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-DAT57221.2023.10134036

Accelerating Binarized Neural Network Inference by Reusing Operation Results and Elevating Resource Utilization on Edge devices

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

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

Abstract— Convolutional Neural Networks (CNNs) have shown their abilities in computer vision such as self-driving car and image classification applications. However, computation, power consumption, and memory requirement remain challenges for CNNs when applied in the domains of edge devices. To address these challenges, one of the possible solutions is using Binarized Neural Networks (BNNs). Researchers have demonstrated that BNN models dramatically reduce computational complexity and memory requirements with acceptable accuracy loss. This paper presents an accelerator design for BNN inference that minimizes the number of operations and elevates resource utilization on edge devices. We implemented our accelerator on the Xilinx ZCU104 FPGA and evaluated it with VGG-13 like BNN networks for Tiny ImageNet. Experimental results show that our accelerator can reduce, on average, 99.6% operations and achieve up to 1974 speedup on an FPGA platform compared to state-of-the-art design.

I. Introduction

In recent years, Deep Neural Networks (DNNs) have achieved great success in computer vision. CNN is a particularly popular type of DNN as it can effectively capture spatial features from images. However, CNN models require massive memory and consume much power in operation, which are bottlenecks when CNNs are deployed to resource and power-constrained edge devices.

A reduced bit width CNN - Binarized Neural Network (BNN)[11][9] recently receives more attentions in academia and industry. BNNs simplify the original activations and weights from floating-point values to bipolar values (+1 or -1), which are realized with 1 or 0 in actual hardware implementations. Therefore, memory usage and computational complexity are dramatically reduced. Meanwhile, MeliusNet [1] has also shown that BNNs were still able to achieve good accuracy and even outperformed the floating-point network such as MobileNet [5].

[9][6] proposed that using the characteristics of binarized values, BNNs can transform the frequently used multiplications and additions into XNOR operations and popcount operations, respectively, which allow processing element (PE) designs to be simplified even further without compromising accuracy.

Furthermore, batch normalization functions in the outputs of the convolutional layers and fully connected layers can be transformed into a threshold comparison operation[12]. With this transformation, we avoid complex batch normalization functions during inference.

However, there exist many redundant operations in the computation of BNN inference. Thus, many studies [12] [3] [2] [4] [7] [8] proposed methods to facilitate the BNN deployment on edge devices. [3] fused the convolutional layers and the first fully connected layer into inter-layer pipelining architecture. Hence, the latency of binarized AlexNet, VGGNet, and ResNet can be reduced a lot. Other studies removed redundancy in the BNN inference. [4] proposed Out-of-Order architecture that checks whether the binary output can be generated earlier, thus reducing redundant operations in execution. It pruned up to 30% operations on average without any accuracy loss. Moreover, [7][8] combined their proposed threshold-comparable-popcount (TCP) operations and the threshold value update method to achieve a 79% operation reduction. In [2], the authors introduced a filter's convolutional result sharing approach for convolution layers such that the total number of operations could be reduced up to 70%.

In this work, we propose a design of accelerator which minimizes the number of operations during BNN inference. The state-of-the-art PE-based architecture such as [12][4][7][8] decomposed an XNOR-popcount operation between inputs and filters into several segments. The decomposed filter is called a partial-filter, where the length of each partial-filter is determined to balance the throughput and hardware utilization. The result generated by an XNOR-popcount operation between the partial-filter and its corresponding input is called a partial-result. We exploit multiplexer-based similarity operation to reuse the partial-result and increase the resource utilization. The number of redundant operations during inference can be reduced effectively.

The main contributions of this work are as follows:

- We propose a design of accelerator for the convolution layers and fully connected layers, which reduces the number of the operations dramatically without any accuracy loss and elevates the resource utilization.
- The proposed architecture can be easily deployed and modified with respect to the hardware resources, which achieves up to 1974 speedup in BNN inference in our experimental results.

II. Preliminaries

A. Convolutional Neural Networks

A convolutional neural network is a type of supervised deep learning architecture that can capture an image's spatial features. A common convolutional neural network such as VGG-16 [10] contains convolutional layers, fully connected layers, and pooling layers. A convolutional layer uses a sliding-window with a stride of S to traverse and perform convolutional operations between the input feature map and filter.

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.

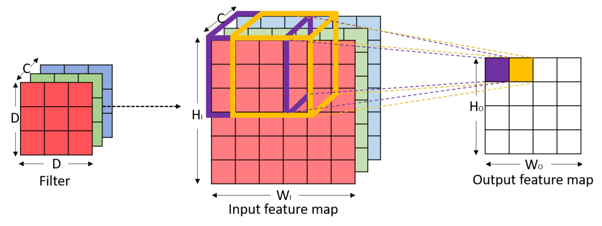


Fig. 1. An example of a convolutional operation.

An example of a convolutional operation is shown in Fig. 1, $C \times D \times D$ filter can be expressed as follows:

$$Output = F_{act}(\sum_{i=0}^{C-1}\sum_{j=0}^{D-1}\sum_{k=0}^{D-1} w_{i,j,k} \cdot x_{j,k})$$

where $w_{i,j,k}$ is the weight of location (i, j, k) in the filter, $x_{j,k}$ is the input on the location of (j, k) of input feature map, C is the channel size of the input feature map, and D is the window size of the filter. After the summation, the activation function F_{act} is used to determine the output. Popular activation functions include *Sigmoid*, *ReLU*, and *Tanh*. A fully connected layer classifies the input feature map into various classes. The input feature map in the fully connected layer is unfolded into a 1-D input vector and then performs matrix multiplications with the 1-D weight vector. The following equation determines the output of a fully connected layer:

$$Output = F_{act}(\sum_{i=0}^{N-1} w_i \cdot x_i + b)$$

where w_i and x_i are the i^{th} value in the 1-D weight vector and input vector respectively, b is the bias, and N represents the length of the 1-D weight vector. The pooling layer down-samples the input feature map with some pooling methods. Common pooling methods are max pooling and average pooling for extracting value from the $K \times K$ window, and we use the max pooling in this work. The max pooling operation can be expressed as follows:

$$Output = \max_{i,j=0}^{K-1}(x_{i,j})$$

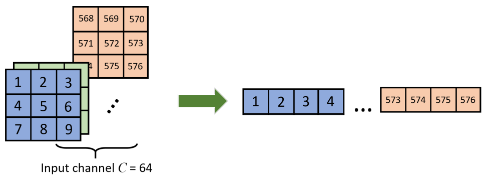
where $x_{i,j}$ is the data located at (i, j) of the input feature map. The max function extracts the maximum value from the corresponding positions in the input feature map.

B. Binarized Neural Networks

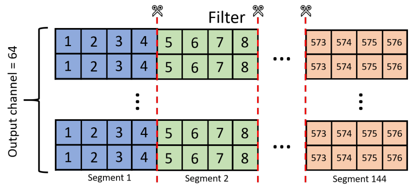
The parameters in BNNs are restricted to +1 or -1. Therefore, in the hardware implementation, we can use a 1-bit value to represent them. With this property, [6] proposed that in BNNs, only XNOR and bit accumulation (popcount) operations are needed instead of multiplications and additions. An XNOR-popcount operation can be expressed as follows:

$$Output = \sum_{i=0}^{N-1} w_i \odot x_i$$

where w_i and x_i are the values of weight and input bit on the location i in the 1-D weight and input vectors, N is the length of the unfolded 1-D weight, and input vectors. [12] proposed that the batch normalization layer on the output of the convolutional and fully connected layers in BNNs can be converted into a threshold comparison operation. Thus, we

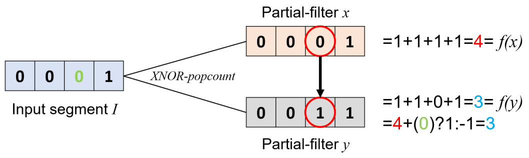


(a)

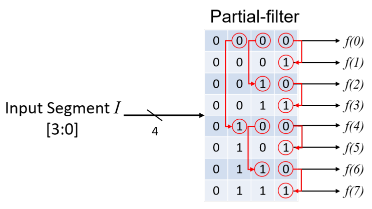


(b)

Fig. 2. (a) An example of unfolding a 64x3x3 filter. (b) An example of decomposing a 64x3x3 filter when output channel size = 64.



(a)



(b)

Fig. 3. (a) An example of a multiplexer-based similarity operation. (b) Relationship between any two filters.

can combine the batch normalization and binarized operation into one threshold comparison operation to reduce the usage of hardware resources.

Moreover, [2] proposed a result sharing approach that significantly reduces the number of redundant operations in the convolutional layer. Using the property of BNNs, they decomposed 3-D filters into 2-D filters and utilized the repeated filters to share the convolutional results. Furthermore, they observed that the results between two inverted filters can be inferred from each other. These methods minimize the total number of operations without any accuracy drops and achieve a high operation reduction ratio.

III. Proposed Approach

A. Multiplexer-based Similarity Operation

In hardware realization, a filter will be unfolded into a 1-D weight vector, turning the convolutional operation into the same matrix multiplication as a fully connected layer. Fig. 2(a) shows an example of the unfolding operation for a 64×3×3 filter. Here, we describe our first variable L, which controls how we decompose an operation. In each clock cycle, L-bit

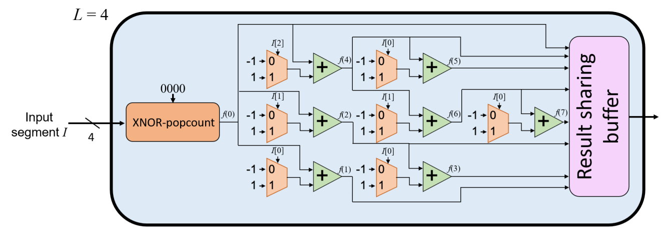


Fig. 4. An example of our PE architecture when L = 4.

XNOR operation and the succeeding popcount operation will be calculated in the PE. An L-bit segment in the 1-D weight vector will be considered as one partial-filter. Fig. 2(b) shows the decomposed result of the weight vector when $L = 4$. Here, we introduce our multiplexer-based similarity operation inspired by the following two observations. First, when $L = 4$, the number of the combinations of the partial-filter is $2^L = 16$. However, the total number of operations we need to perform for this segment is equal to the output channel size $= 64$, which is much larger than the combination of this partial-filter. Second, when two partial-filters have only one bit different, their XNOR-popcount results can be derived from each other, ie., when we obtain one result, the other result can be obtained by derivation. The following formula shows our multiplexer-based similarity operation:

$$f(y) = \begin{cases} f(x) + 1, & \text{if } I_i = 1 \\ f(x) + (-1), & \text{if } I_i = 0 \end{cases}$$

where $f(x)$ and $f(y)$ are the XNOR-popcount results of the partial-filter x and y, I_i is the input bit located at position i of the input segment I. As shown in Fig. 3(a), weight values in red circles in the partial-filter x and y are 0 and 1, respectively. After generating $f(x)$, we can infer $f(y)$ from $f(x)$ by evaluating I_i. With this method, we can obtain all the partial-results of the partial-filters with their corresponding input segments in one clock cycle. Moreover, motivated by the study [2], we utilize the concept of inverse filter to reduce the needed operations. Fig. 3(b) shows an example of one possible relationship between any two partial-filters when $L = 4$. In our PE design, we first perform an XNOR-popcount operation between the input segment and 0000 as the base for the derivations. Using the multiplexer-based similarity operations, we can derive the remaining $(2^L - 1)$ partial-results from the computed partial-results. Fig. 4 shows the corresponding architecture of our PE design for the multiplexer-based similarity operation. Our PE will generate $2^L/2$ partial-results in one clock cycle and store them in the $result\ sharing\ buffer$. Benefiting from reusing the results, we can further reduce the usage of hardware resources.

B. Average Redundancy Ratio

Since we will generate the partial-results from all the combinations of the partial-filter and their corresponding input segments, there must be redundant operations. Therefore, we introduce the *average redundancy ratio* to evaluate the number of redundant operations in our PE design as follows:

$$average\ redundancy\ ratio\ = \frac{|redundant\ operation|}{|operation|}$$

where $|redundant\ operation|$ is the number of redundant operations and $|operation|$ is the number of performed operations. Since we find that different layers with the same output channel size will have a similar number of redundant operations, the following discussion focuses on the combinations of the output channel size and L. By calculating the *average redundancy ratio* between various lengths of L for each output channel size, we can find the most appropriate length of L minimizing the number of redundant operations for each output channel size. We set the *average redundancy ratio* to be less than 0.1 in our design.

C. Accumulation Controller

The accumulation controller is used to realize the sharing of the partial-results generated from the PE. We utilize the accumulation index vector to indicate how the accumulation controller distributes the partial-results, which is illustrated in Fig. 5(a). Each partial-result from the result sharing buffer will pair with an accumulation index vector of length *output channel size* $\times$ 2. A set of two bits represents the distribution information. The left bit indicates whether the accumulation result of the current output channel needs this partial-result or not. The right bit indicates that the accumulation result needs the inverse result or the original result. The architecture of the accumulation controller is illustrated in Fig. 5(b). After receiving the partial-results from the PE, the weight distributor will distribute the result according to the accumulation index vector. The results will be added to the accumulation result buffer by the result accumulator.

D. Pruning Comparator

After generating the accumulation result of each output channel, a pruning comparator is used to further reduce redundant operations. When the current accumulation results of all the output channels can imply the binarized outputs, we skip the remaining operations and generate the outputs in advance. We use the pruning comparator to evaluate whether all the accumulation results meet one of the two pruning conditions as follows:

$Condition1 : Accumulation \geq T$. $Accumulation$ is the current accumulation result, and T is the threshold value of the corresponding output channel. Once the $Accumulation$ is greater than or equal to T during the operation, the binarized output can be set to 1 and skip the remaining operations.

$Condition2 : Accumulation + L * (Acc_t – Acc_c) < T$. Acc_t and Acc_c are the amount of partial-results needed to generate this binarized output and the amount of current accumulated partial-results, respectively, i.e., $Acc_t – Acc_c$ represents the amount of remaining partial-results. Since in each clock cycle, our PE calculates L-bit multiplexer-based similarity operation, the maximum value of each partial-result is L, and the maximum accumulation result of the remaining partial-results is $L * (Acc_t – Acc_c)$. Therefore, if the summation of the current accumulation result and the maximum accumulation result of the remaining partial-results is less than the threshold value T, we can set the binarized output to 0.

E. Overall Architecture

The architecture of our design is shown in Fig. 6. Data in the input feature map from the previous layer are streamed into our accelerator via the image data stream. Variable P is the number of parallel PEs in our accelerator. Both P and L are configurable to control the throughput and the hardware usage. When PEs generates all the combinations of partial-results, the accumulation controller will distribute and accumulate the partial-results. Next, the pruning comparator will

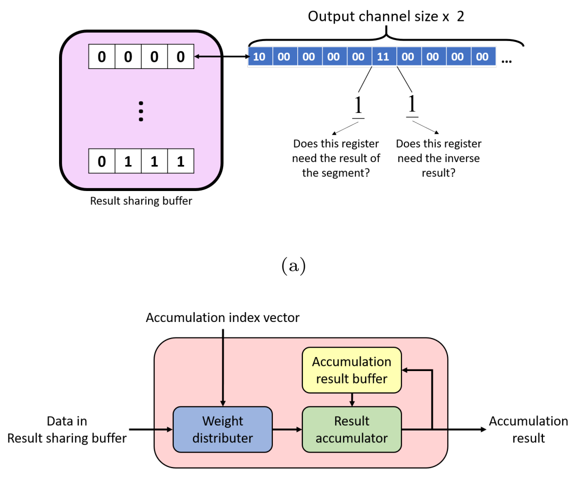

Output channel size x 2
0 0 0 0
0 1 1 1
Result sharing buffer
10 00 00 00 00 11 00 00 00 00 ...
1
1
Does this register need the result of the segment?
Does this register need the inverse result?
(a)
Accumulation index vector
Accumulation result buffer
Data in Result sharing buffer
Weight distributer
Result accumulator
Accumulation result
(b)

Fig. 5. (a) An illustration of accumulation index vector. (b) An illustration of accumulation controller.

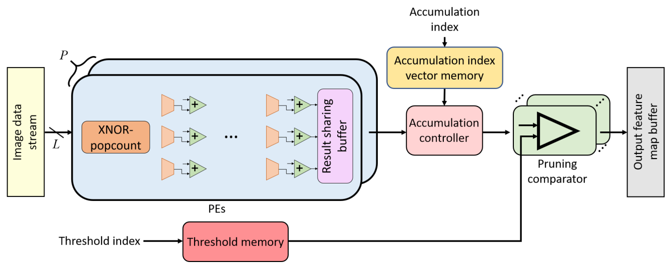


Fig. 6. The overall architecture of our proposed accelerator.

evaluate the current accumulation results based on the pruning conditions and skip the operations if the accumulation results satisfy one of the pruning conditions. The generated binarized outputs will then be stored in the output feature map buffer for the succeeding layers.

IV. Experimental Results

We use Xilinx ZCU104 FPGA to evaluate our proposed architecture. As for the network models and datasets, we use VGG-13 like model we trained for Tiny ImageNet. The number of LUTs, registers, and operation time are reported in Vivado HLx Editions 2020.2. Results show that our proposed architecture can reduce 99.7% operations for the network without any accuracy loss.

In TABLE I, we compare the hardware utilization and the inference time of the proposed architecture against the reimplemented [12] in the case of $L = 4$, $P = 2$ for each layer in Tiny ImageNet. Since the inputs of the first layer of Tiny ImageNet is floating point numbers and our architecture is based on binarized numbers, the implementations of this dataset will start from the LAYER2. The CONV refers to a convolutional layer, and the FC refers to a fully connected layer. According to TABLE I, our accelerator can achieve speedups very close to the output channel size of convolutional layers, and the output size of fully connected layers, while elevating hardware utilization. The reason for the speedup is that we can simultaneously accumulate the results of the number of the output channel size in one clock cycle.

TABLE I
Results Comparison Between Reimplemented [12] and Our Approach with $L = 4$, $P = 2$ on Tiny ImageNet.

LAYERi - type (Size)	Work	LUTs	Registers	Time(ns)	Speedup
LAYER2	[12]	125	52	69,120,010	-
CONV (64x60x60)	Ours	78	30	1,080,010	64.0
LAYER3	[12]	125	52	30,105,610	-
CONV (128x28x28)	Ours	78	30	235,210	128.0
LAYER4	[12]	125	52	50,878,474	-
CONV (128x26x26)	Ours	78	30	397,498	128.0
LAYER5	[12]	125	52	18,213,898	-
CONV (256x11x11)	Ours	78	30	71,158	256.0
LAYER6	[12]	125	52	24,136,714	-
CONV (256x9x9)	Ours	78	30	94,294	256.0
LAYER7	[12]	125	52	29,202,442	-
CONV (512x7x7)	Ours	78	30	57,046	511.9
LAYER8	[12]	125	52	29,644,810	-
CONV (512x5x5)	Ours	78	30	57.910	511.9
LAYER9	[12]	125	52	10,672,138	-
CONV (512x3x3)	Ours	78	30	20,854	511.8
LAYER10	[12]	125	52	1,185,802	-
CONV (512x1x1)	Ours	78	30	2,326	509.8
LAYER11	[12]	125	52	548,874	-
FC (512x2048)	Ours	78	30	278	1,974.4
LAYER12	[12]	125	52	530,442	-
FC (2048x512)	Ours	78	30	1,046	507.1
LAYER13	[12]	125	52	53,610	-
FC (512x200)	Ours	78	30	278	192.8

V. Conclusion

In this paper, we propose an accelerator design to reduce the operations of convolutional and fully connected layers in BNNs. We convert the original XNOR-popcount operation into a multiplexer-based similarity operation to reuse the computed partial-results. In addition, with the accumulation controller, we can accumulate the partial-results into the accumulation result buffer. Furthermore, the pruning comparator is used to lower the number of unnecessary operations. Experimental results show that our accelerator can achieve up to 1974 speedup and reduce operations by an average of 99.6%, without any accuracy loss.

References

[1] J. Bethge, *et al.*, “MeliusNet: Can binary neural networks achieve mobilenet-level accuracy?,“ in *arXiv:2001.05936*, 2020
[2] Y. Chang, *et al.*, “A convolutional result sharing approach for binarized neural network inference,“ in *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2020, pp. 780-785.
[3] T. Geng, *et al.*, “LP-BNN: Ultra-low-Latency BNN Inference with Layer Parallelism,“ in *International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, 2019, pp. 9-16.
[4] T. Geng, *et al.*, “O3BNN-R: An Out-of-Order Architecture for High-Performance and Regularized BNN Inference,“ in *IEEE Transactions on Parallel and Distributed Systems*, 2020, pp. 199-213.
[5] A. G. Howard, *et al.*, “MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications,“ in *arXiv:1704.04861*, 2017.
[6] M. Kim, P. Smaragdis, “Bitwise Neural Networks,“ in *CoRR*, abs/1601.06071, 2016.
[7] Q. Liu, *et al.*, “TCP-Net: Minimizing Operation Counts of Binarized Neural Network Inference“ in *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2021, pp. 1-5.
[8] Q. Liu, *et al.*, “An Efficient Channel-Aware Sparse Binarized Neural Networks Inference Accelerator,“ in *IEEE Transactions on Circuits and Systems II: Express Briefs*, 2022, Issue 3, 1637-1641.
[9] M. Rastegari, *et al.*, “XNOR-Net: ImageNet classification using binary convolutional neural networks,“ in *Proc. Eur. Conf. Comput. Vis.*, 2016, pp. 525–542.
[10] K. Simonyan, A.Zisserman, “Very Deep Convolutional Networks for Large-Scale Image Recognition“ in *International Conference on Learning Representations*, 2015.
[11] W. Tang, *et al.*, “How to train a compact binary neural network with high accuracy,” in *Proc. 31st AAAI Conf. Artif. Intell.*, 2017, pp. 2625-2631.
[12] Y. Umuroglu, *et al.*, “Finn: A framework for fast, scalable binarized neural network inference,“ in *Proc. ACM/SIGDA Int. Symp.Field-Programmable Gate Arrays*, 2017, pp. 65–74.