2023 International VLSI Symposium on Technology, Systems and Applications (VLSI-TSA/VLSI-DAT) | 979-8-3503-3416-6/23/$31.00 ©2023 IEEE | DOI: 10.1109/VLSI-TSA/VLSI-DAT57221.2023.10134083

Minimizing Computation in Binarized Neural Network Inference using Partial-Filter Sharing

You-Hsuen Tsai[1], Yu-Chang Huang[1], Yi-Ting Li[1], Yung-Chih Chen[2], and Chun-Yao Wang[1]

[1]National Tsing Hua University, Taiwan, R.O.C.
[2]National Taiwan University of Science and Technology, Taiwan, R.O.C.

Abstract— Binarized Neural Network (BNN), which is a variant of Convolutional Neural Network (CNN) with binary weights and binary outputs on a neuron, has emerged as a promising approach to deploy artificial intelligence on resource-restricted devices in recent years. Due to the binarized weights, there are no longer required multipliers for computation, and there exist relatively high similarities among filters as well. In this work, we propose a partial-filter sharing approach and integrate it with the state-of-the-art to reduce the hardware cost and the synthesis time onto Field Programmable Gate Arrays (FPGAs). As compared to the state-of-the-art, the LUT reduction ratio by our approach is 47.71% on average without any accuracy loss, and 62.5% synthesis time in the Tiny ImageNet layers can be also saved on average.

I. Introduction

In recent years, Convolutional Neural Networks (CNNs) have achieved a huge success in artificial intelligence. It conquered plenty of problems that were hard to deal with using computers in the early days. Currently, deploying AI on edge devices such as smartphones, wearable devices, and IOT devices has become a trend to meet people's daily need. However, a great computational challenge still remains for achieving this goal. CNNs usually contain vast numbers of floating-point parameters and require enormous numbers of floating-point operations in both training and inference phases. This scenario makes CNNs difficult to be implemented on edge devices. As a result, minimizing the computational complexity of neural network inference on edge devices has become crucial.

To respond to this challenge, one attempt is to reduce the bit-width of parameters for CNN training and inference [5][6][12][15]. Among these works, a promising approach is to use binarized neural networks (BNNs)[2] proposed a method that confines the weights to be either +1 or -1 such that the multiply-accumulate operations in CNNs can be replaced with addition and subtraction operations. Then in [3][10], XNOR-Net was proposed with a restriction that both inputs and weights of the convolutional layers and fully connected layers, except the first layer, are binarized. Hence, it can further simplify the additions and subtractions to logical operations and be capable of exhibiting an efficient hardware realization.

Researchers have proposed many ideas to improve the inference performance by removing redundancies in BNNs. [13] demonstrated that batch normalization, activation, and max-pooling functions can be substituted with threshold functions and boolean OR operations. [4] decomposed 3-D filters into 2-D filters and exploited the repeated filters, inverse filters, and similar filters to share results in convolutional layers of BNNs. [8] observed that the process of calculating the outputs of related filters has a specific inclusion similarity relationship. With the relationship, the original XNOR-Popcount operations in BNNs can be further reduced to Threshold-Comparable-Popcount (TCP) operations. Different from [4], [14] proposed a flexible result sharing approach and reused the computed result among partial 1-D filters instead of 2-D filters.

According to the observation that enormous redundant operations still remain after these optimizations, this work proposes a partial-filter sharing approach with a covering technique to reduce the number of operations in the inference model of BNNs and saves the resource usage on edge devices.

To demonstrate the effectiveness of the proposed approach on edge devices, we synthesized the inference models with the proposed approach on a field programmable gate array (FPGA) platform, Xilinx ZCU104. Experiments on Tiny ImageNet [9] BNN models show that the LUT reduction rate by our approach is 47.03% on average compared to the previous works [8]. Furthermore, 62.5% time for synthesis in the Tiny ImageNet layers can also be saved on average.

II. Preliminaries

A BNN is a variant of CNN, which maintains the similar structure of CNN but its weights and activations are either -1 or 1. With the effort of [3][10][13], only three types of operations remain in BNNs. The Type-I is an XNOR-Popcount operation, which can be expressed as:

$$y = \sum_{i=1}^{n} (w_i \odot x_i), \tag{1}$$

where XNOR operation is denoted by $\odot$, and x_i, w_i are the inputs and weights, respectively.

The Type-II is an OR operation, which substitutes the original max-pooling operation in the pooling layer. The Type-II operation can be expressed as:

$$y = OR(x_0, ..., x_i, ..., x_n), \tag{2}$$

where x_i is the value to be processed by the max-pooling operation.

The Type-III is a threshold operation, which is transformed from the batch normalization and activation function. The Type-III operation can be expressed as:

$$y = \begin{cases} 1 \ ; \ \text{if } x \geq T \\ 0 \ ; \ \text{otherwise} \end{cases} \tag{3}$$

where T is the threshold value.

Since the same position of different filters corresponds to the same input, repeated operations in both convolutional layers and fully connected layers may occur. The authors in [4] proposed an approach to find the repeated filters for sharing the convolutional results, thereby reducing the number of operations in BNNs. As the number of unique 3-D filters grows exponentially with the increase of number of bits in a 3-D filter, the authors proposed to decompose all the 3-D filters into 2-D filters for having a smaller number of unique 2-D filters. To further reduce this number, they considered

This work is supported in part by the Ministry of Science and Technology of Taiwan under MOST 109-2221-E-007-082-MY2, MOST 109-2221-E-155-047-MY2, MOST 110-2224-E-007-007, MOST 111-2218-E-007-010, MOST 111-2221-E-007-121, and MOST 111-2221-E-011-137-MY3.

Authorized licensed use limited to: National Tsing Hua Univ.. Downloaded on August 12,2023 at 02:23:54 UTC from IEEE Xplore. Restrictions apply.

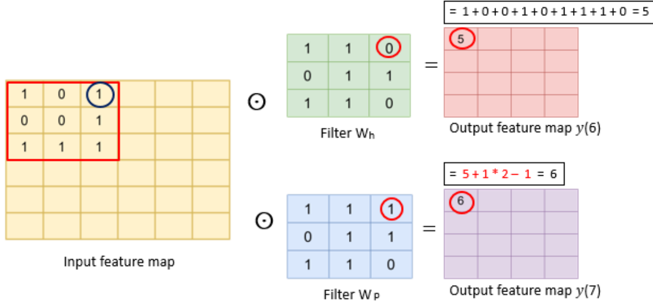


Fig. 1. An example of filter inclusion similar phenomenon.

the inverse filter as a filter repetition as well. As a result, the possibility that the result of a 2-D filter performing XNOR-Popcount operations can be shared to others is greatly increased.

[14] inherited the ideas of repeated filters and inverse filters, and flattened 3-D filters into 1-D filters. Then the 1-D filters are divided into multiple segments with a length of l bits. For example, when we have 64 $64 \times 3 \times 3$ 3-D filters, these filters will be flattened as 64 576-bit 1-D filters. When the length l of a segment is set to 4, each 1-D filter will be decomposed into 144 segments and the number of unique segments is only $2^4 = 16$. Since the same index of segment corresponds to the same input, the result of one segment performing XNOR-Popcount operation can be shared to the same index segment in different filters when the weights of these segments are all the same. It is more possible to share the results when a 1-D filter is decomposed into multiple segments since the number of unique segments are much fewer than the number of unique 2-D filters.

When dividing a 1-D filter into multiple segments, the results of all the segments after performing XNOR-Popcount operations have to be accumulated to obtain the output of a complete operation. Since the accumulations of these results also require computation, the authors of [14] extended the idea of result sharing to the partial-result accumulation. They set the quantity of filters sharing the accumulation results to two for pursuing more partial-result sharings.

The authors in [8] aimed to simplify the XNOR-Popcount operation even further. They observed that when there is only one bit with different values between two filters, only one convolutional result of these two filters corresponding to the same input is different and the others are the same. The mathematical relationship between their corresponding convolutional outputs can be expressed as:

$$\begin{aligned} y(n) &= y(n-1) + (input_n \times 2) - 1 \\ &= y(n-1) + in_n - 1 \end{aligned} \tag{4}$$

where $n = 1 \sim$ *filter size* (H)[1], *n is the number of bits with value of 1 in the filter*, $input_n$ is the input bit of the sliding window corresponding to the only different weight between filters, in_n is two times of $input_n$, and $y(n)$ and $y(n-1)$ are the results of the input of the sliding window convolving with the filters that have n and $(n-1)$ bits of 1, respectively. Note that $y(0) = H - \sum_{i=1}^{H} input_i$ due to XNOR-Popcount operation.

For example in Fig. 1, the input of the sliding window is [1, 0, 1, 0, 0, 1, 1, 1, 1] [2], the filter W_h is [1, 1, 0, 0, 1, 1, 1, 1, 0], and the filter W_p is [1, 1, 1, 0, 1, 1, 1, 1, 0]. By performing the XNOR-Popcount operation for the input and filter W_h, the result in the output $y(6)$ for this operation is five. Since the only difference between the filters W_h and W_p is at the 3rd bit, the result in the output $y(7)$ can be obtained by using $y(6)$ and the 3rd bit in the sliding window of the input feature map according to EQ(4). Thus, $y(7) = y(6) + (1 \times 2) - 1 = 6$, which is the same as performing the XNOR-Popcount operation for the input in the sliding window and the filter W_p.

[1]The filter size H is the decimal number of an $m \times m$ filter. For example, the filter size H of a 3×3 filter is 9.

[2]The order is from the left to the right, from the top to the bottom.

Given a filter having zero bit of 1 and another filter having one bit of 1, the mathematical relationship between their outputs can be expressed as: $y(1) = y(0) + in_1 - 1$. Similarly, $y(2) = y(1) + in_2 - 1$. As replacing $y(1)$ with $y(0)+in_1-1$, $y(2) = y(0)+in_1-1+in_2-1 = y(0) + in_2 + in_1 - 2$. Hence, EQ(4) can be generalized as EQ(5) without involving any XNOR-Popcount operation:

$$\begin{aligned} y(n) &= y(0) + \sum_{i=1}^{n} in_i - n \\ &= (H - \sum_{i=1}^{H} input_i) + \sum_{i=1}^{n} in_i - n \end{aligned} \tag{5}$$

Since n is the number of bits with value of 1 in the filter, it can be computed beforehand and merged into the corresponding threshold value without influencing the final results. As a result, we rewrite the new threshold value t_{new} as:

$$t_{new} = t_o + n \tag{6}$$

where t_o is the original threshold value, and EQ(5) will become EQ(7) accordingly.

$$y(n) = y(0) + \sum_{i=1}^{n} in_i \tag{7}$$

III. Proposed Approach

We exploit the repeated partial-result to minimize operation counts in BNNs. Also, the original XNOR-Popcount operation in BNNs can be simplified as the accumulation of the inputs corresponding to the weight with value of 1. Thus, we can formulate the problem of finding the partial-filter repetition as a covering problem. First, all the 3-D filters are flattened in the row-major order as 1-D filters. Then, all the 1-D filters with l bits are arranged in the *filter table* as shown in Fig. 2. A rectangle covering algorithm is then used to find the common partial-filters iteratively such that more 1s are covered by rectangles.

A. Rectangle Covering

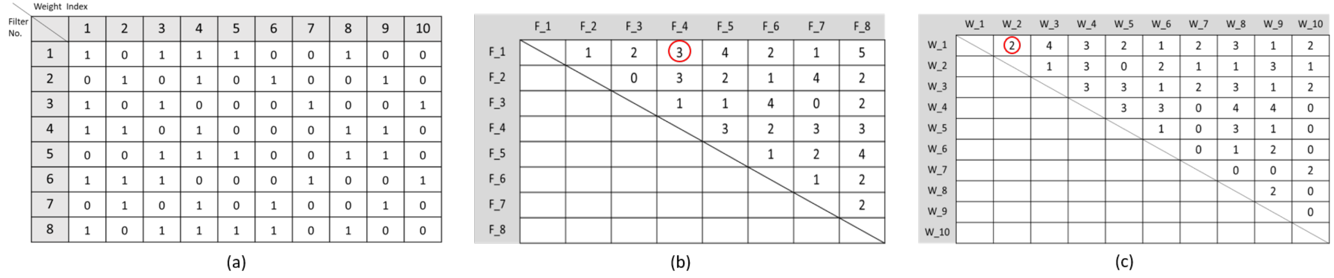
A rectangle is defined as a set of rows and columns such that all the elements are 1. Each rectangle has a *gain* representing the number of addition operations that can be saved in the implementation. For example in the table of Fig. 2, there are four filters with length of 10 bits. The row set $\{1, 2, 3\}$ and the column set $\{2, 3, 4, 10\}$ form a rectangle meaning that Filters 1, 2, 3 share the partial-result of $(in_2 + in_3 + in_4 + in_{10})$. Thus, the number of addition operations saved by this rectangle, i.e, *gains*, is $2 \times 3 = 6$. The *gain* of a selected rectangle r_i is expressed as:

$$gain(r_i) = (F - 1) \times (C - 1) \tag{8}$$

Filter No. \ Weight Index.	1	2	3	4	5	6	7	8	9	10
1	1	1	1	1	1	0	0	0	0	1
2	1	1	1	1	0	0	1	1	1	1
3	0	1	1	1	0	0	0	0	1	1
4	0	1	0	0	1	0	1	1	1	0

Fig. 2. An example of a filter table.

Authorized licensed use limited to: National Tsing Hua Univ.. Downloaded on August 12,2023 at 02:23:54 UTC from IEEE Xplore. Restrictions apply.



(a)

Filter No. \ Weight Index	1	2	3	4	5	6	7	8	9	10
1	1	0	1	1	1	0	0	1	0	0
2	0	1	0	1	0	1	0	0	1	0
3	1	0	1	0	0	0	1	0	0	1
4	1	1	0	1	0	0	0	1	1	0
5	0	0	1	1	1	0	0	1	1	0
6	1	1	1	0	0	0	1	0	0	1
7	0	1	0	1	0	1	0	0	1	0
8	1	0	1	1	1	1	0	1	0	0

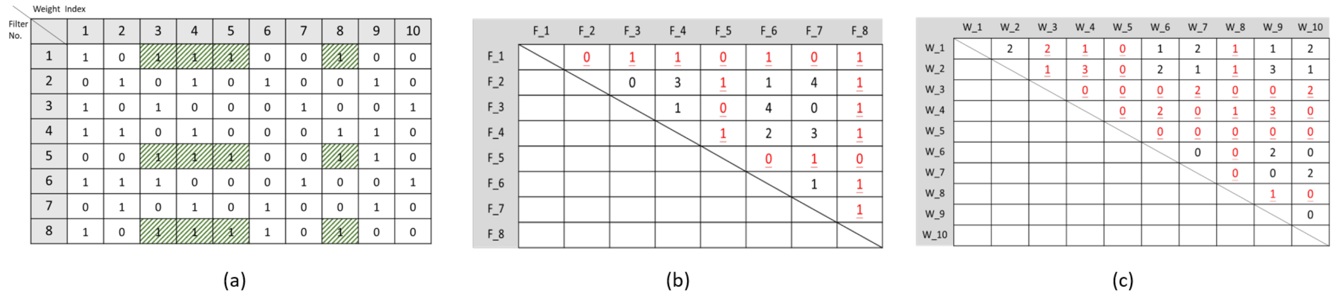
(b)

	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8
F_1		1	2	3	4	2	1	5
F_2			0	3	2	1	4	2
F_3				1	1	4	0	2
F_4					3	2	3	3
F_5						1	2	4
F_6							1	2
F_7								2
F_8								

(c)

	W_1	W_2	W_3	W_4	W_5	W_6	W_7	W_8	W_9	W_10
W_1		2	4	3	2	1	2	3	1	2
W_2			1	3	0	2	1	1	3	1
W_3				3	3	1	2	3	1	2
W_4					3	3	0	4	4	0
W_5						1	0	3	1	0
W_6							0	1	2	0
W_7								0	0	2
W_8									2	0
W_9										0
W_10										

Fig. 3. (a) Filter table with 8 filters and 10 weights. (b) The filter similarity table. (c) The weight similarity table.



(a)

Filter No. \ Weight Index	1	2	3	4	5	6	7	8	9	10
1	1	0	1	1	1	0	0	1	0	0
2	0	1	0	1	0	1	0	0	1	0
3	1	0	1	0	0	0	1	0	0	1
4	1	1	0	1	0	0	0	1	1	0
5	0	0	1	1	1	0	0	1	1	0
6	1	1	1	0	0	0	1	0	0	1
7	0	1	0	1	0	1	0	0	1	0
8	1	0	1	1	1	1	0	1	0	0

(b)

	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8
F_1		0	1	1	0	1	0	1
F_2			0	3	1	1	4	1
F_3				1	0	4	0	1
F_4					1	2	3	1
F_5						0	1	0
F_6							1	1
F_7								1
F_8								

(c)

	W_1	W_2	W_3	W_4	W_5	W_6	W_7	W_8	W_9	W_10
W_1		2	2	1	0	1	2	1	1	2
W_2			1	3	0	2	1	1	3	1
W_3				0	0	0	2	0	0	2
W_4					0	2	0	1	3	0
W_5						0	0	0	0	0
W_6							0	0	2	0
W_7								0	0	2
W_8									1	0
W_9										0
W_10										

Fig. 4. (a) The updated filter table after the first iteration in Fig. 3. (b) The updated filter similarity table after the first iteration. (c) The updated weight similarity table after the first iteration.

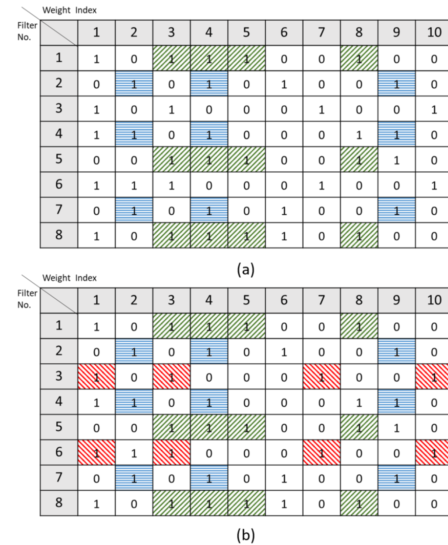
where F is the number of filters and C is the number of weights in this rectangle. Our covering problem is to find a set of disjoint rectangles R, $R = \{r_1, ..., r_i, ..., r_n\}$, such that the *total gain* $= \sum_{i=1}^{n} gain(r_i)$ is maximized.

B. Find the Best Rectangle

We first build a *filter similarity table* for selecting the *best row* of the *filter table*. The number of weights with value of 1 in the same column of the *filter table* between Filters i, j is recorded in the position of row i, column j of the *filter similarity table*. For example, there are 8 filters with 10 weights in Fig. 3(a). The weights of the 1st, 4th and 8th column in Filters 1 and 4 are both 1. Hence, 3 is recorded in the position of row 1 and column 4 of the *filter similarity table* as shown in Fig. 3(b), indicating that three weights of 1 are in common between Filters 1 and 4.

Since the maximum value 5 in the *filter similarity table* is at the row of F_1, the row of F_1 is selected as the *best row*. We find the *best rectangle* from the *best row* as follows. The *best row* in this example represents Filter 1. We add another filter to form a rectangle with a larger gain. Hence, the criterion for selecting the added filter in each step is based on the maximum value in the *best row* of the *filter similarity table*. The process of finding the *best rectangle* will be terminated until the value in the *best row* of the *filter similarity table* is less than 2. The rectangle with the best *gain* is recorded in this procedure.

According to the *filter similarity table* in Fig. 3(b), the order of the added filter is the 8th row, the 5th row, the 4th row, the 3rd row, and then the 6th row of Fig. 3(a). Thus, we first add Filter 8 into the initial *best rectangle*. The *best rectangle* is then updated to the row set $= \{1, 8\}$ and the column set $= \{1, 3, 4, 5, 8\}$, and the $gain(r_{best}) = (2 - 1) \times (5 - 1) = 4$. In the next step, Filter 5 is added to the rectangle. The *gain* of the *best rectangle* is updated as $gain(r_{best}) = (3 - 1) \times (4 - 1) = 6$, which corresponds to the row set $= \{1, 5, 8\}$ and the column set $= \{3, 4, 5, 8\}$. Then in the succeeding steps, the new rectangles created by adding other filters cannot have a better gain than the *best rectangle*. Thus, the *best rectangle* in the first iteration is the row set $= \{1, 5, 8\}$ and the column set $= \{3, 4, 5, 8\}$ as shown in Fig. 4(a). We can also build a *weight similarity table* as shown in Fig. 3(c) for selecting the *best column* of the *filter table*. The number of filters having the same weights with value of 1 between columns i and j of the *filter table* is recorded in the position of row i, column j of the *weight similarity table*. In Fig. 3(a), the weights of the 1st and 2nd column in Filters 4 and 6 are both 1. Hence, 2 is recorded in the position of row 1 and column 2 of the *weight similarity table* as shown in Fig. 3(c), indicating that there are two filters in common between the 1st and 2nd weights. The remaining steps of finding the *best rectangle* from the *best column* follow the same procedure of finding the *best rectangle* from the *best row*, except expanding the rectangle by adding one column each time.



(a)

Filter No. \ Weight Index	1	2	3	4	5	6	7	8	9	10
1	1	0	1	1	1	0	0	1	0	0
2	0	1	0	1	0	1	0	0	1	0
3	1	0	1	0	0	0	1	0	0	1
4	1	1	0	1	0	0	0	1	1	0
5	0	0	1	1	1	0	0	1	1	0
6	1	1	1	0	0	0	1	0	0	1
7	0	1	0	1	0	1	0	0	1	0
8	1	0	1	1	1	1	0	1	0	0

(b)

Filter No. \ Weight Index	1	2	3	4	5	6	7	8	9	10
1	1	0	1	1	1	0	0	1	0	0
2	0	1	0	1	0	1	0	0	1	0
3	1	0	1	0	0	0	1	0	0	1
4	1	1	0	1	0	0	0	1	1	0
5	0	0	1	1	1	0	0	1	1	0
6	1	1	1	0	0	0	1	0	0	1
7	0	1	0	1	0	1	0	0	1	0
8	1	0	1	1	1	1	0	1	0	0

Fig. 5. The results of the second and third iterations after applying our approach in Fig. 3.

After identifying these two *best rectangles* starting from the *best*

Authorized licensed use limited to: National Tsing Hua Univ.. Downloaded on August 12,2023 at 02:23:54 UTC from IEEE Xplore. Restrictions apply.

row and the *best column*, we choose the better one as the *best rectangle* in this iteration. We then update the *filter table* as follows. First, those 1s in the *filter table* corresponding to the *best rectangle* are changed to 0. Next, the data in the *filter similarity table* will be updated as well. For example, in the first iteration of finding the *best rectangle* in Fig. 3(a), we select Filters 1, 5, and 8, and Weights 3, 4, 5, and 8 as the *best rectangle*. Thus, those 1s in the *filter table* are changed to 0. Then, in the *filter similarity table*, those updated values are underlined in red in Fig. 4(b) about F_1, F_5 and F_8. Similarly, the data in the *weight similarity table* corresponding to the column set of the *best rectangle* is updated as Fig. 4(c). The above process of finding the *best rectangle* will be iterated until $gain(r_{best}) < 2$. The results of finding the *best rectangle* in the second and third iterations are shown in Figs. 5(a) and 5(b). In the second iteration, we select Filters 2, 4, and 7, and Weights 2, 4, and 9 as the *best rectangle*, whose *gain* is 4. Then in the third iteration, we select Filters 3 and 6, and Weights 1, 3, 7, and 10 as the *best rectangle*, whose *gain* is 3. Finally, since the *gain* of the *best rectangle* in the fourth iteration is smaller than 2, the whole process of finding the *best rectangle* is terminated.

IV. Experimental Results

To demonstrate that the proposed approach can be realized in BNNs, one BNN model trained by ourselves in the experiments. We use python 3.8 as our programming language to train BNNs and run our approach to generate designs in Verilog files for the BNNs realization. The experimental environment is Linux Ubuntu 18.04, with Intel i7-8700 CPU, Nvidia 1080Ti GPU, and 32GB RAM. We use Xilinx Vivado 2019.1 for synthesis and deploy the BNNs on ZCU104 Development Board with Xilinx XCZU7EV-2FFVC1156 MPSoC.

Since the functionality of our implementation is exactly the same as the one obtained from the FINN compiler, there is no further accuracy loss with our approach. The synthesized results are shown in TABLE I. Column 1 "LAYER i (SIZE)" in the table indicates the ith layer of the network, and the size of the layer, which is the width of the input × the number of the filters to be processed. Note that the synthesized results only include the convolutional layers and fully connected layers. We reimplemented the approach of [8] for each layer on ZCU104 Development Board for comparison. Columns 3, 4, and 5 in the tables show the number of required LUTs, the LUT reduction ratio, and the synthesis time, respectively. We compare the results of required LUTs in our approach with the approach [8] to get the LUT reduction ratio. The last column of the table shows the CPU time when performing the proposed approach.

We trained the network for Tiny ImageNet dataset [9] based on the architecture of VGG-13[11]. The network topology on Tiny ImageNet datasets has six and ten convolutional layers. It has three fully connected layers and two max-pooling layers right after two convolutional layers, LAYERs 1 and 4. Our approach is not applied to the first convolutional layer, LAYER 0, and LAYERs 2 and 5 since the inputs of LAYER 0 are not binarized, and LAYERs 2 and 5 are max-pooling layers. TABLE I shows the synthesis results.

When considering all the layers, our approach on Tiny ImageNet achieves 47.71% reduction of the number of the required LUTs on average as compared to [8]. Furthermore, most layers in TABLE I have a significant reduction of the synthesis time. Take all the layers of the two models into consideration, the synthesis time is reduced by 62.5% on average in our approach as compared to [8]. It takes 331.48 seconds on average for running the program, which is relatively inexpensive compared to the synthesis time. Thus, the proposed approach simultaneously reduces the requirement of hardware resources and synthesis time when deploying BNNs on the FPGA platforms.

TABLE I
Synthesis results of BNN layers on Tiny ImageNet dataset.

LAYER i (SIZE)		R. LUT	Red. Ratio (%)	Syn. time (s)	CPU Time (s)
LAYER 1	[8]	23899	-	134	-
(576×64)	Ours	13209	44.73	100	2.52
LAYER 3	[8]	46848	-	123	-
(576×128)	Ours	25469	45.63	89	5.94
LAYER 4	[8]	104015	-	228	-
(1152*128)	Ours	48412	53.46	110	13.81
LAYER 6	[8]	201812	-	566	-
(1152×256)	Ours	93875	53.48	202	46.03
LAYER 7	[8]	381120	-	1621	-
(2304×256)	Ours	184327	51.64	431	101.02
LAYER 8	[8]	752395	-	3983	-
(2304×512)	Ours	359737	52.19	990	470.69
LAYER 9	[8]	1204342	-	7231	-
(4608×512)	Ours	671851	44.21	3089	773.81
LAYER 10	[8]	1225791	-	7770	-
(4608×512)	Ours	672517	45.14	3430	810.21
LAYER 11	[8]	1216081	-	8028	-
(4608×512)	Ours	657739	45.91	3168	783.82
LAYER 12	[8]	631318	-	3901	-
(512×2048)	Ours	346045	45.19	1213	499.22
LAYER 13	[8]	632193	-	3369	-
(2048×512)	Ours	333951	47.18	904	458.65
LAYER 14	[8]	70059	-	239	-
(512×200)	Ours	39378	43.79	203	11.22
AVERAGE	[8]	540823	-	3099.4	-
	Ours	287209	47.71	1160.8	331.48

V. Conclusion

In this paper, we propose a partial-filter sharing approach and integrate it with the state-of-the-art for minimizing computation in BNNs. The proposed approach exploits the filter repetitions of partial-filters to share their partial-results. Experimental results show that the proposed approach significantly reduces the number of required LUTs and the synthesis time on FPGA platforms for BNN realization.

References

[1] Chia-Chih Chi and Jie-Hong R Jiang, "Logic Synthesis of Binarized Neural Networks for Efficient Circuit Implementation," in *Proc. of ICCAD*, 2018.

[2] Matthieu Courbariaux, et al., "Binaryconnect: Training Deep Neural Networks with Binary Weights during Propagations," *arXiv preprint arXiv:1511.00363*, 2015.

[3] Matthieu Courbariaux, et al., "Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1," *arXiv preprint arXiv:1602.02830*, 2016.

[4] Ya-Chun Chang, et al., "A Convolutional Result Sharing Approach for Binarized Neural Network Inference," in *Proc. of DATE*, 2020, pp. 780-785.

[5] Tong Geng, et al., "O3BNN: An Out-Of-Order Architecture for High-Performance Binarized Neural Network Inference with Fine-Grained Pruning," in *Proc. of ICS*, 2019, pp. 461-472.

[6] Itay Hubara, et al., "Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations," *JMLR*, 2015, vol. 18, no. 1, pp. 6869-6898.

[7] Alex Krizhevsky and Geoffrey Hinton, "Learning Multiple Layers of Features from Tiny Images," *Citeseer*, 2009.

[8] Qingliang Liu, et al., "TCP-Net: Minimizing Operation Counts of Binarized Neural Network Inference," in *Proc. of ISCAS*, 2021, pp. 1-5.

[9] Ya Le and Xuan S. Yang, "Tiny ImageNet Visual Recognition Challenge," 2015.

[10] Mohammad Rastegari, et al., "Xnor-net: Imagenet Classification Using Binary Convolutional Neural Networks," in *Proc. of ECCV*, Springer, 2016, pp. 525-542.

[11] Karen Simonyan and Andrew Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition" in *Proc. of ICLR*, 2015.

[12] Wei Tang, et al., "How to train a compact binary neural network with high accuracy?," in *Proc. of AAI*, 2017, pp. 2625-2631.

[13] Yaman Umuroglu, et al., "Finn: A Framework for Fast, Scalable Binarized Neural Network Inference," in *Proc. of FPGA*, 2017, pp. 65-74.

[14] Sheng-Hsiu Wei, et al., "A Flexible Result Sharing Approach to Binarized Neural Networks Optimization," MS thesis, University of Tsing Hua, 2021.

[15] Shuchang Zhou, et al., "Dorefa-net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradient," *arXiv preprint arXiv:1606.06160*, 2016.

Authorized licensed use limited to: National Tsing Hua Univ.. Downloaded on August 12,2023 at 02:23:54 UTC from IEEE Xplore. Restrictions apply.