPCOUNTs. At each cycle, one BCONV component performs $K \times K \times 32$ 1-bit×1-bit multiplications (i.e. bit-and) and their accumulations (i.e. POPCOUNT). Figure 2 illustrates the design of a Q-CONV module for a 2-bit×3-bit QNN layer with NIC = 64 and K = 3. This Q-CONV module is made up of 12 BCONV components.

To calculate an output feature, 576 2-bit×3-bit multiplications need to be performed. The calculation of 64 input channels is mapped to 2 groups of BCONV components. Each group handles 32 input channels, i.e 288 multiplications. Each multiplication can be further divided into 6 bit-and operations which are mapped onto 6 BCONV components in the same BCONV group. Each BCONV component therefore calculates 288 bit-and operations in each cycle. Figure 2 illustrates how to transform a 2-bit×3-bit multiplication into 6 bit-and operations and map them to 6 BCONV modules. To calculate all output features, intermediate results of each of these 12 BONV components needs to be reduced. In this design, intra-group result reduction is performed with add-after-shift operations (from the most significant bits to least significant bits); inter-group reduction is realized by summing the intra-group reduction outputs.

B. QT-BN Module

As mentioned in Section 2, QT-BN is performed by comparing the QCONV results to multiple thresholds. For a QNN layer with q-bit features, each output channel has at least $2^q - 1$ thresholds. To calculate each quantized output feature, a QCONV result needs to be compared to q thresholds. In CQNN, each QT-BN is implemented with q T-BN modules. Each T-BN module compares the QCONV result to one threshold and the comparison outcome determines one bit of the QT-BN result. T-BN modules work in a pipelined manner and q modules determine the final value of the quantized feature collaboratively.

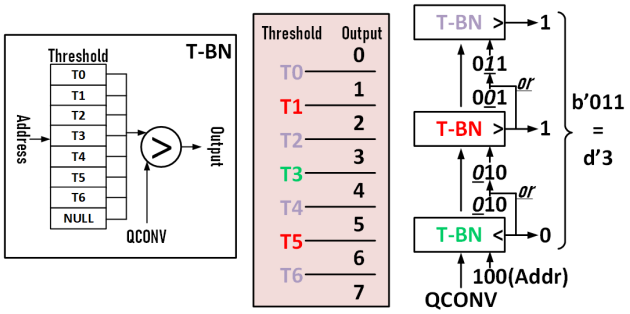

Fig. 3. QT-BN architecture for a QNN layer with 3-bit features.

CQNNs mainly focus on QNNs with 1-bit to 6-bit features and parameters. Figure 3 illustrates the structure of a TQ-BN module for a QNN layer with 3-bit features. Quantized features have 8 possible values, i.e. 0 - 7. The value is determined by comparisons between a QCONV result and 7 thresholds, i.e. T0 - T6 in Figure 3. The first TQ-BN determines the most significant bit of the final output, while the last one decides the least significant bit. Each T-BN has a threshold table for storing the thresholds required to generate the quantized features. Each T-BN has 2 inputs, the address for threshold table access and the QCONV output for comparison, and 1 output, 1-bit of quantized output feature.

C. Q-POOL

Q-POOL (MAX-Pooling) modules are composed of multiple bit-based pooling components (BPOOL) working in a pipelined manner. Each Q-POOL is connected to a QT-BN and consumes the features generated by the QT-BN. For a QNN layer with q-bit features, a Q-POOL module has q BPOOLs. The q BPOOLs and q T-BNs are connected one-to-one. All BPOOLs and T-BNs work in a 2-level pipeline. Different BPOOLs compare different bits of the quantized features and send comparison outcomes to their successors for further comparison if necessary. Figure 4 illustrates the structure of a Q-POOL module for a QNN layer with 3-bit features. Each BPOOL has an output called EN which is used to activate the successive BPOOLs. The EN signal is 0 when a BPOOL finds the incoming bit to be smaller than the one currently stored in its MAX Value buffer. In this case further comparisons and updates are not needed and its successors will neither compare the newly coming data nor update the max value of the bit they are working on. Each BPOOL calculates one bit of the Pooling results.

IV. Design of CQNN Framework

This section introduces the CQNN framework, which includes hardware architecture, compiler, cycle-accurate simulator, and RTL generator. We begin with an overview, the present details, and finally discuss how the compiler works with the architecture for NoC configuration.

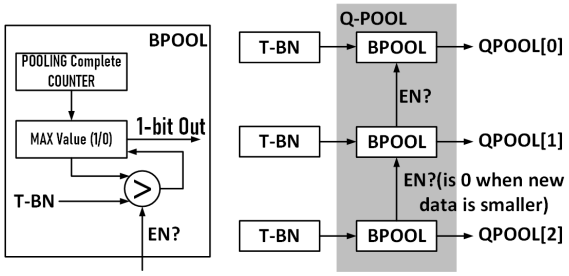

Fig. 4. Q-POOL architecture for a QNN layer with 3-bit features.

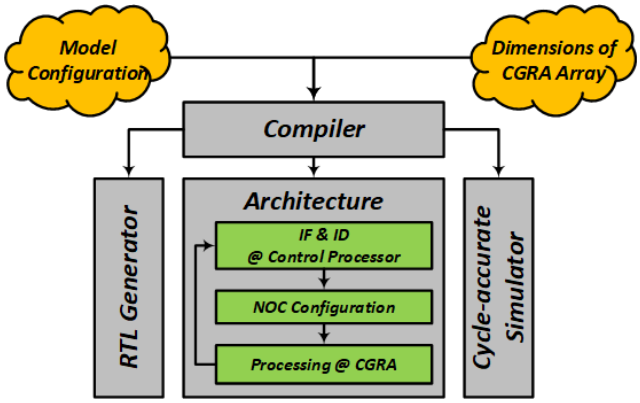

Fig. 5. Framework of CQNN

A. Framework Overview

Figure 5 illustrates the framework of CQNN. The compiler is used to generate instructions to reconfigure CGRA NOC at run time. Instruction generation is based on the model configurations per layer and the hardware constraints, e.g. dimensions of target CGRA arrays. After instructions are generated for an entire QNN model, they are stored in the Control Processor (CP). During processing, the CP fetches and decodes the instructions and generates signals to configure the CGRA network. The RTL generator is used to generate the Verilog-based hardware. A cycle-accurate simulator provides fast and accurate performance evaluation.

B. Architecture

Figure 6 illustrates the architecture of CQNN. CQNN has 2 main modules: CGRA array (CGRA) for inference calculation and Control Processor (CP) for CGRA NOC run-time configuration. We first introduce the architecture.

1) CGRA Array: A CGRA Array consists of a reconfigurable NOC, Parameter Scheduler (PS), Feature Scheduler (FS), and computation components including BCONVs, T-BNs, and BPOOLs. During the processing of a certain layer, FS performs two tasks. First, it prefetches input features from off-chip memory and forwards those features which are going to be consumed in the coming iterations to BCONVs in the expected order. Second, it receives output features calculated in the current iteration from BPOOLs, selects valid outputs, and writes them back to off-chip memory. In parallel, the Switch Reconfiguration Controller (SRC) at CP sends reconfiguration signals of the next layer to PS, WS, and NOC. These signals are cached in these 3 modules with double buffering where

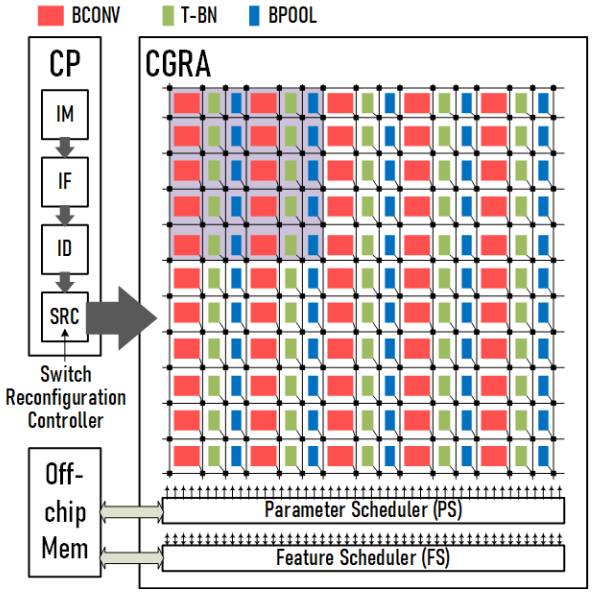

Fig. 6. The overall architecture of CQNN including CP and CGRA array. Black blocks are switches. Each Switch connected to a red block is equipped with an accumulator and a left shift logic. The blocks covered by the gray window can be integrated as an engine for 2-bit×5-bit QNN layers.

they await the completion of the layer being processed. After PS receives the reconfiguration signals, it begins prefetching weights, biases, and thresholds of the next layer and sends them to the BCONVs and T-BN engines.

At the completion of the layer under processing, the NOC is reconfigured based on the signals received previously so that a new configuration that matches the requirements of the coming layer is realized. In this design, in order to reduce the reconfiguration time and routing complexity of the reconfiguration network, the CGRA NOC is reconfigured by column from right-most to left-most. Assuming that the left-most column receives the reconfiguration signals at one column per cycle and that there are c columns in the CGRA, at cycle c, all columns have received their reconfiguration signals and are reconfigured simultaneously. In this design, the number of columns that can be configured at each cycle can be customized. More columns per cycle means faster configuration but a more expensive networks. In this implementation, the SRC configures 1 column per cycle.

As mentioned in Section 3, in this CGRA architecture, all computation components are designed to support binary functions, but, through run-time NOC reconfiguration, can be efficiently and easily integrated into modules with various data-widths. Figure 7 illustrates 3 types of configurations for a motivating QNN engine (a 3-bit×2-bit QNN layer with 32 input channels).[1] Each of these engines is created by integrating 36 binary components which work collaboratively to calculate one complete output feature per cycle. These 3 types of configurations realize the engines with the same functionality but have different shapes. These configurations can be combined to fit on a CGRA with a certain size to

[1]a QNN engine is defined as a group of modules that can produce one complete output feature per cycle

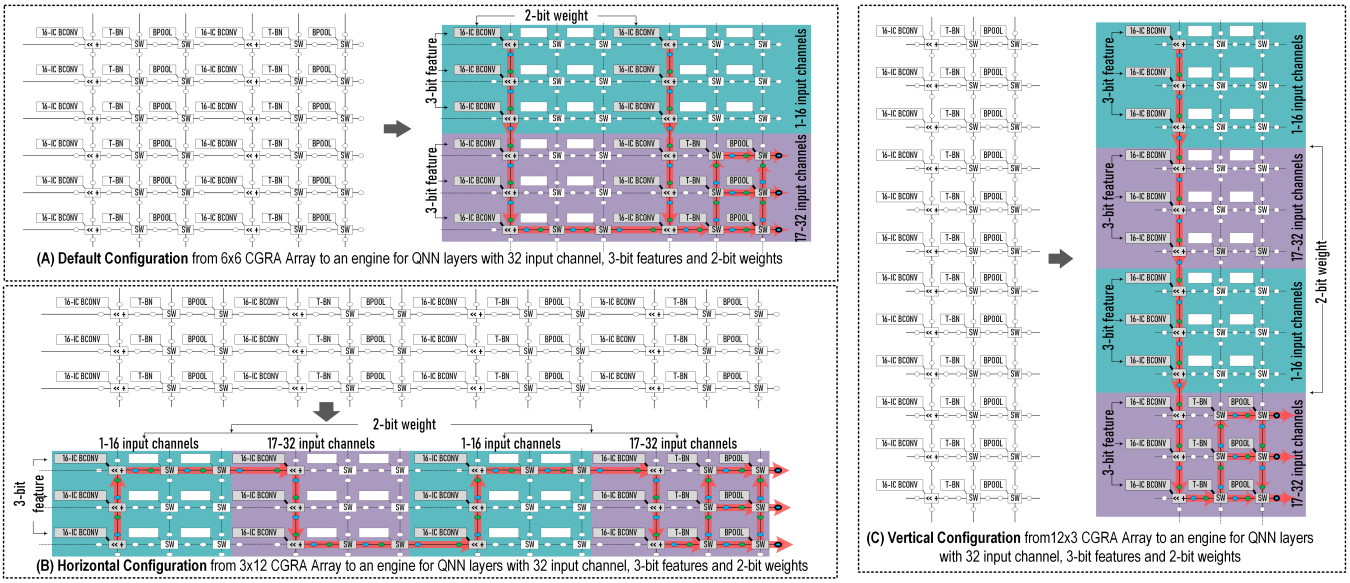

Fig. 7. CGRA Array details and 3 types of integration for a QNN engine for a layer with 32 input channels, 3-bit features, and 2-bit parameters. (A) Default configuration implements a QNN engine by grouping binary components in a 6×6 raw CGRA array. (B) & (C) Implements QNN engines by grouping binary components in a 3×12 & 12×3 raw CGRA arrays. All components are pipelined.

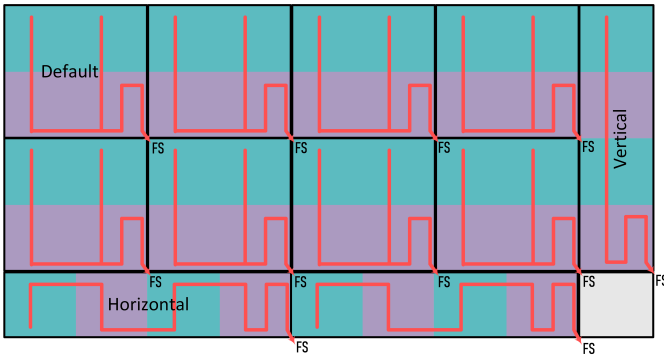

Fig. 8. Engine mapping with 3 types of configuration

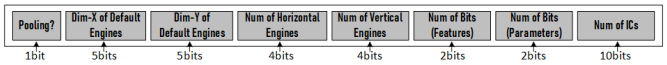

Fig. 9. Instruction structure

maximize the number of engines that can be adopted. As shown in Figure 8, we use the default configuration to map QNN engines onto a CGRA array until there is insufficient space remaining. Horizontal and vertical configurations are then used to fill up the remaining rows and columns. The mapping scheme is generated offline by the compiler. As shown in Figure 7, in a QNN engine, BCONVs are always fully utilized, but only 25% T-BNs and BPOOLs are activated. This overhead is inevitable to support any type of QNN layers dynamically at runtime. The overall overhead, however, is very small as the hardware resources of T-BNs and BPOOLs are only 9.1% and 2.9% of BCONVs.

As for the hardware support for parameter and feature on-chip storage and data forwarding, we use similar designs to the ones used in LP-BNN [6]. Parameters and features at multiple channels are packed and stored in line buffers for easier data movement and reuse.

2) Control Processor: As mentioned above, the CGRA configuration is determined offline by the compiler, and configuration signals are generated by the CP. During compilation a series of instructions is generated based on the model configurations and CGRA dimensions. Each instruction describes the CGRA configuration for one QNN layer. Figure 9 shows the instruction structure. The instructions are stored in Instruction Memory (IM). At the start of processing of a certain layer, Instruction Fetch and Decode (IF & ID) access the instruction for the next layer from IM and send the decoded information to SRC for configuration signal generation. Further details are omitted due to limited space.

C. Compiler

The compiler calculates the maximum numbers of default, horizontal, and vertical QNN engines that can be implemented on the target array; maps these QNN engines onto the array; and generates instructions for all layers. A greedy algorithm is used to determine the mapping strategy. For a particular layer, the compiler first determines the numbers of rows and columns occupied by each default, horizontal, or vertical engine. The compiler then maps as many default engines as possible from the upper left corner to the lower right corner of the CGRA array. The remaining space is filled up with horizontal and vertical engines. At this point the compiler has enough information to generate an instruction for a layer.

V. Evaluation

In this section, we evaluate the performance of CQNN instantiations. We implement a CQNN with a 64×48 CGRA array (64×16 each of BCONVs, T-BNs and BPOOLs) on a Xilinx VCU118 FPGA. We use Vivado Design Suite 2019.1 for design synthesis, implementation, and bit-file generation. The QNN models are implemented in Pytorch and trained with

TABLE I

EXECUTION LATENCY (μs) AND BCONV UTILIZATION OF QNN LAYERS WITH DIFFERENT NUMBERS OF INPUT CHANNELS (NIC) AND DATA-WIDTHS (DW) OF FEATURES AND PARAMETERS. ALL LAYERS HAVE 2×2 MAX-POOLING AND 128 OUTPUT CHANNELS. IMAGE SIZE IS 128 × 128.

Design	CQNN with 64x48 CGRA Array								
Freq	300MHz								
Device	Xilinx VCU118 FPGA (67% LUTs and 40% Flip-Flops)								
DW(F,P)	(1,1)	(2,1)	(2,2)	(3,2)	(3,3)	(4,3)	(4,4)	(5,4)	(5,5)
NIC:32	6.9(100%)	13.8(100%)	27.7(100%)	42.5(98.4%)	63.2(98.4%)	82.9(99.6%)	110.7(100%)	147.3(93.8%)	187.3(92.8%)
NIC:64	13.8(100%)	27.8(100%)	55.0(100%)	83.7(98.4%)	128.3(96.7%)	167.5(98.4%)	220.1(100%)	293.1(93.8%)	372.0(92.8%)
NIC:96	20.8(99.9%)	41.6(99.6%)	83.9(98.4%)	125.6(98.4%)	190.2(98.4%)	254.1(97.6%)	332.9(98.4%)	440.1(93.8%)	587.2(87.9%)
NIC:128	27.7(100%)	54.9(100%)	110.3(100%)	175.3(93.8%)	270.1(91.4%)	334.6(98.4%)	439.2(100%)	585.9(93.8%)	781.9(87.9%)

TABLE II

CROSS-PLATFORM COMPARISON AND EVALUATION: INFERENCE LATENCY IN MS, ENERGY EFFICIENCY IN IMG/KJ. CQNNS ARE COMPARED TO EXISTING FPGA QNN WORK AND A GPU TENSORFLOW-BASED IMPLEMENTATION.

	GPU				FPGA			
Device	P100 [14]		V100 [14]		Stratix-V [15]	VCU108 [8]	ZC706 [7]	
Frequency	1.48GHz		1.53GHz		150MHz	200MHz		
Network	Vgg-like(4,3)	AlexNet(4,3)	Vgg-like(4,3)	AlexNet(4,3)	AlexNet(1,1)	AlexNet(1,1)	Vgg-like(1,1)	VGG-16(1,1)
Latency (ms)	114.65	2183.62	131.09	1164.95	1.16	1.92	0.61	5.63
Energy (Image/KJ)	29.1	1.52	42.1	4.93	3.31E4	2.72E4	1.99E5	2.23E4
	FPGA		***FPGA: Our Design***					
Device	KCU1500 [6]		***CQNNs: VCU118***					
Frequency	200MHz		300MHz					
Network	AlexNet(4,3)	VGG-16(4,3)	Vgg-like(4,3)	AlexNet(4,3)	VGG-16(4,3)	Vgg-like(1,1)	AlexNet(1,1)	VGG-16(1,1)
Latency (ms)	0.27	4.19	0.10	0.13	2.63	0.0081	0.0096	0.19
Energy (Image/KJ)	1.15E5	6.87E3	3.70E5	2.65E5	1.27E4	4.94E6	4.01E6	1.98E5

TABLE III

STRUCTURES OF THE NETWORKS USED TO EVALUATE CQNN.

Network	Network Structure	Dataset	Input Size	Categories
VGG-like	(2x128C3)-MP2-(2x256C3)-MP2-(2x512C3)-MP2-(2x1024FC)	*Cifar-10*	$32 \times 32 \times 3$	10
AlexNet	(64C11/4)-MP3-(192C5)-MP3-(384C3)-(256C3)-(256C3)-MP3-(2x4096FC)	*ImageNet*	$224 \times 224 \times 3$	1000
VGGNet	(2x64C3)-MP2-(2x128C3)-MP2-(3x256C3)-MP2-(3x512C3)-MP2-(3x512C3)-MP2-(2x4096FC)	*ImageNet*	$224 \times 224 \times 3$	1000

a high-end CPU Intel Xeon E5-2680v3 and an NVIDIA Tesla V100 GPU.

A. Performance with different data-widths

We first evaluate the CQNN performance working with QNN layers with different data-widths and numbers of input channels. Table I shows the latency results and hardware utilization of BCONV components. All layers have 128 output channels and 2×2 MAX-Pooling. The size of Input feature maps is 128×128. The CQNN with 64×48 CGRA array consumes 358K (67%) LUTs and 427K Flip-Flops (40%) of the FPGA chip. Results show that the overall utilization of CQNN is over 90% for most of the QNN layers. With a certain CQNN, the execution latency increases linearly with the increase in the QNN layer workload. The end-to-end latency reported in Table I includes instruction fetch, decode, SRC signal generation, feature and parameter read, write from and to off-chip memory, and QNN layer processing. Offline compilation is not included.

B. Cross-platform Comparison

We compare the performance of CQNN to TensorFlow-based QNN implementations on NVIDIA Tesla V100 and P100 GPUs. We use the Vgg-like network [2] of Cifar-10, AlexNet [12], and VGG-16 [21] of ImageNet as benchmarks (Table III). We also compare our results to existing FPGA-based QNN accelerators designed for specific NN models; this shows, in addition to better flexibility, higher performance as well (Table II). Admittedly, we do not claim CQNNs outperform existing designs for QNNs since the improvement of performance may be due to a larger board and higher clock frequency. Still, the comparison confirms that the CQNN design does not sacrifice performance to achieve high flexibility. With CQNN, the inference of Vgg-like, AlexNet, and VGG-16 with 4-bit features and 3-bit parameters takes only 0.10ms, 0.13ms, and 2.63ms, respectively. The inference of binary Vgg-like, AlexNet, and VGG-16 take only 8.1μs, 9.6μs and 190μs, respectively. Board-level power consumption is measured with a power meter.

VI. CONCLUSION

We propose CQNN, a CGRA-based acceleration framework for QNNs. The architecture of CQNN is composed of a programmable control processor, binary components for CONV, BN, Pooling kernels, and reconfigurable NOCs. The control processor reconfigures the NOCs and integrates the binary components at runtime to realize the optimal designs for QNN layers being processed. CQNN has compilation, simulation, and RTL generation support for fast implementation and evaluation. Experimental results show the proposed framework can compute inference of AlexNet and VGG-16 within 0.13ms and 2.63ms, respectively.

REFERENCES

[1] M. Blott, T. B. Preusser, N. J. Fraser, G. Gambardella, K. O'brien, Y. Umuroglu, M. Leeser, and K. Vissers, "Finn-r: An end-to-end deep-learning framework for fast exploration of quantized neural networks," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 11, no. 3, p. 16, 2018.

[2] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," in *Advances in Neural Information Processing Systems*, 2015, pp. 3123–3131.

[3] 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.

[4] T. Geng, T. Wang, A. Li, X. Jin, and M. Herbordt, "A scalable framework for acceleration of cnn training on deeply-pipelined fpga clusters with weight and workload balancing," *arXiv preprint arXiv:1901.01007*, 2019.

[5] T. Geng, T. Wang, A. Sanaullah, C. Yang, R. Xuy, R. Patel, and M. Herbordt, "Fpdeep: Acceleration and load balancing of cnn training on fpga clusters," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2018.

[6] T. Geng, T. Wang, C. Wu, C. Yang, S. L. Song, A. Li, and M. Herbordt, "Lp-bnn: Ultra-low-latency bnn inference with layer parallelism," in *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, vol. 2160. IEEE, 2019, pp. 9–16.

[7] T. Geng, T. Wang, C. Wu, C. Yang, W. Wu, A. Li, and M. C. Herbordt, "O3bnn: an out-of-order architecture for high-performance binarized neural network inference with fine-grained pruning," in *Proceedings of the ACM International Conference on Supercomputing*. ACM, 2019, pp. 461–472.

[8] 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)*. IEEE, 2018, pp. 57–64.

[9] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *The Journal of Machine Learning Research*, vol. 18, no. 1, pp. 6869–6898, 2017.

[10] S. Ji, W. Xu, M. Yang, and K. Yu, "3d convolutional neural networks for human action recognition," *IEEE transactions on pattern analysis and machine intelligence*, vol. 35, no. 1, pp. 221–231, 2013.

[11] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei, "Large-scale video classification with convolutional neural networks," in *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, 2014, pp. 1725–1732.

[12] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[13] M. Lam, Z. Yedidia, C. Banbury, and V. J. Reddi, "Quantized neural network inference with precision batching," *arXiv preprint arXiv:2003.00822*, 2020.

[14] A. Li, T. Geng, T. Wang, M. Herbordt, S. L. Song, and K. Barker, "Bstc: a novel binarized-soft-tensor-core design for accelerating bit-based approximated neural nets," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–30.

[15] S. Liang, S. Yin, L. Liu, W. Luk, and S. Wei, "Fp-bnn: Binarized neural network on fpga," *Neurocomputing*, vol. 275, pp. 1072–1086, 2018.

[16] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh *et al.*, "Mixed precision training," *arXiv preprint arXiv:1710.03740*, 2017.

[17] D. Miyashita, E. H. Lee, and B. Murmann, "Convolutional neural networks using logarithmic data representation," *arXiv preprint arXiv:1603.01025*, 2016.

[18] E. Park, D. Kim, and S. Yoo, "Energy-efficient neural network accelerator based on outlier-aware low-precision computation," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 688–698.

[19] E. Park, S. Yoo, and P. Vajda, "Value-aware quantization for training and inference of neural networks," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 580–595.

[20] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *European Conference on Computer Vision*. Springer, 2016, pp. 525–542.

[21] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[22] W. Tang, G. Hua, and L. Wang, "How to train a compact binary neural network with high accuracy?" in *AAAI*, 2017, pp. 2625–2631.

[23] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "Finn: A framework for fast, scalable binarized neural network inference," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2017, pp. 65–74.

[24] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, "Haq: Hardware-aware automated quantization with mixed precision," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2019, pp. 8612–8620.

[25] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, "Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients," *arXiv preprint arXiv:1606.06160*, 2016.