A Deep Learning Accelerator Based on a Streaming Architecture for Binary Neural Networks

QUANG HIEU VO, NGOC LINH LE, FAAIZ ASIM, LOK-WON KIM, AND CHOONG SEON HONG, (Senior Member, IEEE)
Department of Computer Science and Engineering, Kyung Hee University, Yongin 17104, Republic of Korea
Corresponding authors: Lok-Won Kim (lwk@khu.ac.kr) and Choong Seon Hong (cshong@khu.ac.kr)

This work was supported in part by the Grant from DEEPX; in part by the Technology Innovation Program or Industrial Strategic Technology Development Program (Memristor fault-aware neuromorphic system for 3D memristor array) under Grant 10085646; in part by the Kyung Hee University, in 2017, under Grant KHU-20170720, and in part by the National Research Foundation of Korea (NRF) Grant funded through the Korea government (MSIT) under Grant 2020R1A4A1018607.

ABSTRACT Deep neural networks (DNNs) have played an increasingly important role in various areas such as computer vision and voice recognition. While training and validation become gradually feasible with high-end general-purpose processors such as graphical processor units (GPU), high throughput inferences in embedded hardware platforms with low hardware resources and power consumption efficiency are still challenging. Binarized neural networks (BNNs) are emerging as a promising method to overcome these challenges by reducing bit widths of DNN data representations with many optimal prior solutions. However, accuracy degradation is a considerable problem of the BNN, compared to the same architecture with full precision, while the binary neural networks still contain significant redundancy for optimization. In this paper, to address the limitations, we implement a streaming accelerator architecture with three optimization techniques: pipelining-unrolling for streaming each layer, weight reuse for parallel computation, and MAC (multiplication-accumulation) compression. Our method first constructs streaming architecture by pipelining-unrolling method to maximize throughput. Next, the weight reuse method with the K-mean cluster is applied to reduce the complexity of the popcount operation. Finally, MAC compression reduces hardware resources used for remaining computation on MAC operations. The implemented hardware accelerator integrated into a state-of-the-art field programable gate array (FPGA) provides the maximum performance of the classification at 1531k frames per second with 98.4% accuracy for the MNIST dataset and 205K frame per second with 80.2% accuracy for the Cifar-10 dataset. Besides, the proposed design’s ratio FPS/LUTs is approximately 57 (MNIST) and 0.707 (Cifar-10), which is much lower than the state-of-the-art design with a comparable throughput and inference accuracy.

INDEX TERMS Binary neural networks, deep learning accelerators, FPGAs.

I. INTRODUCTION
In recent years, machine learning (ML) has become a popular terminology because of its applicability in a variety of fields. Particularly, deep neural networks (DNNs) have shown remarkable ability in classification tasks with high accuracy, such as computer vision [1] and speech recognition [2]. To adapt to more applications with more enormous datasets that need to have higher accuracy, researchers have tended to devise deeper networks that contain more parameters layers with bigger model sizes [3]. More complex DNNs directly affect power, resource efficiency, and throughput as many computations and memory requirements for model parameter storage increase. Notably, in a design implemented on FPGA or ASIC, a more significant number of logic gates are required to handle the increased number of operations that consequently incur higher energy consumption and lower processing performance during runtime. Meanwhile, data required for most of the larger DNNs may not be entirely stored in on-chip RAMs, which results in off-chip
Researchers have proposed many optimization strategies to improve DNNs performance [4], [5], and one among several approaches was to reduce calculation precision, which was proven to be containing potential redundant information [6]. Following some of the publications focusing on precision reduction, in 2016, Courbariaux et al. proposed a binarized training method that produces BNNs giving the best accuracy on most popular datasets [7]. This approach was rapidly deployed on most state-of-the-art network models because of its feasibility in addressing the challenges above. With binary precision, memory size was considerably reduced, while all multiplications are replaced by XNOR functions, which can dramatically shrink the computational size and save a vast amount of energy [8]. In addition, to further improve the design’s quality, in [9]–[12], the authors proposed other optimization techniques to reduce computation on Batchnorm, pooling, and adder tree operation by using threshold comparison, OR logic gates and counting “1” values, respectively.

Recently, two approaches have been used to deploy DNNs on hardware implementation, including BNNs. The first is single-layer architecture [13], [14], [15], in which one hardware block processes the network layer-by-layer one at a time. In particular, the design with this architecture is used to perform computation for all layers with the same shape at different times. Hence, after loading a specific number of input pixels, the next layer has to wait for the previous layer to finish before execution. Consequently, the hardware design needs to postpone loading new input pixels to process the loaded data. When increasing the number of layers, the total waiting time between two loaded input images increases, causing the throughput to be down. Regarding BNNs, most previous designs were implemented with this architecture [9]–[12]. In addition to independent optimization for every BNN as mentioned above, to optimize for only single-layer architecture when applying to BNN models, variable-width line buffer [9] or sliding window unit [6], [16], [17] were proposed to provide input data for computation effectively. Especially in [18], Cheng Fu et al. proposed weight-reuse and input-reuse methods that can reduce a considerable amount of computation on popcount operation, based on the similarity among lists of parameters or input pixels.

In general, the single-layer architecture would be the best option for enormous models with limited-resources devices. However, for the BNN implementations, this architecture may not utilize the full advantages of BNNs to enhance high throughput, resources, and power efficiency, compared to the streaming architecture’s second approach. More specifically, in a streaming architecture, because the whole network of a BNN model is fully implemented, computation on all layers is simultaneously performed, leading to high utilization. Moreover, the architecture is streaming, while the input is images, the output is the classification result, and no residual connection exists. Hence, input pixels are regularly loaded into the design without interruption. After a specific latency, the classification result is generated with a constant speed regardless of how many layers. In consequence, the throughput is significantly improved, compared to the single-layer architecture [19]. On the other hand, for BNN optimization, with streaming architecture, the BNN’s design can further optimize by replacing XNOR logic gates with NOT gates or directly connecting to the output. In doing so, the memory used to store parameters is eliminated, resulting in energy reduction for data delivery. In addition, compared to the single-layer architecture, another streaming architecture’s advantage is that the processed data is directly delivered from the previous to the next layer without any intermediate buffer or memory. However, development cost and flexibility may make implementation for the streaming architecture challenging, even impossible, especially for conventional neural networks. In BNN, although these problems become less critical when the kernel and feature map’s precision is reduced to one, they are still drawbacks for big model sizes. In this paper, to tackle mentioned problems and maintain the advantages from this kind of architecture, we target successfully implementing a streaming BNN hardware architecture on FPGA devices with high throughput, the best power and area efficiency, based on ratio FPS (frame per second)/power and FPS/LUTs. The main features of the proposed architecture are as follows.

1) An efficient pipeline-unrolling mechanism that maximizes utilization of all layers.
2) A combination of the weight reuse method [18] and K-mean [20], called MAC optimization, is applied to both convolution and fully connected layers to reduce the complexity of popcount operations.
3) A MAC compression method, that compresses the popcount tree with two options for adders: 6bit-adder [21], and 3bit-adder [22] compressor, without accuracy impact. Besides, the method allows more than one output channel to use the same popcount operation, which helps the design considerably reduce resources and energy, especially for large models.
4) An automated hardware implementation flow for the proposed accelerator that helps users facilitate hardware implementation for different topologies.

Compared to the previous work in [18], the proposed design (a streaming architecture) consumes more resources for the same BNN model. However, for practical applications, impacts of area issues in the proposed design are minimized via the above compression techniques. Meanwhile, with the advantages of the streaming characteristic, the proposed design provides better throughput, utilization, and efficiency in resources and energy. Especially, it is not as in [18], in the proposed design, with combination with K-mean cluster [20], the weight reuse method is directly applied without any additional resources and effects on throughput-latency.

The architecture capability is demonstrated by using the MNIST and Cifar-10 benchmark datasets. Compared to
previous best-published results, our proposed architecture consumes 3× fewer LUTs and eliminates FPGA Block RAM and DSP while producing approximately the same FPS/W with the same accuracy. This is the BNN hardware architecture with the highest power, area efficiency, and acceptable accuracy to the best of our knowledge.

The rest of the paper is organized as follows. Section II introduces a common background related to BNN and theoretical methods used to optimize BNN hardware implementation. Section III describes our proposed architecture. The process of generating RTL design, experimental results, and comparison among our design and other papers are presented in section IV. Finally, the conclusion is the last section.

II. BACKGROUND

A. BINARY NEURAL NETWORK

According to [7], BNN is also a type of neural network (NN) in which both the weights and activation outputs are constrained to −1 and +1 for negative and positive values, respectively. To transform real variables to these values, they used two different binarization functions. Firstly, the deterministic function:

\[
x^b = \text{Sign}(x) = \begin{cases} +1, & \text{if } x \geq 0 \\ -1, & \text{otherwise} \end{cases}
\]  

(1)

Secondly, stochastic function:

\[
x^b = \text{Sign}(x) = \begin{cases} +1, & \text{with probability } \rho = \sigma(x) \\ -1, & \text{with probability } 1 - \rho \end{cases}
\]  

(2)

where \(\sigma(x) = \max(0, \min(1, \frac{x + 1}{2}))\) and \(x^b\) is output of the function after binarizing. The deterministic function is mainly used for practical hardware implementations, while the stochastic function also can be used. In BNN, using the above binarize functions, all weights and outputs of both convolution and fully connected layers are reduced to a single bit before being used in the subsequent operations. Therefore, all the hardware-consuming multiplications can be replaced with XNOR logic. The adder tree used in the follow-up accumulation is also significantly simplified with a pop-count operation, as described in Figure 1. Even though the multiply-accumulate (MAC) operation holds most of the neural net workload, optimizing other cheaper operations such as batchnorm and max pool can still benefit the BNN performance. The applied techniques for each type of operation are described in the following subsections.

1) BATCHNORM OPERATION IN BNN

Unlike the MAC operation, the batchnorm function [23] uses floating-point parameters and complicated calculations, including division, root square, and multiplication. As usual, the batch normalized value of X is calculated as:

\[
Y = \frac{X - \mu}{\sqrt{\text{var} + \epsilon}} + \beta
\]

In which \(\epsilon\) is a small number to avoid round-off problem, \(\mu\) and var is mean, and variance of the train data batch, \(\gamma\), and \(\beta\) are constants learned from the training process. The normalized value is then binarized as:

\[
Z = \begin{cases} 1, & \text{if } Y \geq 0 \\ 0, & \text{if } Y \leq 0 \end{cases}
\]

The two steps of normalizing and binarizing can be combined into much simpler threshold comparing as following:

\[
Z = 1 \iff \frac{X - \mu}{\sqrt{\text{var} + \epsilon}} + \beta \geq 0
\]

If

\[
\text{sign}(\gamma) = \begin{cases} 1, & \text{if } \gamma > 0 \\ 0, & \text{if } \gamma < 0 \end{cases}
\]

the final formula can be described:

\[
Z = (X \geq (\frac{-\beta \sqrt{\text{var} + \epsilon}}{\gamma} + \mu) \text{ XNOR sign}(\gamma))
\]  

(3)

Hence, in hardware implementations, a combination of the batchnorm and binarization is now an output of a comparison and a successive XNOR gate.

2) MAX POOL OPERATION IN BNN

In BNN, after the batchnorm operation, some layers use the maxpooling operation to reduce input activation for successive layers. Theoretically, the output of maxpooling operation is binarized before being transferred to the next layer. By swapping the binarization and maxpooling module, the batchnorm and binary function can be combined, while retaining output results (Z). In addition, computing max values of the binary window is the same as finding outputs of the OR operation with binary value inputs. Eventually, there is a serial combination starting from the Batchnorm to the OR operation, as shown in Figure 2. From the hardware implementation point of view, the OR operation is simpler than the max operation in non-binary neural networks.

3) WEIGHT REUSE METHOD

As mentioned above, the BNN’s emergence with its particular characteristic has been considered as a promising solution in minimizing hardware resource and power consumption. In particular, the one bit-width in BNN models has been effectively utilized to maximize the parallel level of the data.
process and reduce computation effort by XNOR logic gates and pop-count operations. However, if we analyze the pattern of weight values, the current BNN models still have a considerable amount of redundancy requiring further optimization. Because of this, some computational operations and memory resources can be saved.

As introduced in [18], a binary number is used to express two states, “0” and “1”. Hence, when we pick a random binary number, the probability of zero or one is the same with 50%. If we have two random sets of N binary bits, we assume that we need more than K bits repeated on the second-bit set, compared to the first N bits. Thus, the probability happening this expectation is as the following equation:

$$P = 0.5^{N} \sum_{i=1}^{N-K} C_N^{K+i}$$

(4)

For example, if we need more than N/2 bits repeated in the second bit set (N is even),

$$P = 0.5^N \sum_{i=1}^{N/2} C_N^{N/2+i} = 0.5^N (2^{N/2} - 1) = 50\%$$

(5)

If K is smaller than N/2, we can get a larger probability. Therefore, the number of bits on the first set repeated respectively on the second set would be considerable. For a binary convolution layer, to calculate K output channels, each bit in a set of input values with $M \times M \times C$ binary bits XNORs with its corresponding bit in K different sets of $M \times M \times C$ binary kernel values, in which, M is window size, C is the number of input channels. Consequently, the number of similar bits between two arbitrary sets of kernel bits is considerable.

To take this advantage of a binary convolution for optimization, in [18], Cheng Fu et al. proposed a weight reuse method. This method executes calculating an output channel by implementing entire XNORs and pop-count operations for the corresponding set of $M \times M \times C$ binary weight values and again uses this output to calculate other outputs. To be easier in visualizing, we describe in more detail in this subsection with accompanying formulas. We assume that there are N different bits between two $M \times M \times C$ binary kernel bit sets. A $(A_1 A_2 A_3..A_N)$ for the first kernel set and B $(B_1 B_2 B_3..B_N)$ for the second kernel set, so $A_i = NOT(B_i)$ for all i from 1 to N. Call X $(X_1 X_2 X_3..X_{M \times M \times C})$ an unknown input feature map set, in which $(X_1 X_2 ... X_N)$ corresponds to the N different bits of two binary kernel bit sets. When we execute XNOR one random bit with 1 and 0, then sum these two outputs, the final result is always 1. Generalizing for N random bits, we have the following equation:

$$N = \sum_i^N Xnor(A_i; X_i) + \sum_i^N Xnor(B_i; X_i)$$

(6)

The left C ($(C_1 C_2 C_3..C_{M \times M \times C-N})$) kernel bits on two kernel sets are the same. Thus, the popcount output of $(A_C \ XNOR \ X)$ can be calculated as the following:

$$P_1 = \sum_i^N Xnor(A_i; X_i) + \sum_i^{N+1} Xnor(C_{1-N}; X_i)$$

(7)

For the popcount output of $(B_C \ XNOR \ X)$, based on $P_1$, we can transform as the following:

$$P_2 = \sum_i^N Xnor(B_i; X_i) + \sum_i^{N+1} Xnor(C_{1-N}; X_i)$$

$$= \sum_i^N Xnor(B_i; X_i) + P_1 - \sum_i^N Xnor(A_i; X_i)$$

$$= \sum_i^N Xnor(B_i; X_i) + P_1 - (N - \sum_i^N Xnor(B_i; X_i))$$

Finally, we have the equations using to calculate the $P_2$, based on the $P_1$ as the following:

$$P_2 = P_1 - N + 2 \sum_i^N Xnor(B_i; X_i)$$

(8)

According to [12], the output of the second channel is $2P_2 - M \times M \times C$. In this case, $P_2$ is calculated as in equation 8, thus the final output is described below:

$$O_2 = 2(P_1 - N + 2 \sum_i^N Xnor(B_i; X_i)) - M \times M \times C$$

(9)

There is a quite similar feature for the first convolution layer, which uses full precision input pixels. In doing so, the result is zero for the sum of the two multiplications: Ax-1 and Ax1 (A is an arbitrary full precision pixel). So, regards the number of different bits between the two sets of kernel bits and the sum of D multiplications between the D arbitrary full-precision input pixels and the corresponding D different bits of the second channel as D and $S_2$, respectively. We can calculate the output of the second channel as the following equation:

$$O_2 = O_1 - (0 - S_2) + S_2 = O_1 + 2S_2$$

(10)

In this way, we do not need to implement whole convolution operations, which leads to considerable hardware resource reduction while guaranteeing output accuracy.
III. PROPOSED ARCHITECTURE

A. HIGH-LEVEL STREAMING ARCHITECTURE

Figure 3 shows a generic high-level system architecture of the proposed BNN architecture. The BNN design is connected to a DDR memory via a Direct Memory Access (DMA) IP using the AXI-4 Stream bus. Two fixed memory areas are used to store input images and output classification from the BNN design in the DDR memory. The DMA block provides the initial configuration’s start address and data length of these two memory areas. Each input pixel from the input memory area is sequentially delivered to the streaming BNN architecture. After a particular latency, output classification is back to the second DDR memory area. Depending on the design’s unrolling levels, which would discuss in the next section, the bandwidth of the streaming data bus can be changed to satisfy the requirements.

In contrast with the single-layer accelerator, where a single hardware accelerator processes a neural network layer by layer sequentially, at the coarse-grained level, we propose a multi-layer streaming architecture, which implements the whole network architecture on a hardware implementation. In particular, as the idea in [24] proposed by Kim et al. on Boltzmann Machine ANN, the streaming architecture in this paper is implemented in a coarse grain pipeline manner. This approach is achieved by dividing an inference workload into a layer granularity, in which the number of pipeline stages is equal to the number of layers. This way, input pixels are received continuously, and every layer can be simultaneously processed, resulting in the highest performance. Additionally, because the output of a previous layer is directly delivered to the next, without any intermediate storage, the proposed accelerator can mitigate propagation delay and reduce a significant amount of memory.

On the other hand, since all layers are fully implemented and input data is continuously processed, increasing the number of layers causes a longer initial latency. It does not affect the throughput of the steady-state system time. For example, suppose the dimension of a primary input frame is $E \times F$, and input data is delivered into the design every clock cycle. In that case, the accelerator requires $E \times F$ clock cycles to classify the frame after the initial filling time. Consequently, the throughput of the design can be flexibly changed based on the number of input pixels loaded into the design on each clock cycle.

In detail, each layer is also constructed under the pipelining technique. A convolution layer is divided into four more minor phases; the first phase is multiplication operation, the second phase is the pop-count, the third phase is the normalization, and the final phase is the binarizing. If the next is a pooling layer, the data from the binarizing phase can be delivered directly into the pooling process.

Figure 4. Fully conv-batchnorm-pooling layer.

In particular, as shown in Figure 4, a comprehensive layer including the convolution line buffer, multiplication-popcount, batchnorm, and pooling layer is described. This proposed architecture is a complete combination of a typical convolution and pooling layer. In doing so, we only need to store threshold values for batchnorm operations. Meanwhile, the memory used for storing weight values and $\text{sign}(\gamma)$ of the batchnorm operation is totally removed. Hence, the working process is enabled without delay in parameter delivery. Firstly, to finish a process from the input to the output of a layer, input pixels from the previous layer are carried to the convolution line buffer. When a specific number of input data are loaded into the line buffer, all window values are generated and shifted into the “Conv $W \times Wb$” block as inputs of the multiplication-popcount module. Output values from the “Conv $W \times Wb$” block are compared with a threshold parameter at the batchnorm block. Then they are processed with a suitable operation depending on the sign of the batchnorm weights ($\gamma$), which is detailed in subsection B. Finally, if there is no pooling layer, outputs of the batchnorm are directly carried to the next convolution layer. By contrast, data is pushed into the pooling line buffer. When the number of values loaded into the line buffer reaches an acceptable condition described in the following subsection, output windows are produced. Data is transformed and delivered to the next layer through the OR operations.

B. MICROARCHITECTURE FOR WINDOW GENERATION INTO CONVOLUTION AND POOLING LAYER

This subsection describes the details of a convolution and pooling layer architecture. In addition, the efficiency of the combination among pipelining, unrolling method, and popcount optimization in each layer is mentioned as well.
To facilitate the explanation, as shown in Figure 5, we regard ExF as the dimension of the input feature map; MxM as the dimension of filters; C as the number of input channels; and K as the number of output channels.

As we know, in convolution, windows including input feature map data with size M×M are multiplied by the corresponding filters with the same size before the adder tree, in which M is 3 in the proposed design. On the other hand, in pooling layers, pooling windows with size 2×2 are also needed to be provided for the pooling process. To provide these windows for the next processing, we design a lightweight shift-register-based line buffer to store only the needed data streamed from the previous layer. In this way, the hardware implementation for a specific layer only waits for the necessary input data to start its operation instead of waiting until the previous layer to ultimately produce the complete necessary input data to start its operation instead of waiting until the previous layer to ultimately produce the complete feature map. Besides, each layer does not need to wait to complete the previous layer; layer-level pipeline stages can be overlapped, thus significantly reducing the initial pipeline filling time.

**Algorithm 1 Convolution Line Buffer Pseudocode**

**Input:** Activation output of previous layer.

**Output:** Window (W) with size: MxM

1. for ix = 0 to M-1 do
2.  for iy = 0 to M-1 do
3.   for i = 0 to N-1 do
4.     y = ry + iy - (M/2)
5.     x = rx + ix + i - (M/2)
6.     if x < 0 or x ≥ W then
7.       W[ix, iy] = padding_value
8.     else
9.       W[ix, iy] = L[(M-1-iy)E+(M-1-ix)+i]
10. end if
11. end for
12. end for
13. end for

In particular, two types of line buffers are implemented in the proposed accelerator: convolution line buffer (CLB) and pooling line buffer (PLB). Firstly, the CLB stores a specific number of input pixels from the previous layer and then sequentially provides convolution windows for the multiplication operation. Usually, the CLB includes (M-1)×E+M pipeline registers if the number of generated windows is only one. Meanwhile, as shown in Figure 6, the size of CLB is (M-1)×E+M+N-1 if the number of generated parallel windows increases to N every clock cycle. In the multiplication-popcount process, the number of calculations would be N times of generating one window. In terms of working mechanism, when the number of filled registers on the CLB is equal to E×(M−⌊M/2⌋−1)+⌊M/2⌋+N, the CLB starts selecting appropriate values for the output windows via their coordinates on the CLB and feature map, then pops them out and simultaneously asserts a corresponding output valid signal (out_valid). For the case with N > 1, the CLB needs to receive new N input values continuously within the same period to generate N windows continuously for each clock cycle. If the input value is from the previous layer, which is also received N input values, the requirement would be solved. All layers will eliminate the issue for the first convolution layer if the memory can provide it with N input pixels at each clock cycle. The detailed process is illustrated in algorithm 1 with the following denotations: rx, ry is the coordinate of the center of window on the frame; ix, iy is the coordinate of each pixel on the window, and L is the convolution line buffer with length (M−1)×E+M+N−1.

In order to make the Algorithm 1 clear, the paper provides a visual description about line buffer situations when the CLB starts generating the output with one window (Figure 7) and when the CLB is at the middle of a frame transmission (Figure 8), in which M = 3 and E = F = 28 for the MNIST dataset, E = F = 32 for the Cifar-10 dataset.

The second type of line buffer is the pooling line buffer (PLB). The PLB is considered an intermediate bridge between the batchnorm and the pooling operation. Output data from the batchnorm is gradually pushed into the PLB, while generated windows from the PLB are pushed into the pooling operation. Compared to the CLB, the PLB does not consider border padding, and the valid signal is only asserted at the even position in the input feature map as shown in Figure 9. Typically, suppose the window size is 2 × 2, and...
the parallel output level is only N/2. That means hardware resources responsible for the next multiplication-popcount operation are also reduced a half, compared to the previous multiplication-popcount operation.

Regarding the PLB size, if the PLB outputs N/2 windows corresponding N parallel inputs, the line buffer size is E + N registers. Hence, depending on the number of parallel input values provided from the previous layer, the size of the PLB may (N/2 > 1) or may not (N/2 = 1) need to increase when compared to providing single input data. Besides, it is worth noting that the number of parallel input pixels must be a power of 2 and a divisor of E. For instance, if the pooling window size is 2 × 2, and the number of parallel inputs from the previous layer is N = 2, then the PLB size is E + 2 registers and produces 1 pooling window for each clock cycle yielding twice the throughput, compared to N = 1. Meanwhile, if the previous layer provides N = 4 parallel inputs, the PLB size is E+4 registers and produces 2 windows after each clock cycle with four times throughput than N = 1.

In terms of the working mechanism, the PLB starts generating pooling windows when the PLB is entirely filled by valid data. In particular, the latency is (E + N)/N clock cycles from the input valid signal is asserted. The detailed process is described in Algorithm 2, accompanied by Figure 9 that shows the line buffer operation from the beginning to the end with N = 1.

Algorithm 2 Pooling Line Buffer Pseudocode

Input: Sequential pixel chain from output of the Batchnorm.
Output: window (W) with size: 2 × 2

1: if (x+1 : N) and y % 2 then
2: for i = 0 to N/2-1 do
3: W(i)(0,0) = L[E+1+2i]
4: W(i)(0,1) = L[E+2i]
5: W(i)(1,0) = L[1+2i]
6: W(i)(1,1) = L[0+2i]
7: end for
8: end if

C. MICRO-ARCHITECTURE FOR MAC OPERATION AND BATCHNORM

After the mentioned windows are generated via the lightweight shift-register-based line buffers, the data needs to go through a series of computations, including the multiplication operation, pop-count for summation, and batchnorm (normalization) operation, before transferring to the next layer. At any degree of parallelism, this process is always considered as the stage causing the most critical timing paths and consuming most of the energy in the design. This problem consequently leads to limiting the operating clock frequency and system throughput. In our architecture, all data paths of this process are optimized at the highest level to mitigate this harmful influence, and every unnecessary computation is
eliminated to avoid processing delay and power consumption. Firstly, since all the weight values are constants in multiplication operations, we can replace the XNOR gates with NOT gates when the weight value is zero and directly connect the input to pop-count when the weight value is one. Similarly, in the batchnorm operation, the sign ($\gamma$) value determines whether it uses the NOT gate or not instead of the XNOR gate. In doing so, we reduce the considerable amount of latency by using the XNOR gate and also minimize the memory resource.

Secondly, parallelism, pipelining technique, and weight reuse optimization are implemented in the proposed design to process binary neural networks efficiently. To facilitate how the method works in the design, the details are explicitly explained as the following.

1) PIPELINING AND PARALLELISM MECHANISM

One factor that determines the throughput of a processor is the maximum clock frequency, while critical timing paths in a design limit the clock frequency. In the proposed BNN architecture, critical timing issues are in the paths from matrix multiplications to the batchnorm. However, these paths can easily be broken down into shorter paths by adding intermediate registers. The system throughput can be significantly improved at the expense of a few initial latency cycles. Based on the requirement (frequency, area, power) and parameter inputs (number of channels, input bitwidth), we can locate registers to good points so that combinational logic delay paths become shorter than or equal to the target clock period at the expense of additional latency. Our experimental design inserts one pipelining stage to the output of the multiplication operations, along with the popcount operation, and at the output of the batchnorm layer, which is described in Figure 10.

Regarding the parallelism technique, massively parallel computing typically helps increase the total system throughput in hardware implementations. However, the trade-offs are i) a considerable amount of computing hardware is needed, which leads to higher power consumption, and ii) substantial fan-outs increase congestion and may eventually render the design unrouteable. In BNN, since the precision of both weight and feature map data are reduced to a single bit, many loops in the convolution operation can be completely unrolled without massively increasing hardware resource requirements and routing congestion. Total six loops used in the convolution computation are described in Figure 11. In this proposed design, we choose to unroll every inner loop from the 3rd loop to the 6th loop and optionally unroll loop 2.

Firstly, unrolling loops 3 to 6 across all layers offers a balance on the rate between data consuming and producing in each layer accelerator; hence the technique allows them to operate continuously without idle time at later layer or back-pressure to the previous layer, regardless of filter size and the number of kernels in different layers. In particular, by unrolling all these loops, all input windows generated from the line buffer on each layer are executed simultaneously, leading to a significant propagation delay reduction. In addition to the loop 3 to 6, we also unroll the loop 2 to investigate the hardware utilization among different throughput requirements. Unrolling the loop 2 is achieved by increasing the number of generated windows from the line buffer each clock cycle and duplicating the computation-MAC block sequence to handle more spatial locations in parallel. That means all generating windows from the CLB are simultaneously executed through the corresponding sequence of the computation-MAC block.

2) MAC OPTIMIZATION

As mentioned in the section II, the weight reuse method is proved to be substantially effective in optimizing pop-count
case, the number of Flip-flops which is used to synchronize is a Hamiltonian graph is applied for K output channels. In that initial latency and hardware resources. For example, suppose the first channel. Consequently, many registers must be added depending on the output of the previous channel, except for using the Hamiltonian path makes each channel’s output connected graph with N vertices. Recently, two approaches example, we seek the shortest Hamiltonian path for a fully implemented. This paper uses the K-mean cluster to find the shortest Hamiltonian path. Hence, the more output channels, the more fully hardware implemented output channels that require more operations, resulting in high hardware resources and power consumption.

In this paper, we also set up a similar graph. Each set of $M \times M \times C$ binary weight values represents a vertex, and the number of different bits between two sets is considered the distance of the edge connecting two vertices. To partition the graph, we used the K-mean cluster algorithm for every $R$ from 1 to $K$ (the number of output channels) [27]. The optimal R-value gives us the smallest number of binary bits used to calculate results for all output channels. That means all repeated binary bits compared with corresponding centroids are eliminated. Figure 12 b) is an intuitive diagram describing the proposed method with the K mean cluster, and the following equation is the exact formula for finding the optimal R-value.

$$R = \arg\min_{R} \sum_{i=1}^{R} \sum_{j=1}^{m_i} \text{Dist}_{ij} + R \times C \times M \times M$$

(11)

where $R$ is the number of sub-graphs, $m_i$ is the number of vertices possessing the same centroid $i^{th}$, Dist$_{ij}$ is the distance connecting the centroid $i^{th}$ and vertex $j$, $R \times C \times M \times M$ is the total number of bits on R output channels which are fully implemented. This paper uses the K-mean cluster to find R groups of vertices and their corresponding centroids.

In theory, the output of the K means cluster algorithm includes R sub-graphs and R corresponding centroids based on R initial centroids and coordinates of all vertices. Firstly, each vertex is clustered into a group where the distance from that vertex to the group’s centroid is the shortest. In second step, in each group of vertices, new centroid of the group is selected based on the following formula $M_i = \sum_{j=1}^{m_i} x_j / m_i$, where $M_i$ and $x_j$ are coordinates of the centroid and $j^{th}$ vertex of the group $i^{th}$, respectively. These two steps are have been applied to find the shortest Hamiltonian cycle [25], which can be used to find the shortest Hamiltonian path. The first one is “exact solution,” which can be used to find precisely the shortest Hamiltonian cycle by reducing the number of searched Hamiltonian cycles. However, this approach still takes a long time and effort to calculate the final result, especially with large graphs. Typically, when increasing the number of vertices, the processing time can increase exponentially with branch and bound method in [26], or increase with a ratio $N^2 2^N$ with dynamic programming method in [25]. The second one is approximation algorithms being more common for massive graphs, but they may produce poorer quality results than previous algorithms.

To mitigate the above problem, Cheng Fu et al. proposed a partitioning approach, which splits the whole graph into a specific number of sub-graphs, in which the current number of vertices in a sub-graph is limited to 64. However, this way increases the number of fully implemented output channels in hardware design. Besides, the number of sub-graphs depends on the limited number of vertices in a sub-graph (64) to feasibly find the shortest Hamiltonian path. Hence, the more output channels, the more fully hardware implemented output channels that require more operations, resulting in high hardware resources and power consumption.

On the other hand, there is currently no practical algorithm to find the shortest Hamiltonian graph, especially it is more challenging if the number of vertices is considerable. For example, we seek the shortest Hamiltonian path for a fully connected graph with N vertices. Recently, two approaches operation. In [18], Cheng Fu et al. utilized this method effectively by using graph partitioning and finding Hamiltonian shortest path algorithm for each sub-graph. For the Hamiltonian shortest path, this is a good way to increase the number of weight reused operations. However, for convolution layers including many channels, this technique may cause the number of required Flip-flops and latency to drastically increase if the method is applied to streaming architectures. Typically, using the Hamiltonian path makes each channel’s output depending on the output of the previous channel, except for the first channel. Consequently, many registers must be added to be synchronized with the next layer, leading to increased initial latency and hardware resources. For example, suppose a Hamiltonian graph is applied for K output channels. In that case, the number of Flip-flops which is used to synchronize is $K \times \lceil \frac{N}{m} - 1 \rceil \times$ bitwidth, in which m is the number of output channels is calculated in the same clock period, and bitwidth is data width used for storing output popcount operation. The Figure 12 a) describes clearly the problem.

In theory, the output of the K means cluster algorithm includes R sub-graphs and R corresponding centroids based on R initial centroids and coordinates of all vertices. Firstly, each vertex is clustered into a group where the distance from that vertex to the group’s centroid is the shortest. In second step, in each group of vertices, new centroid of the group is selected based on the following formula $M_i = \sum_{j=1}^{m_i} x_j / m_i$, where $M_i$ and $x_j$ are coordinates of the centroid and $j^{th}$ vertex of the group $i^{th}$, respectively. These two steps are
repeated until the sum of all distances from all vertices to their centroids becomes minimum. However, we know only the distance between any two vertices according to our consideration. Thus, the second step performs finding the vertex having the shortest sum of the distance from it to all vertices in the group.

On the other hand, the K-mean cluster has a limitation; different R initial centroids give a different partitioning way. Meanwhile, running all different R initial centroids and every R-value from 1 to K (the number of output channels) consumes a massive amount of time, especially for layers having an enormous number of output channels. For example, if a layer has K output channels, the total number of cases that we need to explore:

$$\text{Number of cases} = \sum_{i=1}^{K-1} C^i_K = 2^K - 2 \quad (12)$$

To reduce the number of cases, we used K-mean++ [27] for initializing the first R centroids if the number of search cases is more than 100,000. In addition, to make the output result more correct, all cases of the first centroid are executed, and then the optimal value is selected instead of randomizing the first centroid. Hence, if one layer has K output channels and the number of the cluster varies from 1 to K, the total of cases we execute is $K^2$, which is much smaller than the number of cases using the basic K-mean algorithm ($2^K - 2$) when $K \geq 5$.

3) MAC COMPRESSION

To further optimize the MAC operation, which consumes most hardware resources and power consumption in any BNN architecture, as shown in Figure 16 and Figure 15, we propose applying the two well-known techniques used in the popcount compression. Firstly, as described in Figure 13, the 6:3 compression [21] is applied to reduce the number LUTs by using only the adder 6:3 type to implement the adder tree. Each bit of the output result from the LSB (low significant bit) to the MSB (most significant bit) is sequentially calculated with adder 6:3. Two carry bits from the adder 6:3 are stored to calculate more significant output bits. In doing so, we can avoid using adders with too many bit inputs, leading to hardware resource reduction. Similarly, in the automated hardware implementation flow, 3:2 compression [22] using adder 3:2 type is also provided to implement adder tree as shown in Figure 14.

| TABLE 1. Hardware resource and power consumption when implementing adder tree with three options: non-compression, compress 3:2, and compress 6:3. |
|------------------|------------------|---------------|-------------|
|                  | Num of window =1 | LUTs          | FFs         | Power       |
| MNIST (100Mhz)   | Non-compress     | 10,527        | 5,923       | 0.428       |
|                  | Compress 3:2     | 9,740         | 5,723       | 0.413       |
|                  | Compress 6:3     | 10,310        | 5,720       | 0.399       |
| CIFAR-10 (50Mhz) | Non-compress     | 311,546       | 38,571      | 6.598       |
|                  | Compress 3:2     | 281,861       | 38,566      | 6.256       |
|                  | Compress 6:3     | 290,600       | 38,530      | 5.837       |

Depend on the input’s bitwidth of the popcount operation, compression 3:2 and compression 6:3 can be selected to implement the adder tree effectively. According to Table 1, in our experiments, the compression 3:2 uses fewer resources (reduce 7.5% LUTs for MNIST model, and 9.5% LUTs for Cifar-10), while the compression 6:3 uses less power consumption for both software models (6.7% for MNIST and 11.5% for Cifar-10).

In this subsection, we continue proposing another technique, called popcount reuse, to reduce hardware resources significantly. More specifically, as shown in Figure 17,
without the popcount reuse method, K popcount operations are implemented for K output channels, while with the popcount reuse, the number of popcount operations can be reduced X times, where X is the number of output channels using the same popcount operation. To maintain continuity of the streaming architecture, the clock source used for popcount operation is X times faster than the clock source used for others. The value of X is based on the required performance and area. When increasing X value, hardware overhead and performance decrease. By contrast, decreasing X, hardware overhead, and performance increase.

D. OPERATION TIME OF ARCHITECTURE

As discussed in subsection 3.1, the proposed design is a coarse-grained pipeline architecture with the number of pipeline stages equaling the number of layers. By overlapping these stages, throughput and initial pipeline filling time are critically improved. In particular, the convolution line buffer of a specific layer starts generating window values after \(N_f = E \times (M - \lfloor \frac{M}{2} \rfloor - 1) + \lfloor \frac{M}{2} \rfloor + N\) clock cycles. Besides, by applying the loop unrolling and pipelining mechanism, the multiplication-popcount module requires a typical number of clock cycles \(N_p\). Depending on the number of output channels and input bitwidth, the number of clock cycles can be changed to have the highest frequency and match timing constraints. Thus, as shown in Figure 18, after \(N_f + N_p\) clock cycles later than the start time of a specific layer, the next layer operation can start.

Each pooling layer needs \(E + N\) clock cycles to generate outputs (in the case that window size is \(2 \times 2\) and the number of parallel inputs is \(N\), so the next convolution layer needs to wait for \(E + N\) cycles later than the current pooling layer. In terms of the fully connected layer, we also need \(N_{fc}\) clock cycles from receiving the first data to generating the first temporal maximum value of ten output channels. This number of clock cycles can be flexible changed, depending on the required frequency. In our experimental model, with the frequency being 300 Mhz, the fully connected layer needs 3 clock cycles to find the maximum value from 10 temporal output values.

Because input data is continuously filled into the proposed hardware accelerator for BNN, after the initial time latency, each convolution, pooling, and fully connected layer requires \(E \times F\) clock cycles to process one inference operation if the loop 2 does not unroll. For the loop 2 unrolling case, the number of required clock cycles can be reduced by \(N\) times (\(N\) is the number of windows of the convolution line buffers). Eventually, the model needs \(E \times F / N\) clock cycles to finish one classification for one input image.

IV. EXPERIMENTAL RESULT

A. BNN MODEL EXPLORATION

To explore potential model architecture space and get the optimal BNN model, some training conditions are required for all models. In particular, the batchnorm operation is added after each convolution layer, and the maxpooling layer follows batchnorm operations from the second convolution layer. For the model using the MNIST dataset [28], binary search is further used to minimize the number of channels on each layer. The initial inputs are as the following for the training on the MNIST dataset [28]:

---

**FIGURE 18.** Processing time line of the proposed architecture.
1) Range for the number of layers of the BNN model: 
   \[ L = \{3, 4, 5\} \]
2) Maximum number of channels per layer: \( C_i \leq 50 \)
3) Target accuracy threshold.

Based on these inputs, for each L value, we use binary search to reduce the number of channels of each layer uniformly until a BNN model with a minimum number of channels for all layers is found. Next, based on this model, the binary search continues to be used in minimizing the number of channels for each particular layer. Eventually, the optimal BNN model corresponding to a particular L value is determined. Each model has a variable number of layers represented by the elements of set L. Therefore, the number of output models is represented by the size of set L. Besides, because in each initial BNN model, only the number of channels for each layer is optimized, we independently redefine all the components of our network architecture to reduce our search space. Regarding the training environment, we found a productive optimizer method that Adam optimizer is used for the first 30 epochs and SGD optimizer for the remaining 70 epochs. We set the learning rate at 0.03 and momentum at 0.5.

For the model using the Cifar-10 dataset [29], based on the model structure in [6], we change some training conditions to make it compatible with our proposed hardware architecture. In particular, we add padding with −1 value for each convolution layer to improve accuracy with a smaller number of channels. In addition, the output feature map of the last convolution layer is guaranteed with 1 × 1 dimension, which allows applying the MAC optimization method on all fully connected layers. For the training environment, Adam optimizer is also used with 500 epochs. The learning rate is 0.005 for the first 40 epochs, 0.0001 from the 80th epoch, 5e-05 from the 120th epoch, and 5e-06 from the 160th epoch.

In this paper, we aim to find two models for the MNIST dataset using the aforementioned approach, one model for Cifar-10 dataset [29], and use them to demonstrate the efficiency of the proposed architecture. Firstly, for the MNIST dataset, to compare with previous work, in the first model optimization for MNIST, we try to minimize the BNN model in terms of hardware implementation when we set the target accuracy to above 98.4% (because other implementations in previous work support around 98.4%). We call this model MD1. On the other hand, to find a highly efficient BNN model in hardware implementation with reasonable accuracy, we explored many BNN models with different configurations and found an efficient model having 97.7% accuracy. We call this second model MD2. After doing the architecture search, two series of optimal models are found for two respective accuracy thresholds 98.4% and 97.7%.

In terms of the list models for 98.4%, according to Table 2, the model with 3 convolution layers has the lowest number of layers, leading to the lowest inference latency compared to other models having the same accuracy. In addition, this 3-layer model gives the best result in terms of hardware resources. Therefore, this model is chosen and regarded as the MD1 model. Similarly, we also found the MD2 model by exploring many candidates having similar accuracy with consideration of required hardware resources and corresponding accuracy. In summary, both models have three convolution layers and 1 fully connected layer. The MD1 model has 26 channels for the first convolution layer, 24 channels for the second layer, and 31 channels for the last convolution layer. The MD2 model has 17 channels for the first convolution layer, 15 channels for the second layer, and 22 channels for the last convolution layer. Each convolution layer on both models is followed by a batchnorm function, and max pooling is applied to the last two convolution layers. Finally, as discussed in the background section, both models use 16-bit fixed-point input pixels and binary weight for the first convolution, while both weights and input feature maps are binarized from the second layers. Secondly, for Cifar-10, we get a model with an accuracy of 80.2%, where there are 6 convolution layers (64, 96, 96, 128, 192), and 2 fully connected layers (256, 10) at the end. Batchnorm operation is attached after each layer, and max pooling is added after batchnorm operation from the second to the final convolution layer.

| Layer | Output channels |
|-------|-----------------|
|       | 5 Layers | 4 Layers | 3 Layers |
| 1     | 19       | 24       | 26       |
| 2     | 20       | 22       | 24       |
| 3     | 20       | 23       | 31       |
| 4     | 20       | 22       |          |
| 5     | 19       |          |          |
| LUTs  | 19,954   | 21737    | 19211    |
| Flip-flops | 10026 | 9830     | 9104     |

B. AUTOMATED HARDWARE IMPLEMENTATION AND VERIFICATION PROCESS

Designing each hardware accelerator for each model is an extremely time and effort-consuming and error-prone process. Based on the idea used in [30], this paper proposes an automated hardware implementation framework to automate the generation of hardware architecture in Register Transfer Level (RTL) based on user constraints for the BNN model: a script automatically generates RTL designs according to user-constraints. In particular, all parameters are separated into two sets: module parameters and general parameters, while the proposed hardware architecture includes hardware modules specialized for typical functions such as the batchnorm, convolution line buffer, multiplications+popcount, maxpooling line buffer, and pooling.

To regenerate RTL designs, firstly, the script uses the general parameter set and finally generates the corresponding RTL designs. During the regeneration process, the script uses the general parameter set and finally generates the corresponding RTL designs.
Coarse-grained architecture, in which an overview of design is determined by the number of particular modules and their position on the design. Secondly, module parameter sets are used to configure all input module parameters of each typical module at a specific position on the architecture. Finally, the script automatically generates the entire RTL design by connecting all configured hardware modules. In Figure 19, all parameters in each module are described, and the general parameter set is provided as well.

As shown in Figure 20, to verify the operating equivalence of the implemented hardware accelerators with the BNN models in software implementation, we have verified multiple accelerators based on the proposed architecture for various BNN models with various layers, channels, and accuracy. Firstly, C/C++ models are constructed based on the parameters and model structure from the PyTorch models. In particular, each layer in the python model is built as a C/C++ function. The output of each layer is compared respectively between the C/C++ model and the PyTorch model. After having a C/C++ model, we prepare a series of C/C++ models corresponding to different numbers of channels and numbers of layers. Secondly, we implement the respective hardware accelerators by the automatic script. Then, the VCS simulation tool from Synopsys is used to precisely verify the waveform of each data path by comparing the result with the corresponding C/C++ models. Finally, the implemented accelerators are ported into an FPGA and verified the operation of the hardware accelerators with the C/C++ models. VCS simulation results have been verified bit by bit in the data path through the Integrated Logic Analyzer (ILA) provided by Xilinx FPGA [31]. By using this automated process, after training, the hardware accelerator corresponding to the updated software model can be immediately implemented, which is highly desirable for practical applications on FPGA. Consequently, manual efforts are eliminated from the hardware design stage to the verification stage for target applications.

C. EXPERIMENTAL IMPLEMENTATION

We use the automated script to generate RTL descriptions based on the proposed hardware architecture, input BNN model structure, and user-specified design parameters to evaluate all model functionality. Regarding the hardware device, the proposed architecture is implemented on Xilinx’s Ultra96 evaluation board, which includes an xczu3ev-svbva484-1-e Ultrascale+ MPSoC. In particular, there is a quad-core Arm Cortex-A53 application processing unit and a dual-core Arm Cortex-R5 real-time processing unit on the Process Subsystem (PS). The programmable logic (PL) part comprises 141,120 flip-flops, 70,560 look-up tables (LUTs), 360 DSP slices and 7.6 Mbits block ram. In addition, xczu19eg-fvb1517 part including 522,720 LUTs and 1,045,440 FFs is used to implement for bigger model.

As mentioned in the previous subsection, we use the Synopsys VCS to perform simulations for the generated RTL designs and then compare the classification outputs with the output of a bit-true C++ model for the input BNN models. At the next step, we synthesize and implement the proposed design by using Vivado 2018.3. All experiment results describing the number of LUTS, Flip-flops, estimated power consumption are collected from Vivado’s report. Notably, to estimate the power efficiency on the BNN core, we only
collect power consumption on the PL part of the chip, consisting of FPGA Logic gates. We compare the software-based models and the implemented hardware accelerator bit by bit, while the FPGA bitstream’s functionality is wholly verified against the dataset’s 10,000 images. On the PS side, the host C code running on the ARM processor has two tasks. First, it configures and executes the Direct Memory Access (DMA) to transfer test images frame-by-frame from DRAM to the hardware accelerators and transfer back classification results from the hardware accelerators to DRAM. Then after the last result arrives, it compares all received classification results with pre-known outputs from the C/C++ model. In Figure 21, the comprehensive system architecture is described, in which PS connects to BNN core via DMA. A full AXI-4 bus is used for data transmission between the DDR controller and DMA, while the processor manages DMA’s transmission through the AXI-Lite bus.

$$N = 2$$ and $$N = 4$$, frame rate increases to $$7.65 \times 10^5$$ and $$1.53 \times 10^6$$, respectively. Meanwhile, based on the result in Table 3, in case $$N > 1$$, hardware resource and power consumption increase much less than $$N$$ times, compared to $$N = 1$$. Hence, both throughput-over-power and throughput-over-area efficiency are considerably improved. Typically, the number of Look-up tables (LUTs) used for MD1 without weight reuse and $$N = 1$$ is 19,211, while this number for $$N = 2$$ is 28,534 ($$1.48 \times$$) and $$N = 4$$ is 54,595 ($$2.8 \times$$). The FFs usage also gives an impressive number when compared with $$N = 1$$, being 12,910 ($$1.42 \times$$) for $$N = 2$$ and 23,080 ($$2.53 \times$$) for $$N = 4$$. We also achieved similar results on MD2, both with and without weight reuse. For power assessment, Figure 22 also describes the efficiency enhancement when increasing $$N$$ value for the MD1 model with weight reuse enabled. According to the chart, the FPS/W ratio is doubled when increasing $$N$$ from 1 to 4. Besides, the chart shows that the power efficiency with a higher level of parallelism is always much better without changing frequency. Therefore, the proposed architecture can gain the highest efficiency by maximizing the frequency and parallel level.

D. EXPERIMENTAL EVALUATION

To estimate the efficiency of the proposed architecture, we conducted a list of experiments corresponding to different parameter sets and objectives. In particular, we investigated five factors: clock speed, loop-2 unrolling level, MAC optimization method, MAC compression method, and classification accuracy.

Firstly, we synthesized the MD1 model with different frequency values: 100, 150, 200, 250 and 300 Mhz. The result shows that the operating frequency does not affect hardware resources entirely. In contrast to this, the frame rate (or frame per second) increases the power consumption of the hardware implementation. Specifically, according to Figure 22, it is worth noting that the FPS/W ratio steadily increases along with clock frequency for all loop-2 unrolling levels. This indicates that for the proposed architecture, the speed of image classification increases faster than the rise of power consumption, which leads to a better power efficiency at higher frequencies.

Secondly, to study the impact of the loop-2 unrolling factor, we also synthesized the same MD1 with different values of $$N$$: 1, 2, 4. As discussed in the previous section, if the number of produced windows on each convolution line buffer increases $$N$$ times, total system throughput also increases $$N$$ times. For $$N = 1$$, the accelerator’s frame rate is $$3.83 \times 10^5$$, so with

![FIGURE 21. System architecture for the testing binary neural network IP.](image1)

![FIGURE 22. Power efficiency when under unrolling and frequency influence.](image2)
TABLE 3. Influence of unrolling, accuracy and reuse weight method on hardware area. Frequency is at 300 Mhz.

| Model | Accuracy | HW indices | Number of windows = 1 * | Number of windows = 2 * | Number of windows = 4 * |
|-------|----------|------------|-------------------------|-------------------------|-------------------------|
|       |          |            | (FPS = 3.83x10^5 b)     | (FPS = 7.65x10^5 b)     | (FPS = 1.53x10^6 b)     |
|       |          |            | Noreuse weight          | Reuse weight            | Noreuse weight          |
|       |          |            | LUTs                    | FFs                     | Power (W)               |
|       |          |            | 19,211                  | 9,104                   | 1.126                   |
|       |          |            | 10503 (54.67%)          | 6,023 (66.16%)          | 0.676 (60%)             |
|       |          |            | 28,534                  | 12,910                  | 1.671                   |
|       |          |            | 15,460 (53.63%)         | 8,795 (68.12%)          | 0.973 (58.22%)          |
| MD1   | 98.40%   |            | 54,595                  | 23,080                  | 3,332                   |
|       |          |            | 29,156 (53.4%)          | 15,341 (66.4%)          | 1,735 (52.07%)          |
|       |          |            |                         |                         |                         |
|       |          |            | LUTs                    | FFs                     | Power (W)               |
|       |          |            | 10,891                  | 6,394                   | 0.705                   |
|       |          |            | 6,144 (56.4%)           | 4516 (70.6%)            | 0.47 (66.67%)           |
|       |          |            | 15,470                  | 8,795                   | 0.965                   |
|       |          |            | 8,370 (54.1%)           | 5,961 (67.7%)           | 0.607 (62.9%)           |
| MD2   | 97.70%   |            | 29,115                  | 15,404                  | 1,725                   |
|       |          |            | 15,529 (53.34%)         | 10,058 (65.3%)          | 0.938 (54.4%)           |

* The number of windows is the number of windows generated by the CLB of the first convolution layer.

b If the number of windows increases N times, the frame per second (FPS) also increases N times.

consumption improvement tend to be higher on the models having more channels and (ii) the same tendency appears when increasing the parallelism level.

In terms of correlation between MD1 and MD2, the amount of LUTs used in MD2 for different values of N is 1.7-1.9× fewer than what needed in MD1, while FFs usage and power consumption are also reduced 1.3-1.5× and 1.3-1.5×, respectively. The classification accuracy only suffers a slight reduction of 0.7%. From observation, it is necessary to define a practical accuracy level that produces properly efficient hardware implementation to find an optimal solution.

After all investigations, we can conclude that the proposed architecture can be utilized most effectively by running at the highest frequency and parallel level. In addition, maximizing the number of channels on each layer is a compatible trade-off to keep accuracy instead of changing other factors.

E. COMPARISON TO PRIOR WORK

In this subsection, we continue comparing the most desirable result of the proposed architecture and the previous works using both MNIST and Cifar-10 dataset. For the MNIST dataset, in Table 4, two models MD1 and MD2 implemented with the MAC optimization method, popcount compression, or without popcount compression are selected to compare with previous works, in which there are four MD1 with 2 and 4 windows, and two MD2 with 4 windows. Based on the results, at 300Mhz, the popcount compression method can make the design smaller with lower number LUTs and FFs. However, the power consumption is more than using popcount without compression. The energy difference between using popcount compression and not using popcount compression is reduced with the lower frequency or bigger model, as shown in Table 1.

We choose five architectures that provide competitive performance using the MNIST dataset and binary weight regarding previous works. Firstly, authors in FINN [6] implemented a family of BNN models on hardware. They implemented an MLP model, which comprises three fully connected layers and takes a 28 × 28 binary image as the input. Among the previous work, FINN has the highest speed of image classification on the MNIST dataset (1,561k fps). The second reference, named FINN-R [16], is also an MLP model proposed by Xilinx’s research team. This model gives lower accuracy but consumes much fewer hardware resources. BNN-PYNQ [32] is the latest release of the FINN from Xilinx’s open-source project published on Github. We downloaded the project and synthesized it to reproduce their hardware utilization for comparison. This model gives the same accuracy as FINN, but the architecture consists of four fully connected layers. In addition, compared to FINN, this model uses considerably lower resources but provides much lower performance.
TABLE 4. Performance comparison between previous work and proposed architecture, in which MNIST is the dataset.

| Model     | Accuracy | Platform    | Freq. (MHz) | LUTs  | FFs  | BRAM | DSP | Framerate ($\times 10^3$ FPS) | Power (W) | GOPs | Power eff. (FPS/W) |
|-----------|----------|-------------|-------------|-------|------|------|-----|-------------------------------|------------|------|-------------------|
| MD1      | 98.4%    | Ultra96     | 300         | 15,460| 8,795| 0    | 0   | 765                          | 1.004      | 18,330| 761,952           |
| MD1      | 98.4%    | Ultra96     | 300         | 14,096| 8,923| 0    | 0   | 765                          | 0.973      | 18,330| 786,228           |
| MD1      | 98.4%    | Ultra96     | 300         | 29,156| 15,404| 0     | 0   | 1,530.6                      | 1.735      | 18,330| 882,190           |
| MD1      | 98.4%    | Ultra96     | 300         | 26,780| 15,238| 0    | 0   | 1,530.6                      | 1.795      | 18,330| 852,702           |
| MD2      | 97.7%    | Ultra96     | 300         | 15,529| 10,058| 0     | 0   | 1,530.6                      | 0.938      | 7,647 | 1,631,769         |
| MD2      | 97.7%    | Ultra96     | 300         | 14,361| 10,513| 0     | 0   | 1,530.6                      | 0.977      | 7,647 | 1,566,623         |
| FINN [6]  | 98.4%    | ZC706       | 200         | 82,988| -    | -    | -   | 14,256                      | -          | -     | -                 |
| FINN-R [16] | 97.69    | Ultra96     | 300         | 38,205| -    | 7,560| -   | 847.5                        | -          | 5,110 | -                 |
| BNN-PYMQ [32] | 98.4    | Ultra96     | 300         | 26,809| 30,947| 3,960| 4   | 356.6                        | 1.334      | 2,150 | 267,342           |
| FP-BNN [12] | 98.24%   | Stratix V   | 150         | -    | -    | 44,200| 20  | -                            | -          | 5,904 | -                 |
| Re-BNet [33] | 98.29%   | Spartan XC7S50 | 200     | 25,600| 34,230| 87    | 150 | 330                          | -          | -     | -                 |

a The number of windows is two and without popcount compression.
b The number of windows is two and with popcount compression.
c The number of windows is four and without popcount compression.
d The number of windows is four and with popcount compression.

(FPS = 356.6k fps). The model in the FP-BNN [12] paper also uses four fully connected layers to classify the MNIST dataset. The difference in this paper is that they used Stratix V from Altera Intel, and they suggested a compressor tree to optimize the Pop-count operation. The final reference is Re-BNet [33], which is an improved version of FINN combined with an algorithm optimization. This model shows its efficiency when keeping accuracy at 98.29%, while requiring hardware resources like 25,600 LUTs, being much smaller than the original FINN.

Table 4 shows the whole configurations and hardware implementation results of all references and two of our models. The hardware implementations based on the proposed architecture provide minimal areas compared to all other prior works regarding hardware utilization. For BNN-PYNQ, including the most lightweight architecture among the previous work [32] from Xilinx, our MD1 with 2 windows consumes 1.84× fewer LUTs, 3.77× fewer FFs, and provides 2.14 higher frame rate and 1.36 lower power consumption. Even for 4 windows generation, our model MD1 used fewer resources than the BNN-PYNQ with a little more LUTs count and 2.1× fewer FFs, but still produced 4.3× higher frame rate while maintaining the same accuracy.

When compared with the original FINN [6], the MD1 with 4 windows used 3× fewer LUTs and produced 98% frame rate. On the other hand, the smaller model MD2 provides a modest accuracy with the FINN-R [16], but uses 2.4× fewer LUTs and provides 1.8× higher frame rate when operating at the same clock speed. Unlike all other architecture, both MD1 and MD2 completely eliminated the use of on-chip memory devices and DSP slices, hence improving substantial power consumption. As discussed in the previous section, the power efficiency for our architecture becomes maximized when increasing both the clock speed and loop-unroll level.

At 300MHz and N = 4, our MD1 and MD2 with 4 windows generation provide 3.8× and 6.1× higher FPS/W, compared to BNN-PYNQ [32] respectively. Although not listed all in Table 4, both of our models can be configured with lower values of N if the frame rate is not as prioritized as hardware resources.

For the Cifar-10 dataset, we present 4 different architectures with different X values. With X = 1, the proposed architecture is implemented at 210 and 177 MHz. Based on the result, we can conclude that the area efficiency increase if the design is run at maximum frequency. At X = 2, the frequency used for MAC operation is 300 MHz, while the rest is run with 150MHz. Compared to the case X = 1, the number of LUTs decreases from 18% to 20%. At X = 4, MAC operation continues running at 300MHz, and 75MHz for others.
Hardware overhead reduces 32% and 46%, compared to $X = 2$ and $X = 1$, respectively. To evaluate the area efficiency, according to Table 5, the ratio FPS/LUTs is used to compare between the proposed design and remarkable previous works. The proposed design gives all X values better area efficiency than all previous designs. In particular, the proposed design’s area efficiency with $X = 1$ (0.707) is 1.5 times higher than the best previous design [16] (0.467). Regarding performance, the proposed design can give a super-fast framerate with 205,000 frames per second.

In summary, based on the result in Table 4 and Table 5, the proposed design for both MNIST and the Cifar-10 dataset gives a significantly higher power and area efficiency than the previous works. The main reason is from successfully applying some new optimization methods based on streaming and binary architecture features. In particular, firstly, different from the previous works, which are compared in Table 4 and Table 5, all XNOR logic gates into the design are removed or replaced by NOT gates (smaller than XNOR). As a consequence, memory storing weight kernel values is eliminated. Therefore, internal memory of the proposed design is zero, as shown in Table 4, while [6], [12], [16], [32], [33] need a certain amount of memory (BRAM). Secondly, the MAC optimization method is directly implemented without additional resources, compared to [18]. In addition, the line buffer in the proposed design just stores necessary data to provide for the next layer instead of storing all output feature maps; hence resources used for the proposed design are much lower than previous works. Moreover, the pipeline-unrolling method maximizes utilization of the maxpooling layer with the line buffer supporting various parallel levels, leading to the best power and resource efficiency. The throughput can increase N times, while the required hardware overhead is much lower than N times. Finally, the MAC compression technique mentioned in section III-C-3 is the last feature that helps the proposed design reduce a considerable amount of hardware resources without performance impact.

For more enormous datasets, according to the result for MNIST and Cifar-10 dataset, we can see that the area, power, and performance efficiency can be changed, depending on the compression level of the MAC operations. In particular, when increasing the N values (the number of generated windows from line-buffers), all area, power consumption, and throughput increase, where throughput rise faster than the rest. Hence, energy and area efficiency is improved. Meanwhile, when increasing X values (the number of output channels using one popcount operation), all area, frequency, energy, and throughput decrease, in which the throughput decreases faster than the rest, causing the efficiency of all aspects to be lower. In conclusion, the area, energy, and performance efficiency of a specific design using the proposed architecture depend on the X and N values, while X and N values depend on the available resources.

Recently, with the complex dataset-ImageNet, BNN has been found it challenging to have a compatible accuracy for practical applications (41% in [33]). In addition, currently, because of the lack of previous works implementing BNN for the Imagenet dataset, to more entirely evaluate the proposed architecture on larger networks and complex datasets, the corresponding proposed architectures are going to be implemented in the future work. Because the proposed architecture focuses on compressing the model width to avoid obstacles caused by the large number of layers, FPGA devices with much more hardware resources and ASIC are going to be used to implement larger models with complex datasets. Moreover, to maintain acceptable accuracy for practical applications, increasing the width should be considered rather than increasing the depth when training larger BNN models.

V. CONCLUSION

The BNN with small-sized parameters and low-cost calculation is a perfect fit to be implemented for the internet of things (IoT) or edge applications. Various optimization techniques have been incorporated into the proposed design, both from the hardware and algorithm point of view. The streaming architecture and unrolling mechanism enable high processing throughput, while the BRAM-less architecture and weight reuse method significantly reduce both hardware resources and power consumption in the final routed implementation. The effect of different optimization approaches has been summarized to provide a reference, which is the optimal set of design parameters for a particular design goal. In addition, we presented an automated design generation flow to implement arbitrary BNN models onto FPGA based on users’ specified BNN structures to maximize throughput and minimize power consumption in practical development time. As of date, the proposed BNN implementation delivers the highest performance in terms of throughput and power efficiency without sacrificing inference accuracy. In future work, to increase application range, the proposed architecture will continue to be improved with more complex neural network models, including shortcut connections. With small area and low latency, the proposed design is one of the best candidates for IoT or edge applications, where low power consumption and real-time response are of the essence.

REFERENCES

[1] A. Voulodimos, N. Doulamis, A. Doulamis, and E. Protopapadakis, “Deep learning for computer vision: A brief review,” Comput. Intell. Neurosci., vol. 2018, pp. 1–13, Jan. 2018, doi: 10.1155/2018/7068349.
[2] A. B. Nassif, I. Shahtin, I. Attili, M. Azzez, and K. Shaalan, “Speech recognition using deep neural networks: A systematic review,” IEEE Access, vol. 7, pp. 19143–19165, 2019.
[3] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” 2014, arXiv:1409.1556.
[4] S. Han, H. Mao, and W. J. Dally, “Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding,” 2015, arXiv:1510.00149.
[5] X. Liu, J. Pool, S. Han, and W. J. Dally, “Efficient sparse-Winograd convolutional neural networks,” 2018, arXiv:1802.06567.
[6] Y. Unuroglu, 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. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays, Feb. 2017, pp. 65–74.
[7] 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,” 2016, arXiv:1602.02830.
LOK-WON KIM received the Ph.D. degree from the University of California at Los Angeles, Los Angeles, CA, USA. He has over a decade of full-time research and industry experience with world-leading companies and research organizations, such as Apple Inc.; Cupertino, CA, USA; Cisco Systems, San Jose, CA, USA; the IBM T. J. Watson Research Center, Yorktown Heights, NY, USA; Broadcom, Irvine, CA, USA; the Korea Electronics Technology Institute, Seongnam, South Korea; and Hynix Semiconductor, Icheon, South Korea, where he has played a key role in the design and verification of various computer hardware systems widely used in the world. He has been with Apple Inc., where he has been leading the design and verification of application processors used in the iPhone, iPad, and Apple Watch. He is currently an Assistant Professor with the Department of Computer Science and Engineering, Kyung Hee University, Seoul, South Korea, where he is conducting research on neuromorphic computing for deep learning, security, and reliable computer hardware systems for smart and autonomous applications, automated optimization methodology for the power, area, and performance of digital designs, and reconfigurable computing. He has authored or coauthored nine journal articles, seven conference papers, and five patents. He was a recipient of the Cisco Achievement Program Award in recognition of outstanding employee effort and achievement at Cisco Systems.

CHOONG SEON HONG (Senior Member, IEEE) received the B.S. and M.S. degrees in electronic engineering from Kyung Hee University, Seoul, South Korea, in 1983 and 1985, respectively, and the Ph.D. degree from Keio University, Tokyo, Japan, in 1997. In 1988, he joined KT, Gyeonggi-do, South Korea, where he was involved in broadband networks as a Member of the Technical Staff. Since 1993, he has been with Keio University. He was with the Telecommunications Network Laboratory, KT, as a Senior Member of Technical Staff and the Director of the Networking Research Team until 1999. Since 1999, he has been a Professor with the Department of Computer Science and Engineering, Kyung Hee University. His research interests include future internet, intelligent edge computing, network management, and network security. He is a member of the Association for Computing Machinery (ACM), the Institute of Electronics, Information and Communication Engineers (IEICE), the Information Processing Society of Japan (IPSJ), the Korean Institute of Information Scientists and Engineers (KIISE), the Korean Institute of Communications and Information Sciences (KICS), the Korean Information Processing Society (KIPS), and the Open Standards and ICT Association (OSIA). He has served as the General Chair, the TPC Chair/Member, or an Organizing Committee Member of international conferences, such as the Network Operations and Management Symposium (NOMS), the International Symposium on Integrated Network Management (IM), the Asia-Pacific Network Operations and Management Symposium (APNOMS), the End-to-End Monitoring Techniques and Services (E2EMON), the IEEE Consumer Communications and Networking Conference (CCNC), the Korean Institute of Communications and Information Processing Society of Japan (KIPS), the Open Standards and ICT Association (OSIA). He has served as the General Chair, the TPC Chair/Member, or an Organizing Committee Member of international conferences, such as the Network Operations and Management Symposium (NOMS), the International Symposium on Integrated Network Management (IM), the Asia-Pacific Network Operations and Management Symposium (APNOMS), the End-to-End Monitoring Techniques and Services (E2EMON), the IEEE Consumer Communications and Networking Conference (CCNC), the Korean Institute of Communications and Information Sciences (KICS), the Korean Information Processing Society (KIPS), and the Open Standards and ICT Association (OSIA). He was an Associate Editor of the IEEE Transactions on Network and Service Management, the IEEE Journal of Communications and Networks, and the International Journal of Network Management; and an Associate Technical Editor of the IEEE Communications Magazine. He currently serves as an Associate Editor for the International Journal of Network Management and Future Internet Journal.