Quantized Neural Network Synthesis for Direct Logic Circuit Implementation

Yu-Shan Huang, *Student Member, IEEE,* Jie-Hong R. Jiang, *Member, IEEE,* and Alan Mishchenko, *Senior Member, IEEE*

***Abstract*—Hardware acceleration enables neural network (NN) inferencing on edge devices and for high throughput applications. Most approaches use neural processing elements for computation while storing weights in memory blocks. To avoid costly memory access, recent efforts seek direct logic implementation with weights hardwired into the circuit. However, special training strategies are often needed, and they could not maintain accuracy. In contrast, we take a trained and quantized NN as input and synthesize it by Booth encoding and logic sharing, resulting in a hardware accelerator without degrading accuracy. Experiments demonstrate that our method outperforms existing work in area reduction and/or throughput and power efficiency.**

***Index Terms*—AI accelerators, circuit synthesis, field programmable gate arrays, neural network hardware.**

I. Introduction

RECENTLY, machine learning based on deep neural networks (DNNs) demonstrated the ability to address a variety of challenging tasks once believed to be too difficult for computers to handle. The success of DNNs makes them indispensable in many practical applications. As a result, DNN-based models have been widely used in both cloud and edge devices. Implementing DNNs using circuits with low latency, small size, and low power becomes a crucial issue.

Quantization is frequently used to achieve these goals, resulting in quantized neural networks (QNNs). An extreme case of quantization leads to binarized neural networks (BNNs) [1], where both weights and activations are binary. This reduces integer computations to bit operations, leading to even smaller and faster hardware. However, BNNs suffer from a substantial accuracy degradation. In QNNs, floating-point operations can be replaced by fixed-point ones, which tend to improve a hardware implementation. Directly converting a high precision model to a quantized one may degrade its accuracy. Among the existing methods, there are two strategies to obtain quantized models: One is training with special techniques and the other is converting from floating-point DNNs. For those that focus on training, the *fake quantization* proposed in [2] is useful to reduce quantization errors. Other techniques, such as learnable clipping range of activations [3], learnable scale of quantization [4], and learnable regularization [5], are also available. When the focus is on converting high precision models to QNNs, weight equalization [6], [7] is proposed to improve the accuracy. In this work, we assume QNNs are given, and focus on synthesizing them for hardware acceleration.

There have been many efforts focusing on different aspects of hardware implementations for QNNs and BNNs. Neural processing elements (NPEs) are often used. Several frameworks, e.g., [8], [9], are proposed to automatically convert a QNN or BNN to a field programmable gate array (FPGA) for hardware implementation. There are also efforts, e.g., [10], targeting application specific integrated circuits (ASICs) for hardware implementation. NPE-based implementations often require frequent memory accesses and apply time multiplexing for NPE reuse, resulting in high energy consumption, long latency, and low throughput.

To tackle these issues, there are recent endeavors, e.g., [11]–[16], targeting *direct logic implementation* of NNs with all weights hardwired into the circuit. In [11], [12], BNNs are synthesized by sharing common bit-counting circuitry. In [14], ternary-weight NNs are simplified using common subexpression elimination. In [13], activation-binarized NNs are converted to Boolean circuits using don't care optimization. In [15], an effective mapping from the XNOR operations in BNNs to look-up tables (LUTs) used in FPGAs is proposed. In [16], the sparsity of QNNs is exploited to direct map neurons into LUTs. The above methods require special model architectures, such as BNNs, ternary weights, binarized activations, or sparse networks. These special requirements could degrade the attainable accuracy and result in excessive training workloads. In this work, we intend to overcome these weaknesses by taking a trained QNN as input and maintaining the accuracy of the hardware implementation without restricting the QNN.

To achieve our goal, we propose a synthesis flow to convert a QNN to an efficient hardware implementation. In particular, we target edge applications, where latency, throughput and power consumption are important and the deployed NN models are relatively simple. The enabling techniques employed in the flow include Booth encoding and computation sharing for interconnect cost reduction. The synthesis results for models trained on MNIST and CIFAR10 show that the logic sharing reduces the LUT count and the net count by up to 37% and 43%, respectively. Compared with recent efforts, our method is

Manuscript received 4 September 2021; revised 19 January 2022; accepted 26 May 2022. This work was supported in part by the Ministry of Science and Technology of Taiwan under Grant 108-2221-E-002-144-MY3. This article was recommended by Associate Editor J. Cortadella. *(Corresponding author: Jie-Hong R. Jiang.)*

Yu-Shan Huang is with the Graduate Institute of Electronics Engineering, National Taiwan University, Taipei 10617, Taiwan (e-mail: R09943100@ntu.edu.tw).

Jie-Hong R. Jiang is with the Department of Electrical Engineering and the Graduate Institute of Electronics Engineering, National Taiwan University, Taipei 10617, Taiwan (e-mail: jhjiang@ntu.edu.tw).

Alan Mishchenko is with the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, Berkeley, CA 94720, USA (e-mail: alanmi@berkeley.edu).

Digital Object Identifier 10.1109/TCAD.2022.3183547

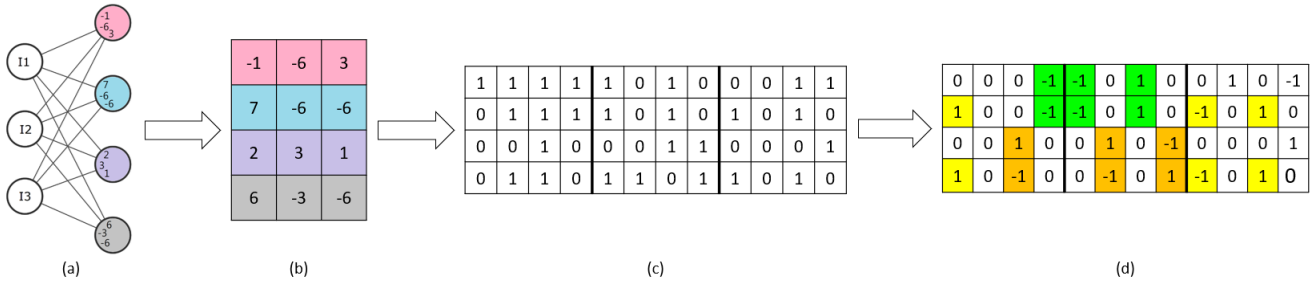

Fig. 1. NN and its matrix representation: (a) neuron layer, (b) its weight tensor, (c) binary encoded matrix, and (d) Booth encoded matrix.

superior or competitive in meeting design constraints in terms of circuit size, throughput, latency, and power efficiency. Our synthesized circuits are suitable for being used in edge devices and various applications.

The rest of this paper is organized as follows. After preliminaries are given in Section II, the proposed logic implementation style and circuit synthesis algorithm are presented in Sections III and IV, respectively. The experimental evaluation is performed in Section V. Finally, conclusions and future work are presented in Section VI.

II. Preliminaries

A. Neural Network Implementation

Neural networks after training are often realized in hardware for inferencing using a parameterized implementation, which is different from a direct logic implementation used in this work. The former stores parameters in memory and loads them into the NPEs during inference. It achieves a small-area-cost design by reusing NPEs repeatedly, but incurs long latency and large power overhead due to memory access. In contrast, the latter makes the weights hardwired into the circuit and eliminates memory access. Apart from pipelining, it requires only combinational logic to implement an NN. Consequently, it can be time-efficient because all the computations of different neurons are performed concurrently and the throughput is not limited by memory access. Although it may incur large area cost, the synthesis algorithm has potential to greatly reduce the overhead.

B. Booth Encoding

In this work, we employ Booth encoding [17], which comes from Booth's multiplication algorithm, to optimize the arithmetic operations of a neural network. The idea is to use $\{-1, 0, 1\}$ instead of $\{0, 1\}$ for each bit of a binary encoded number to reduce the number of nonzero bits when there are consecutive 1's. For example, the binary number 0111 can be rewritten as $100(-1)$, meaning $1000-0001$ effectively, so that the number of nonzero bits is reduced by one. When doing multiplication using the column method, the number of terms we need to sum up equals the number of non-zero bits in the multiplier. For example, $0110\times0111 = 0110+01100+011000$ needs to sum up three terms, since there are three non-zero bits in 0111. If the multiplier is Booth-encoded, the number of terms to be summed up is reduced, and so is the overall amount of computation. For example, 0110×0111 becomes $0110 \times 100(-1) = 0110000 - 0110$, so we only have to sum two terms since there are two nonzero bits in $100(-1)$. The procedure of Booth encoding can be done in time linear in the number of bits of the number [17].

Note that the number of add operations needed for constant multiplication using Booth encoding can be further reduced at the cost of increasing logic levels [18]. We observe that when the number of bits of the multiplier is not large, the number of add operations cannot be reduced much. In particular, when the number of bits of the multiplier is less than 7, the number of add operations cannot be reduced. Therefore, in this work we simply apply Booth encoding without using the reduction technique of [18].

III. Circuit Model

The main computation of neural network inference is multiplication and accumulation. Given a neural network with its weights and activations quantized to fixed-point values, the multiplication can be realized using shift and addition operations. Assuming a b-bit quantization, a weight w can be represented as a 2's complement number $w_{b-1}w_{b-2}\dots w_0$, for $w_i \in \{0, 1\}$ and w_{b-1} being the most significant bit. Given a neuron input I, its multiplication with a weight w is computed by

$$\Sigma_{j=0}^{b-1} w_j \cdot (I \ll j) \tag{1}$$

where $I \ll j$ is the bit string produced by left-shifting I by j bits. This computation can be extended to the case when weights are transformed by Booth encoding. In this case, $w_i \in \{-1, 0, 1\}$.

For a layer of m neurons each with n inputs, its corresponding weight tensor has size $m \times n$.[1] Given quantization into b bits, a fixed-point valued weight tensor of size $m \times n$ can be encoded as a binary valued matrix of size $m \times nb$. The matrix can be further transformed by Booth encoding into a ternary valued matrix of the same size, referred to as the *Booth matrix*, denoted W_B. Fig. 1 shows an example of a 4×3 weight tensor in Fig. 1(b) of a layer of 4 neurons, each with 3 inputs, in Fig. 1(a). With 4-bit quantization, the weight tensor can be

[1]For a fully connected (FC) layer with n inputs and m neurons, the weight tensor has size $m \times n$. For a convolution layer with kernel size $k \times k$, n input channels, and m output channels, the weight tensor has size $m \times n \times k \times k$. If it is reshaped as $m \times nk^2$ and the inputs are properly chosen, the computation of the convolution layer when the filter is at a certain position is the same as that of a FC layer.

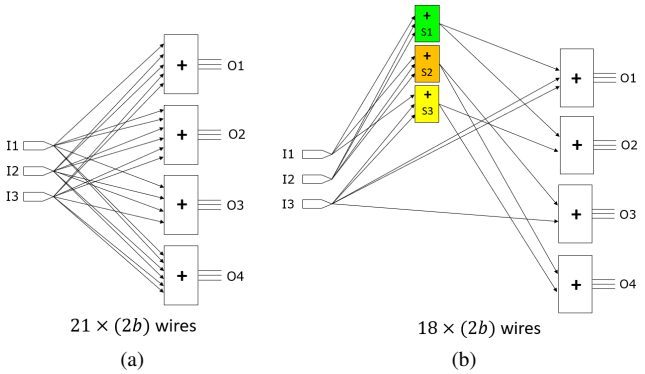

Fig. 2. Circuit model for the Booth matrix in Fig. 1(d): (a) without and (b) with sharing, where the wire counts exclude the adder module outputs $O_1, \ldots, O_4$.

converted to the binary valued matrix shown in Fig. 1(c). With Booth encoding, the matrix can be further transformed into a Booth matrix shown in Fig. 1(d).

The computation of Eq. (1) requires an adder module. Given a Booth matrix, we need a connection between the input and the adder module for each non-zero entry, while no connection is needed for a zero entry. By Eq. (1), if an entry in the Booth matrix is -1, we have to complement I to $-I$ before it enters the adder, and if a non-zero entry is the i^{th} bit of a weight, the input is left-shifted by i bits. Fig. 2(a) shows the corresponding circuit for the Booth matrix in Fig. 1(d) with shift and complement operations omitted.

Different neurons may have some identical additive terms. In this case the summation of these terms can be shared among the neurons. For example, if $X = A + B + C + D$ and $Y = B + C + D + E$, we can first compute $S = B + C + D$ and then X, Y can be computed with $X = S + A$ and $Y = S + E$. When shared terms are extracted, the circuit architecture is of the structure shown in Fig. 2(b). The shared terms are first summed up by sub-adders (the intermediate adders between the inputs and final adders), and the results are then used by the final adders to get the outputs. By sharing the common computation, we can reduce the number of interconnects and circuit area. Our synthesis algorithm to extract logic sharing is detailed in Section IV.

We note that the inputs to an adder are products of two b-bit numbers, i.e., a neuron input value times a weight. To avoid overflow, the number of bits needed for the adder output is $2b + \lceil \log_2 N \rceil$, where N is the number of inputs of a neuron. In practice, we may use training data to gather the statistics of the neuron outputs of a layer and allocate enough bits capable of representing the minimum and maximum values of the outputs of all neurons within the layer. Empirical experience suggests that the number of bits needed is close to $2b$. After an activation function, a weighted summation is clipped and we allocate b bits for its representation. The number of bits for the outputs of the sub-adders is set to that of the final adder.

IV. Synthesis Algorithm

The synthesis flow of circuit implementation for a neural network is sketched in Fig. 3. Given a quantized neural net-

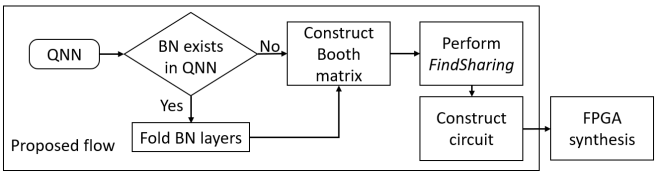

Fig. 3. The synthesis flow.

work[2], the procedure first checks whether the given quantized neural network contains batch normalization (BN) layers. If it does, the *folding*[3] technique proposed in [2] is applied to merge them into preceding fully connected (FC) or convolution layers. Then, the Booth matrices for all layers are constructed as described in Section III. Next, the *FindSharing* algorithm is applied to find computations that can be shared. Accordingly, the corresponding circuit is then built. Finally, the circuit can be further optimized by other logic synthesis tools. In particular, for our target FPGA implementation, the FPGA synthesis tool Vivado is applied. We detail the synthesis algorithm *FindSharing* in the following.

The synthesis algorithm aims at reducing interconnect cost of circuit implementation by sharing common computations. The interconnect cost of a circuit without sharing is defined as follows. For each input connected to the adder, we set the interconnect cost to be $2b$, where b is the number of quantization bits, as discussed in Section III. Hence, the overall cost of a Booth matrix W_B is $cost(W_B) = sum(abs(W_B)) \times 2b$, where abs takes absolute value of each entry and sum sums all entries of the matrix.

For an $m \times nb$ Booth matrix M, let $S_1 \subseteq \{1, \ldots, m\}$ and $S_2 \subseteq \{1, \ldots, nb\}$ be sets of selected row indices and column indices, respectively. Let $M[S_1][S_2]$ denote the sub-matrix of M consisting of rows with their indices in S_1 and columns in S_2. Also, let $M[i][S_2]$, $i \in \{1, \ldots, m\}$, represent a sub-matrix of M that selects the i^{th} row and columns in S_2. A *sharing* in a Booth matrix W_B is defined as a tuple $S(R, C)$, where R is a set of row indices and C column indices such that for each $r_i \in R, W_B[r_1][C] = \pm W_B[r_2][C] = \ldots = \pm W_B[r_{|R|}][C]$ and all entries in $W_B[R][C]$ are non-zero. For a sharing $S(R, C)$, the interconnect cost before sharing is $2b|R||C|$, because each entry in $W_B[R][C]$ is non-zero and indicates that a $2b$-bit connection is needed; the cost after sharing (the computation for the rows in R) is $2b|R| + 2b|C|$, where the terms $2b|R|$ and $2b|C|$ are for the output and input costs, respectively, of the sub-adder. The *gain* of sharing $S(R, C)$ is defined as $2b|R||C| - 2b(|R| + |C|)$, which is the difference between the costs before and after sharing.

[2]We assume an NN is symmetrically quantized. A general quantization from x to q can be expressed by $q = \text{round}(\frac{x}{k} + \text{b})$, where k is a constant scalar and b is a bias constant. A quantization is called symmetric if $b = 0$. Although the symmetric quantization is not the most general quantization scheme, it helps reduce tensor computation while yielding negligible accuracy degradation in NN inference [19]. Therefore we adopt it in our synthesis flow for edge applications.

[3]In [2], fake quantization is applied after folding floating-point weights rather than quantizing the weights before folding. We assume the given NN is trained in the same way, so we can safely fold and then quantize the folded weights to the bit width used during training.

Algorithm 1 FindSharing

Input: An $m \times nb$ Booth matrix W_B
Output: A list of shared terms
1: C := $\{0, \ldots, nb-1\}$;
2: $unSharedCols$:= an empty hash table;
3: **for** $r = 0$ to $m-1$ **do**
4: $unSharedCols$[r] = C;
5: $result$:= [];
6: $sharings$:= *FindMaxPairing*(W_B,$unSharedCols$);
7: **while** *size*($sharings$)> 0 **do**
8: append elements in $sharings$ to $result$;
9: $unSharedCols$:= *refine*($unSharedCols$,$sharings$);
10: $sharings$:= *FindMaxPairing*(W_B,$unSharedCols$);
11: **return** $result$;

Algorithm 2 FindMaxPairing

Input: A Booth matrix W_B, a hash table D that maps row indices to a set consists of column indices
Output: A list of shared terms
1: $G(V, E)$:= *ConstructCompleteGraph*(W_B, D);
2: $pairs$:= *MaxWeightMatching*(V,E);
3: $result$:= [];
4: **foreach** $pair$ in $pairs$ **do**
5: S^{same}, S^{diff} := *PairRows*(vertex 1 of $pair$, vertex 2 of $pair$);
6: **if** $gain(S^{same}) > 0$ **then**
7: append S^{same} to $result$;
8: **if** $gain(S^{diff}) > 0$ **then**
9: append S^{diff} to $result$;
10: **return** $result$;

The algorithm *FindSharing* to find logic sharing in a Booth matrix W_B is sketched in Algorithm 1. The algorithm first initializes the search range to the whole Booth matrix in lines 1-4. The search range is stored in a hash table, which maps a row index to a set of column indices. Then, *FindMaxPairing*, sketched in Algorithm 2, is called in line 6 to find shared terms, which are then appended to the result. After some shared terms have been found, *FindMaxPairing* should exclude those columns that have been included in the resulting terms. Therefore, we trim the search range in line 9 by removing the indices of the used columns from $unSharedCols$. Finding new shared terms and refining the search range are performed repeatedly, as the while-loop in line 7 does, until no new shared terms are found.

To find new shared terms, Algorithm *FindMaxPairing* first builds a complete graph $G(V, E)$ in line 1. The vertices V correspond to the row indices of W_B and the weight of each edge in E is the sum of the gains of the sharings computed by Algorithm *PairRows*, sketched in Algorithm 3, which pairs the two rows connected by the edge. If the gain of a sharing is smaller than 0, it will not be added into the weight of the corresponding edge. We run the maximum weight matching algorithm on the graph to find row pairs and convert them to sharings. The sharings are added to $result$ if the gain is grater than 0.

Algorithm 3 PairRows

Input: Two rows, $r1$ and $r2$, of a Booth matrix with the same selected columns and their indices, $r1Ind$ and $r2Ind$, in the Booth matrix
Output: Two sharings S^{same} and S^{diff}
1: $r1Pos$:= $\{i \mid r1[i] = 1\}$;
2: $r1Neg$:= $\{i \mid r1[i] = -1\}$;
3: $r2Pos$:= $\{i \mid r2[i] = 1\}$;
4: $r2Neg$:= $\{i \mid r2[i] = -1\}$;
5: $colsSame$:= $(r1Pos \cap r2Pos) \cup (r1Neg \cap r2Neg)$;
6: $colsDiff$:= $(r1Pos \cap r2Neg) \cup (r1Neg \cap r2Pos)$;
7: S^{same} := $S(\{r1Ind, r2Ind\}, colsSame)$;
8: S^{diff} := $S(\{r1Ind, r2Ind\}, colsDiff)$;
9: **return** S^{same}, S^{diff};

PairRows converts a given row pair to two sharings, S^{same} and S^{diff}. It first creates two index sets for each row in lines 1-4. One set contains the column indices having corresponding 1-entries, and the other contains those having -1-entries. There are two cases when the summation terms are shared. One is when the entries are the same for the two rows, and the other is when the entries of one row are the opposites of the other. In the former case, we construct $colsSame$ by finding the columns that are both 1 or -1 for the two rows in line 5. In the latter case, we construct $colsDiff$ by finding the columns that have opposite signs between the two rows in line 6. Then, corresponding sharings, S^{same} and S^{diff}, are constructed.

To analyze the time complexity of *FindSharing*, we first consider how many times the while-loop in line 7 is executed. Each time, two rows are paired and the refinement in line 9 excludes their common columns from searching. Once two rows are paired, they cannot produce more shared terms after that. Assuming that W_B is of size $m \times nb$, there are $\frac{m(m-1)}{2}$ different possible pairs and $\frac{m}{2}$ such pairs are considered in each iteration, so we know that the while-loop runs for $O(\frac{m(m-1)}{2} \div \frac{m}{2}) = O(m)$ times. In the while-loop, the main operation is *FindMaxPairing*, which does *MaxWeightMatching* and the for-loop in line 4 of *FindMaxPairing*. The *MaxWeightMatching* is done using the python package, *networkx*, running an algorithm [20] with time complexity $O(|V|^3)$. Since the vertices correspond to the row indices of W_B, we have $|V| = m$, and thus the time complexity of *MaxWeightMatching* is $O(m^3)$. The for-loop in line 4 iterates over all pairs and calls *PairRows*. It will run $O(\frac{m}{2})$ times. *PairRows* basically does set operations on rows, and the time complexity is $O(nb)$. As a result, the time complexity of the for-loop is $O(mnb)$. Assuming m and nb are in the same order, which is generally true, we have $O(mnb + m^3) = O(m^3)$. Finally, the overall time complexity of *FindSharing* is $O(m^4)$. We note that the complexity in practical can be much lower than the theoretical upper bound as to be shown in Section V-B.

Consider the Booth matrix in Fig. 1(d) as an example. *FindSharing* first initializes $unSharedCols$ to r1:{1, 2, …, 12}, r2:{1, 2, …, 12}, r3:{1, 2, …, 12}, r4:{1, 2, …, 12}. The first call to *MaxWeightMatching* yields two pairs, (r1, r2) and (r3, r4), which give us the green and orange sharings in Fig. 1(d). Then, $unSharedCols$ are refined to r1:{1, 2, 3, 6, 8,

9, 10, 11, 12}, r2:{1, 2, 3, 6, 8, 9, 10, 11, 12}, r3:{1, 2, 4, 5, 7, 9, 10, 11, 12}, r4:{1, 2, 4, 5, 7, 9, 10, 11, 12}. The second call yields a pair, (r2, r4), corresponding to the yellow sharing in Fig. 1(d). The $unSharedCols$ are refined to r1:{1, 2, 3, 6, 8, 9, 10, 11, 12}, r2:{2, 3, 6, 8, 10, 12}, r3:{1, 2, 4, 5, 7, 9, 10, 11, 12}, r4:{2, 4, 5, 7, 10, 12}. Finally, no more sharing is found, and the while-loop terminates, resulting in three shared terms. The result of *FindSharing* is shown in Fig. 1(d). Accordingly, the circuit of Fig. 2(a) with outputs

$$O_1 = -I_1 - (I_2 \ll 3) + (I_2 \ll 1) + (I_3 \ll 2) - I_3$$
$$O_2 = (I_1 \ll 3) - I_1 - (I_2 \ll 3) + (I_2 \ll 1) - (I_3 \ll 3) + (I_3 \ll 1)$$
$$O_3 = (I_1 \ll 1) + (I_2 \ll 2) - I_2 + I_3$$
$$O_4 = (I_1 \ll 3) - (I_1 \ll 1) - (I_2 \ll 2) + I_2 - (I_3 \ll 3) + (I_3 \ll 1)$$

can be transformed into the one of Fig. 2(b) with $O_1 = S_1 + (I_3 \ll 2) - I_3$, $O_2 = S_1 + S_3$, $O_3 = S_2 + I_3$, $O_4 = -S_2 + S_3$, where

$$S_1 = -I_1 - (I_2 \ll 3) + (I_2 \ll 1)$$
$$S_2 = (I_1 \ll 1) + (I_2 \ll 2) - I_2$$
$$S_3 = (I_1 \ll 3) - (I_3 \ll 3) + (I_3 \ll 1).$$

V. Experimental Results

In our experiments, the neural networks were trained using Python3 and PyTorch. To get the quantized models, we trained them with the technique proposed in [2], but restricting the scale to be power-of-two integers for a more efficient representation of the quantization range. The quantized models were then processed by our synthesis flow presented in Section IV to generate Verilog circuits on a Windows 10 laptop with Intel i7-1065G7 CPU and 16GB RAM. Finally the circuits were synthesized with Vivado 2020.1 on a computer running Ubuntu 18.04.4 with Intel i7-8700 CPU and 32GB RAM for the intended FPGA implementation.

As an independent check of the accuracy of the quantized models, we implemented inference in software using integer computations and truncations that are identical to the hardware implementations. The accuracy of the software-based inference is the same as that of the circuit-based inference as they perform identical computations. The equivalence between the software model and circuit implementation were checked by simulation in our experiments.

The experiments were conducted to answer two questions: 1) How do Booth encoding and logic sharing improve the FPGA synthesis quality in terms of LUT and net reductions? 2) How does our synthesis method perform compared to other existing efforts targeting FPGA implementation of NNs?

A. Effect of Booth Encoding and Logic Sharing

The first experimental evaluation was performed on neural networks for MNIST and CIFAR10 datasets. For the MNIST dataset, we trained a model composed of three FC layers (with 48, 48, and 10 neurons in order). The activation function is *relu6*, which is defined as $f(x) = \max(0, \min(x, 6))$. This activation function is often used to prepare quantized models, since it clips the weighted summation to the range [0,6], which avoids high-magnitude activations, and reduces the quantization error. We quantize both weights and activations of the model to 4 bits. The testing accuracy of the quantized model is about 97.3%. The testing accuracy for the floating-point model with the same architecture is about 98.4%. For the CIFAR10 dataset, we trained a model composed of five convolution layers and three FC layers. Each layer is followed by a BN layer, except for the output layer. Each BN layer was folded into its previous layer after training. All the convolution layers used kernels of size 3×3, and the padding and stride were set to 1. The numbers of input and output channels for the first convolution layer are 3 and 32, respectively. For all other convolution layers, the numbers of input and output channels are both 32. We used max-pooling with kernel size 2×2 and stride 2 after the 2^{nd}, 4^{th}, and 5^{th} convolution layers. The 1^{st} and 2^{nd} FC layers have 16 neurons each, and the output layer has 10 neurons. The activation function is *relu6*. Both weights and activations of the model were quantized to 6 bits. The testing accuracy of the quantized model is 82.5%, whereas that of the floating-point model with the same architecture is about 83.1%.

TABLE I
Synthesis results under different optimization settings.

dataset	setting	cost (ratio)	LUT (ratio)	net (ratio)
MNIST	plain	-	81.6k (1.00)	276.7k (1.00)
	baseline	255.6k (1.00)	192.5k (2.36)	627.5k (2.27)
	opt-l	148.9k (0.58)	121.0k (1.48)	360.2k (1.30)
	opt-b	99.8k (0.39)	80.0k (0.98)	249.8k (0.90)
	opt-b+l	67.6k (0.26)	63.0k (0.77)	176.9k (0.64)
CIFAR10	plain	-	318.0k (1.00)	1047.3k (1.00)
	baseline	1411.5k (1.00)	616.4k (1.93)	2129.4k (2.03)
	opt-l	776.4k (0.55)	360.8k (1.13)	1174.4k (1.12)
	opt-b	620.0k (0.44)	297.0k (0.93)	963.6k (0.92)
	opt-b+l	387.8k (0.27)	200.9k (0.63)	593.1k (0.57)

The synthesis results under different optimization settings are shown in Table I, where setting "plain" indicates the synthesis flow using the default multiplication of Verilog code and without applying Booth encoding and logic sharing, "baseline" indicates multiplication using shift and addition without applying Booth encoding and logic sharing, "opt-l" indicates the application of logic sharing *FindSharing* (on the baseline setting), "opt-b" indicates the application of Booth encoding, and "opt-b+l" indicates the application of both Booth encoding and logic sharing. Note that the above settings only differ in how the Verilog circuits are generated. The circuits of different settings will be synthesized by Vivado under the same configuration. Note also that in this table the reported hardware usages of the CIFAR10 model only count one implementation copy for each filter in a convolution layer assuming that the filter implementation is reused for processing different inputs through time-multiplexing. The intermediate results can be stored in registers to avoid expensive memory access. For the CIFAR10 model reported in Table I, the registers needed for the five convolution layers are 196608, 49152, 49152, 12288, and 3072, respectively. Typical FPGAs have twice more flip-flop resources than the LUT resources [21], [22], and this model needs 310272 registers in total, which is about 1.54 times the LUT count. Therefore, this amount of registers is

TABLE II
RUNTIME OF DIFFERENT SYNTHESIS STEPS.

dataset	layer	W_B gen (s)	FindSharing (s)	Vivado (s)
MNIST	fc1	5.33	0.01	142.0
	fc2	0.42	0.51	60.0
	fc3	0.11	0.03	35.0
CIFAR10	conv1	0.23	0.13	35.0
	conv2∼5	2.12∼2.86	0.51∼0.88	102.0∼117.0
	fc1	2.19	0.28	98.0
	fc2	0.11	0.03	27.0
	fc3	0.09	0.02	28.0

not a critical factor limiting the FPGA implementation.

From the table, we observed that the hardware cost of "baseline" is higher than that of "plain." Even if logic sharing is further applied, the hardware cost can be reduced substantially, but the cost of "opt-l" is still greater than that of "plain." The results suggest that Vivado can synthesize the multiplication to LUTs more efficiently when the Verilog code uses the native multiplication operator. It may be because the tool recognizes multiplication and enables dedicated optimization, which might not be performed when shift and addition circuits are used for multiplication. When Booth encoding is applied solely, the hardware usage of "opt-b" is slightly lower than to that of "plain." Ultimately, the "opt-b+l" setting by combining both Booth encoding and logic sharing effectively reduces LUT and net counts by 23% and 36%, respectively, for the MNIST model, and 37% and 43%, respectively, for the CIFAR10 model. We observed that the larger the size of a Booth matrix is, the more the cost our algorithm can reduce. We also note that the trend of net reduction is similar to that of cost reduction, suggesting that our modeling of the interconnect cost is reasonable.

The runtime information for different synthesis steps is reported in Table II, where "W_B gen" indicates runtime for Booth matrix generation. Generating all the Verilog circuits took about 0.42 seconds for the MNIST model and 1.83 seconds for the CIFAR10 model. From Table II, we note that the time needed for our optimization is much less than that for Vivado synthesis.

Our flow optimizes the NN circuits at the arithmetic operation level, while existing logic optimization tools, such as ABC, optimize circuits at the logic level. Our method tends to be orthogonal and complementary to other logic synthesis tools. To justify the claim, we compare the circuit sizes for the following four settings: 1) neither our flow nor ABC optimization is applied, 2) only our flow is applied, 3) only ABC optimization is applied, 4) both our flow and ABC optimization are applied, to see how each optimization technique affects the circuit size reduction. We use a CNN model targeting the CIFAR10 dataset to perform the experiment. The model consists of two convolution and two fully connected layers. We constrain the weights of the two convolution layers and the first FC layer to be power-of-two numbers and quantize the weights of the second FC layer to 4 bits. Our flow can be directly applied on models with power-of-two weights because it is only a special case of quantization. We use the "baseline" and "opt-b+l" settings to generate the Verilog circuits, which are then synthesized to the aiger format by Yosys [23]. Note that we use the "baseline" rather than the "plain" setting since most of the weights in the model are power-of-two numbers and thus multiplications become shift operations. The aiger circuits are fed into ABC to get the AIG gate counts. For the settings with ABC optimization, we perform iterations of `resyn`, `resyn2`, `resyn2a`, `resyn3`, `resyn2rs`, `dc2`, `ifraig` to optimize the circuit. The experimental results are shown in Table III. We can observe that our flow alone can reduce the gate count by about 25.5% and the ABC optimization alone can reduce the gate count by about 30.4%. Combining the two optimization methods can reduce the gate count by about 52.7%. The results suggest that our method and ABC optimizations can take effect independently regardless of whether or not the other is applied.

TABLE III
AIG GATE COUNTS UNDER DIFFERENT SETTINGS.

setting	without ABC optimization	with ABC optimization
baseline	2080547	1448930
opt-b+l	1550164	983548

B. Scalability Study

We emphasize that our synthesis flow targets direct logic circuit implementation of quantized neural network for edge applications. Due to the limited hardware resources in edge devices, we do not expect large models for synthesis. Nevertheless, we investigate how the *FindSharing* algorithm scales for large models in the following experiments. To study it, we run *FindSharing* on two PyTorch pretrained models, ResNet18 and ResNet101 [24], for the ImageNet dataset. We directly convert them to QNNs with 8-bit symmetric quantization.

The results are shown in Table IV, where Columns 4 and 5 list the runtime of Booth matrix generation and the *FindSharing* algorithm, respectively, Columns 6 and 7 list the interconnect cost before and after applying *FindSharing*, respectively, and Column 8 shows the percentage of cost reduction. In the table, the results for the layers with the same size are averaged and shown once. In Column 2, we follow the naming of PyTorch to denote the layers; prefix "li" stands for layer i and suffix "ds" stands for "downsample." For ResNet18, "l1convs" stands for all convolution layers in layer 1, and "l$i$convs," for $i = 2, 3, 4$, stands for all convolution layers except the first convolution layer in layer i. For ResNet101, "lib0conv1" stands for the first convolution layer in block 0 and "l$i$conv1s" stands for the first convolution layers in the blocks other than block 0 of layer i. On the other hand, "l$i$conv$j$s" for $j = 2, 3$, stands for all the jth convolution layers for all blocks including block 0 of layer i.

The results show that *FindSharing* spends reasonable time for most layers and can effectively reduce the interconnect cost. The empirical results suggest that time complexity is smaller than that derived in Section IV. For an $m \times n$ matrix, we observed that the number of iterations of the while-loop in lines 7-10 of Algorithm 1 is only sub-linear in m. Generally speaking, it depends on the matrix content but not sensitive to different matrix sizes; also, the greater the number of columns

TABLE IV
RUNTIME AND COST STATISTICS FOR RESNET18 AND RESNET101.

model	layer	size	W_B gen (s)	FindSharing (s)	ori cost	new cost	saving
ResNet18	conv1	64×147	1.02	1.20	174k	113k	35.3%
	l1convs	64×576	4.14	2.99	870k	506k	41.8%
	l2conv1	128×576	8.45	16.00	1930k	1109k	42.5%
	l2convs	128×1152	16.57	26.76	3382k	1893k	44.0%
	l2ds	128×64	0.92	5.29	180k	124k	31.1%
	l3conv1	256×1152	34.45	165.92	8137k	4488k	44.8%
	l3convs	256×2304	67.70	268.75	14335k	7754k	45.9%
	l3ds	256×128	3.72	44.17	747k	476k	36.2%
	l4conv1	512×2304	138.28	1596.42	314329k	16880k	46.3%
	l4convs	512×4608	275.35	2427.70	61126k	32227k	47.3%
	l4ds	512×256	15.24	548.52	3347k	1991k	40.5%
	fc	1000×512	62.64	6033.03	17387k	9720k	44.1%
ResNet101	conv1	64×147	0.95	1.54	396k	218k	44.9%
	l1b0conv1	64×64	0.41	0.87	144k	82k	42.3%
	l1conv1s	64×256	1.74	1.85	680k	368k	45.9%
	l1conv2s	64×576	3.71	2.80	1516k	804k	47.0%
	l1conv3s	256×64	1.48	27.50	431k	243k	43.7%
	l1ds	256×64	1.51	28.78	463k	260k	43.8%
	l2b0conv1	128×256	3.80	12.39	1938k	10323k	46.7%
	l2conv1s	128×512	7.53	18.81	3649k	1915k	47.5%
	l2conv2s	128×1152	16.60	30.96	8552k	4432k	48.2%
	l2conv3s	512×128	6.20	332.22	2188k	1188k	45.7%
	l2ds	512×256	13.36	536.25	5565k	2958k	46.8%
	l3b0conv1	256×512	15.54	98.61	7135k	3750k	47.4%
	l3conv1s	256×1024	31.36	155.33	15690k	8116k	48.3%
	l3conv2s	256×2304	71.67	266.03	36942k	18906k	48.8%
	l3conv3s	1024×256	28.89	3615.58	12936k	6830k	47.2%
	l3ds	1024×512	55.15	5085.04	24232k	12637k	47.8%
	l4b0conv1	512×1024	65.57	975.88	34298k	17697k	48.4%
	l4conv1s	512×2048	127.57	1502.65	68165k	34883k	48.8%
	l4conv2s	512×4608	290.34	2623.38	153082k	77805k	49.2%
	l4conv3s	2048×512	127.18	37096.37	67276k	34871k	48.2%
	l4ds	2048×1024	235.57	51968.59	117806k	60491k	48.6%
	fc	1000×2048	250.60	9433.69	144232k	73624k	48.9%

of the matrix, the more iterations the while-loop would take. By applying the `curve_fit` function in the SciPy package to fit the data points of the experimental results, we found that the time complexity is about $O(m^{2.8}n^{0.49})$.

We synthesized some layers of ResNet18 to see whether the hardware costs reduce accordingly. The results are shown in Table V, where other larger layers are excluded due to their sizes too large to be synthesized successfully. The reported runtime is the Vivado synthesis time. As shown, the hardware cost can be effectively reduced by the proposed method. Note that for "opt-b", the hardware cost may be lower or higher than that for "plain". Basically, the hardware cost for "plain" and "opt-b" are close. However, the amount of non-zero bits reduced by Booth encoding depends on the binary numbers. Therefore, the hardware cost could be lower when the reduction rate is higher, and the hardware cost could be higher when the reduction rate is lower.

C. Comparison with Related Work

We compared our method with recent work on FPGA NN implementation, including logic synthesis of BNN (LSBNN) [11], NullaNet [13], LogicNet [16], NullaNet Tiny [25], AutoHQ [26], FINN [8], [9], resource-optimized BNN (ROBNN) [12]. We note that due to the different experimental settings and reported data among the related efforts, it is difficult to have a direct comparison for all the methods. However we tried our best to make individual comparisons as fair as possible.

To compare with LSBNN [11], MNIST and CIFAR10 are common datasets. The MNIST model in [11] consists of 6 FC layers and each hidden layer contains 256 neurons. However, because only FC layers of the CIFAR10 model were studied in [11], we mainly compared the MNIST model as shown in Table VI, where "-" indicates data unavailable. The LUT, CARRY and net counts exclude the input layer because of fundamental implementation difference[4]. Our implementation requires about 8.3% and 11.7% of the LUT and net counts, respectively, of LSBNN while achieving a similar accuracy. The result shows that the severe accuracy degradation of BNNs requires more hardware resources to achieve similar accuracy.

To compare with NullaNet [13], MNIST is the common dataset. In [13], the trained model consists of three hidden layers, each containing 100 neurons, and achieves accuracy about 97%. Their synthesis results are only reported for the 2^{nd} and 3^{rd} hidden layers with the clock frequency set to 65.3 MHz. Due to their network architecture difference and partial synthesis results, a head-to-head comparison is not possible. Thus, we use the synthesis results of the 2^{nd} (hidden) and 3^{rd} (output) layers of our NN to compare with those reported in [13]. Note that we used a relatively larger portion of our model to compare with a smaller portion of their model. The results are shown in Table VII, where the cost of NullaNet is measured by the number of adaptive logic modules (ALMs)[5]

[4]LSBNN uses pipelining to handle the 8-bit input bit by bit since their layer modules is designed to take 1-bit data as input. In contrast, our fixed-point implementation allows 8-bit input. Although the input layer of LSBNN has a smaller circuit size, it incurs a longer latency. Nevertheless, even if the input layer is taken into account, we still use fewer LUTs and nets. We use 62990 LUTs and 176949 nets while LSBNN uses 166338 LUTs and 254748 nets.

[5]An ALM can be configured to different architectures including a 6-input LUT. Hence an ALM is at least as complex as a LUT.

TABLE V
SYNTHESIS RESULTS OF SOME LAYERS OF RESNET18.

layer	setting	LUT (ratio)	net (ratio)	runtime (s)
conv1	plain	82228 (1.00)	285372 (1.00)	174
	opt-b	82703 (1.01)	263051 (0.92)	144
	opt-b+l	57557 (0.70)	167298 (0.59)	120
l1b0conv1	plain	314925 (1.00)	1103756 (1.00)	714
	opt-b	322147 (1.02)	1082360 (0.98)	534
	opt-b+l	206884 (0.66)	627766 (0.57)	342
l1b0conv2	plain	375487 (1.00)	1341042 (1.00)	702
	opt-b	377364 (1.00)	1267985 (0.95)	618
	opt-b+l	238502 (0.64)	728535 (0.54)	396
l1b1conv1	plain	348428 (1.00)	1260435 (1.00)	846
	opt-b	369647 (1.06)	1245446 (0.99)	594
	opt-b+l	233773 (0.67)	716330 (0.57)	396
l1b1conv2	plain	432742 (1.00)	1548713 (1.00)	1200
	opt-b	429555 (0.99)	1441361 (0.93)	684
	opt-b+l	267333 (0.62)	816864 (0.53)	432
l2b0conv1	plain	731644 (1.00)	2574193 (1.00)	1578
	opt-b	827769 (1.13)	2763151 (1.07)	1566
	opt-b+l	508768 (0.70)	1566730 (0.61)	942
l2b0conv2	plain	1212705 (1.00)	4318010 (1.00)	3570
	opt-b	1380469 (1.14)	4595711 (1.06)	2904
	opt-b+l	805935 (0.66)	2547204 (0.59)	1314
l2ds	plain	98935 (1.00)	307033 (1.00)	540
	opt-b	81978 (0.83)	253761 (0.83)	276
	opt-b+l	57380 (0.58)	168164 (0.55)	138
l2b1conv1	plain	1278912 (1.00)	4548210 (1.00)	3498
	opt-b	1473027 (1.15)	4918463 (1.08)	3084
	opt-b+l	854282 (0.67)	2715045 (0.60)	1386
l2b1conv2	plain	1304824 (1.00)	4628889 (1.00)	3636
	opt-b	1480653 (1.13)	4942341 (1.07)	3168
	opt-b+l	858766 (0.66)	2722117 (0.59)	1392
l3b0conv1	plain	2692679 (1.00)	9698021 (1.00)	8256
	opt-b	3375219 (1.25)	11505651 (1.19)	10374
	opt-b+l	1964089 (0.73)	6302067 (0.65)	3798
l3ds	plain	259948 (1.00)	933652 (1.00)	600
	opt-b	321572 (1.23)	1051295 (1.13)	870
	opt-b+l	219129 (0.84)	651447 (0.70)	408
l4ds	plain	1023071 (1.00)	3702576 (1.00)	2868
	opt-b	1412867 (1.38)	4747743 (1.28)	6618
	opt-b+l	890501 (0.87)	2729101 (0.74)	1734

TABLE VI
COMPARISON WITH LSBNN FOR MNIST DATASET.

models	accuracy	LUT (ratio)	CARRY	net (ratio)
LSBNN [11]	97.0%	98832 (1.00)	-	146617 (1.00)
ours	97.3%	8043 (0.08)	1107	17203 (0.12)

and the power consumption of ours is according to the Vivado power report. To further analyze the result, we normalize the LUT or ALM counts with respect to the number of neurons. The reported ALM count for NullaNet is for 200 neurons and thus each neuron needs about 560.9 ALMs. Our reported LUT count is for 58 neurons and thus each neuron needs about 138.7 LUTs, which is smaller. It is also reported in [13] that the overall hardware uses 79,607 MACs and a MAC requires 541 ALMs. Hence their model needs about 40M ALMs.[6] In contrast, our implementation needs only 62990 LUTs in total, which is much smaller. The binary activation used in NullaNet degrades the accuracy and leads to a higher hardware resource usage to achieve similar accuracy. For the power issue, from the table we observe that although our LUT usage is about 14 times lower, the power consumption is only about 1.4 times lower. There might be other sources contributing to the power consumption.

[6]The excessive cost is due to the dominance of the input layer, where the NullaNet optimization cannot be applied.

TABLE VII
COMPARISON WITH NULLANET FOR MNIST DATASET.

models	LUT or ALM	CARRY	power (mW)
NullaNet [13]	112173	-	396.46
ours	8043	1107	281.00

TABLE VIII
COMPARISON WITH PRIOR WORK FOR JSC DATASET.

models	accuracy	LUT	DSP	CARRY	T_{clk} (ns)	latency (ns)
JSC-L [16]	71.8%	37931	0	-	2.6	13.0
JSC-L [25]	73.4%	11752	-	-	2.3	11.5[a]
QE [26]	72.3%	9149	66	-	5.0	55.0
ours	75.0%	2481	0	386	5.7	34.2

[a] Computed.

To compare with LogicNet under the "NID-M" model [16], we train a model on the UNSWNB15 dataset [27], [28], for network-packet malicious detection. We follow the data preparation in [16] except that for each integer or floating-point input, they convert it into a (fixed point) binary vector and view each element in the vector as an input, while we use 6-bit fixed-point numbers to represent the input. Our model consists of four FC layers (with 20, 10, 10, and 1 neurons in order), using *relu6* as their activation function except for the output layer. The model is quantized to 4 bits and its accuracy is about 91.36%, which is slightly higher than that, 91.30%, of [16]. We use the same number of stages of pipelines and enable retiming when synthesizing to compare the delay and latency. The synthesis results show that our implementation needs 316 LUTs, the maximum delay is 4.37 ns and the latency is 21.8 ns. Their implementation needs 15949 LUTs, the maximum delay is 2.12 ns and the latency is 10.5 ns. Our implementation needs much fewer LUTs but the delay and latency is longer, showing that their approach is effective for getting high throughput and ours is resource efficient. Using LUTs only without arithmetic operations in LogicNet results in a lower latency. However, their hardware usage is higher due to the exponentially growing LUT numbers; also, the required sparsity in their model architecture degrades the accuracy.

Aside from UNSWNB15, the JSC [29] dataset is another commonly studied application for low latency. For this dataset, we trained a model which consists of three FC layers, with 20, 20, 5 neurons, respectively, and quantized the model to 4 bits. The testing accuracy for the quantized model is about 75.0%, and that for the floating-point version model is about 76.1%. We use our synthesis flow to generate the circuit and implement it using Vivado with the "`Performance_ExtraTimingOpt`" implementation strategy. We compare our circuit with previous work, including LogicNet [16], NullaNet Tiny [25] and AutoHQ [26]. The results are shown in Table VIII, where "T_{clk}" stands for the clock period and the latency for JSC-L [25] is computed from the information offered in the paper. We can observe that our method, achieving similar accuracy, is much more hardware efficient than the others. The efficiency comes from computation sharing and the accuracy maintaining of our

TABLE IX
COMPARISON WITH FINN FOR MNIST DATASET.

models	platform	accuracy	LUT	CARRY	clock frequency (Hz)	latency (μs)	throughput (kFPS)	on-chip power (W)	board power (W)	power efficiency (kFPS/W)
FINN SFC [8]	ZC706	95.87%	91131	-	200M	0.31	12361	7.3	21.2	1693
FINN LFC [8]	ZC706	98.40%	82988	-	200M	2.44	1561	8.8	22.6	177
FINN-R_1 [9]	Ultra96	97.69%	38205	-	300M	-	852	-	11.8	-
FINN-R_2 [9]	PYNQ-Z1	97.69%	25358	-	100M	-	162	-	2.5	-
FINN-R_3 [9]	AWS F1	97.69%	337753	-	232.9M	-	8463	-	-	-
ours	Artix-7	97.30%	59285	12083	55.56M	0.126	27778	1.886	-	14728

method. Because we impose no extra constraints, such as sparse connections and binary activations, on the NN model, our method requires no extra hardware to mitigate accuracy loss. On the other hand, the longer latency of our circuit compared to those of [16] and [25] may be due to the fact that we use the arithmetic operations in the software model for circuit synthesis without particular optimization for latency.

To compare with FINN [8], [9], we implemented our QNN for MNIST with a Xilinx Artix-7 FPGA. Pipelines are inserted between layers and after the sub-adders, and the input data are stored in block RAMs (BRAMs). The clock frequency is the one that meets timing constraints after the implementation step in Vivado. Since Artix-7 is a small board, the congestion level is high. We use the "`Flow_AreaOptimized_high`" synthesis strategy and "`Congestion_SpreadLogic_high`" implementation strategy in Vivado and reduce the clock frequency to successfully route the circuit. In fact, considering only the logic delay, the frequency can reach over 94 MHz. The power consumption is estimated using the Vivado power report after implementation. The results are shown in Table IX, where entries marked "-" indicate data unavailable. For the models of [8], the SFC is a model consisting of three FC layers, each with 256 neurons and the LFC is a model consisting of three FC layers, each with 1024 neurons. Comparing with them, we can achieve throughput 2.2 and 17.8 times higher than that of SFC and LFC, respectively. Besides, we can achieve 8.7 times and 83.2 times of power efficiency for SFC and LFC, respectively.

For the models of [9], they have the same architecture as LFC but with different hardware implementations. The model FINN-R_3 implemented on AWS F1 has the highest throughput among those reported in [9]. In comparison, our implementation achieves 3.3 times higher throughput and needs 5.5 times fewer LUTs. The model FINN-R_1 implemented on Ultra-96 has the best power-efficiency among those reported in [9], even considering implementations of QNNs for other tasks. Because only the board power consumption is available, we cannot directly compare. However we use LFC [8] as a baseline to estimate the relative power efficiency as the model architectures are identical. The board power efficiencies of LFC and FINN-R_1 are 69 and 72 kFPS/W, respectively. As the board power efficiency of these two are close and our implementation can achieve about 83.2 times chip power efficiency compared with the former one, these facts tend to suggest that the power efficiency of our implementation could be much better than that of FINN-R_1, although we need 1.55 times larger circuit size. We note that the smaller circuit size of FINN is due to its NPE-based implementation, which allows NPEs to be reused for different neuron computation. In contrast, direct logic implementation is advantageous in throughput and power efficiency.

TABLE X
COMPARISON WITH ROBNN ON MNIST DATASET.

models	LUT	CARRY	net	throughput (FPS)	power (W)	pw. eff. (FPS/W)
ROBNN	28618	-	48338	42.23M	0.70	60.33M
ours	5775	778	13694	64.52M	0.352	183.28M

In ROBNN [12], the MNIST dataset was simplified to a binary classification task to decide whether an input digit is smaller than 4. To compare with it, we trained a model consisting of three FC layers (with 16, 16, and 1 neurons in order). The weights and activations are quantized to 4 bits. The accuracy of our model is 97.36% while that in [12] is 96.13%. The accuracy of the floating-point version of our model is 97.67%. The implementation results are listed in Table X, where "pw. eff." stands for power efficiency. Our implementation needs only 20% LUTs and 28% nets to reach a similar accuracy. Moreover, ours has a 1.53 times higher throughput and 3.04 times better power efficiency. From the results, we can also see that the accuracy degradation of BNNs leads to higher hardware usage.

VI. CONCLUSIONS AND FUTURE WORK

We have proposed a synthesis flow for the direct transformation of QNNs into logic circuits. Effective subadder sharing has been achieved based on the use of the Booth matrix. Experimental results have demonstrated substantial reductions in LUT and net counts when implementing NNs using the proposed techniques, and have shown the superiority or competitiveness of our method compared to recent work. The low-latency, high-throughput, and high-power-efficiency implementation characteristics of our method make it suitable for edge devices and various applications. As future work, since our method can be applied to pretrained NNs, we plan to extend it to work for recurrent NNs (RNNs).

REFERENCES

[1] 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:1602.02830*, 2016.

[2] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proc. Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018, pp. 2704–2713.

[3] J. Choi, P. I.-J. Chuang, Z. Wang, S. Venkataramani, V. Srinivasan, and K. Gopalakrishnan, "Bridging the accuracy gap for 2-bit quantized neural networks (qnn)," *arXiv:1807.06964*, 2018.

[4] B.-E. Verhoef, N. Laubeuf, S. Cosemans, P. Debacker, I. Papistas, A. Mallik, and D. Verkest, "FQ-Conv: Fully quantized convolution for efficient and accurate inference," *arXiv:1912.09356*, 2019.

[5] Y. Choi, M. El-Khamy, and J. Lee, "Learning sparse low-precision neural networks with learnable regularization," *IEEE Access*, vol. 8, pp. 96 963–96 974, 2020.

[6] M. Nagel, M. V. Baalen, T. Blankevoort, and M. Welling, "Data-free quantization through weight equalization and bias correction," in *Proc. International Conference on Computer Vision (ICCV)*, 2019, pp. 1325–1334.

[7] E. Meller, A. Finkelstein, U. Almog, and M. Grobman, "Same, same but different-recovering neural network quantization error through weight factorization," *arXiv:1902.01917*, 2019.

[8] 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 *Proc. International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2017, pp. 65–74.

[9] 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 Trans. Reconfigurable Technol. Syst.*, vol. 11, no. 3, 2018.

[10] S. Sharify, A. D. Lascorz, M. Mahmoud, M. Nikolic, K. Siu, D. M. Stuart, Z. Poulos, and A. Moshovos, "Laconic deep learning inference acceleration," in *Proc. International Symposium on Computer Architecture (ISCA)*, 2019, pp. 304–317.

[11] C. Chi and J. R. Jiang, "Logic synthesis of binarized neural networks for efficient circuit implementation," in *Proc. International Conference on Computer-Aided Design (ICCAD)*, 2018, pp. 1–7.

[12] T. Murovič and A. Trost, "Resource-optimized combinational binary neural network circuits," *Microelectronics Journal*, vol. 97, p. 104724, 2020.

[13] M. Nazemi, G. Pasandi, and M. Pedram, "Energy-efficient, low-latency realization of neural networks through boolean logic minimization," in *Proc. Asia and South Pacific Design Automation Conference (ASPDAC)*, 2019, pp. 274–279.

[14] S. Tridgell, M. Kumm, M. Hardieck, D. Boland, D. Moss, P. Zipf, and P. H. Leong, "Unrolling ternary neural networks," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 12, no. 4, pp. 1–23, 2019.

[15] E. Wang, J. J. Davis, P. Y. K. Cheung, and G. A. Constantinides, "LUTNet: Learning FPGA configurations for highly efficient neural network inference," *IEEE Transactions on Computers*, vol. 69, no. 12, pp. 1795–1808, 2020.

[16] Y. Umuroglu, Y. Akhauri, N. J. Fraser, and M. Blott, "LogicNets: Co-designed neural networks and circuits for extreme-throughput applications," in *Proc. International Conference on Field-Programmable Logic and Applications (FPL)*, 2020, pp. 291–297.

[17] A. D. Booth, "A signed binary multiplication technique," *The Quarterly Journal of Mechanics and Applied Mathematics*, vol. 4, no. 2, pp. 236–240, 1951.

[18] A. G. Dempster and M. D. Macleod, "Constant integer multiplication using minimum adders," *IEE Proceedings - Circuits, Devices and Systems*, vol. 141, no. 5, pp. 407–413, 1994.

[19] S. Migacz, "8-bit inference with TensorRT," 2017. [Online]. Available: https://on-demand.gputechconf.com/gtc/2017/presentation/s7310-8-bit-inference-with-tensorrt.pdf (Accessed: Apr. 7, 2021)

[20] Z. Galil, "Efficient algorithms for finding maximum matching in graphs," *ACM Computing Surveys*, vol. 18, no. 1, pp. 23–38, 1986.

[21] Xilinx, *UG474: 7 Series FPGAs Configurable Logic Block*, 2014. [Online]. Available: https://docs.xilinx.com/v/u/en-US/ug474_7Series_CLB

[22] Altera, *FPGA Architecture White Paper*, 2006. [Online]. Available: https://www.intel.com/content/dam/support/us/en/programmable/support-resources/bulk-container/pdfs/literature/wp/wp-01003.pdf

[23] C. Wolf, "Yosys Open SYnthesis Suite," https://yosyshq.net/yosys/ (Accessed: Jul. 7, 2021).

[24] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[25] M. Nazemi, A. Fayyazi, A. Esmaili, A. Khare, S. N. Shahsavani, and M. Pedram, "NullaNet Tiny: Ultra-low-latency DNN inference through fixed-function combinational logic," in *Proc. International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2021, pp. 266–267.

[26] C. N. Coelho, A. Kuusela, S. Li, H. Zhuang, J. Ngadiuba, T. K. Aarrestad, V. Loncar, M. Pierini, A. A. Pol, and S. Summers, "Automatic heterogeneous quantization of deep neural networks for low-latency inference on the edge for particle detectors," *Nature Machine Intelligence*, pp. 1–12, 2021.

[27] N. Moustafa and J. Slay, "UNSW-NB15: a comprehensive data set for network intrusion detection systems (UNSW-NB15 network data set)," in *Proc. Military Communications and Information Systems Conference (MilCIS)*, 2015, pp. 1–6.

[28] N. Moustafa and J. Slay, "The evaluation of network anomaly detection systems: Statistical analysis of the UNSW-NB15 data set and the comparison with the KDD99 data set," *Inf. Sec. J.: A Global Perspective*, vol. 25, no. 1–3, pp. 18–31, 2016.

[29] M. Pierini, J. M. Duarte, N. Tran, and M. Freytsis, "HLS4ML LHC jet dataset (150 particles)," 2020. [Online]. Available: https://doi.org/10.5281/zenodo.3602260

Yu-Shan Huang (Student Member, IEEE) received the B.S. degree in electrical engineering from National Taiwan University, Taipei, Taiwan, in 2020, where he is currently pursuing the master's degree with the Graduate Institute of Electronics Engineering.

His research interests include neural network synthesis and machine learning.

Jie-Hong R. Jiang (Member, IEEE) received the B.S. and M.S. degrees in Electronics Engineering from National Chiao Tung University, Hsinchu, Taiwan, in 1996 and 1998, respectively, and the Ph.D. degree in Electrical Engineering and Computer Sciences from the University of California at Berkeley, Berkeley, CA, USA, in 2004.

He is a Professor with the Department of Electrical Engineering and the Graduate Institute of Electronics Engineering, National Taiwan University, Taipei, Taiwan. He leads the Applied Logic and Computation Laboratory, and has worked extensively on logic synthesis, formal verification, electronic design automation, and computation models of biological and physical systems.

Dr. Jiang is a member of Phi Tau Phi and the Association for Computing Machinery.

Alan Mishchenko (Senior Member, IEEE) received the M.S. degree from the Moscow Institute of Physics and Technology, Moscow, Russia, in 1993 and the Ph.D. degree from the Glushkov Institute of Cybernetics, Kiev, Ukraine, in 1997.

In 2002, he joined the EECS Department, University of California at Berkeley, Berkeley, CA, USA, where he is currently a Full Researcher. His research is in computationally efficient logic synthesis, formal verification, and machine learning.