Abstract—Fueled by the success of mobile devices, the computational demands on these platforms have been rising faster than the computational and storage capacities or energy availability to perform tasks ranging from recognizing speech, images to automated reasoning and cognition. While the success of convolutional neural networks (CNNs) have contributed to such a vision, these algorithms stay out of the reach of limited computing and storage capabilities of mobile platforms. It is clear to most researchers that such a transition can only be achieved by using dedicated hardware accelerators on these platforms. However, CNNs with arithmetic-intensive operations remain particularly unsuitable for such acceleration both computationally as well as for the high memory bandwidth needs of highly parallel processing required. In this paper, we implement and optimize an alternative genre of networks, local binary pattern network (LBPNet) which eliminating arithmetic operations by combinatorial operations thus substantially boosting the efficiency of hardware implementation. LBPNet is built upon a radically different view of the arithmetic operations sought by conventional neural networks to overcome limitations posed by compression and quantization methods used for hardware implementation of CNNs. This paper explores in depth the design and implementation of both an architecture and critical optimizations of LBPNet for realization in accelerator hardware and provides a comparison of results with the state-of-art CNN on multiple datasets.

Keywords— Deep learning hardware accelerator, computer vision, FPGA, high-level-synthesis

I. INTRODUCTION

Convolutional Neural Networks (CNNs) [1]–[3] have outperformed other supervised learning methods in computer vision and have been used in many domains including web searching [4], speech/pattern recognition [5], biomedical analysis [6], etc. For most applications, CNN training through convex optimization requires intensive gradient computations. As a result, often the training phase is offline and done separately from the target platforms where recognition tasks may be needed. This is particularly true of devices at or near the edge of the network, the so-called “edge devices.” Even with this split, the convolution operation in the inference phase still overburdens the resource-limited embedded hardware [7] for Internet of Things (IoT) or real-time edge computing applications. To be specific, the edge devices challenges consist of congested inter-neurons connections, intensive memory accesses, large memory footprint to store parameters and feature maps, and high-latency high-precision multiplication and accumulation (MAC) operations. There are two main approaches to alleviate the burdens of CNNs for hardware implementations. One approach is to prune the less salient weights to skip the arithmetic operations with less significant numbers [8]–[10]. The other is to quantize floating numbers either statically [11] or dynamically [12] to degrade the precision for low-bit arithmetic logic units (ALUs). Binarization [7], [13], [14] pushed the static quantization to the limit, and thereby the original floating point multiplication was replaced with a 1-bit exclusive-nor gate (XNOR). There existed more intricate hybrid works of the two trends [15], as well as other explorations of efficient network structures [16], [17] that mainly aimed to reduce the model size from the network structure level.

While carrying out the trained models of the two aforementioned approaches, hardware platforms driven by CPU and GPU clusters inevitably encountered challenges such as the overhead of irregular memory accesses arisen from the pruned irregular matrices, and the limitation of current computer architecture to support sub-word variable storage units and arithmetic operations. On the other hand, field programmable gate arrays (FPGAs) provide an attractive alternative because they allow a highly customized design to handle the limitations on CPU/GPU machines [12]. Many hardware accelerators for pruned CNNs or binarized CNNs have been proposed [18], [19]. However, the equivalent compression rate of memory footprint and computation latency are still incremental and continue to be a challenge for effective use of machine learning in edge devices.

LBP Nets [20] fundamentally transforms the arithmetic multiply-and-add operations into sampling operations based on logic operations. We note that despite the similarity in names, local binary pattern (LBP) and LBP Nets are two very different techniques. LBP refers to a known method in the computer vision [21] as a type of visual descriptor used in image classification based on texture maps. Such a descriptor could be used by various classifier algorithms including support vector machines or other machine learning algorithms. LBPNet is a new way to implement neural network algorithms, which obviates the need for computing dot products and sliding windows for convolution operations. Instead, LBP Nets sample and compare the input image and records the comparison results to a predefined bit location. In other words, there is no MAC operation in an LBP layer, and only the trained patterns of sampling locations need to be stored. Therefore, the convolution-free LBP Nets are hardware-friendly that can
achieve significant benefits over the other CNN models.

In this paper, we implement and optimize an LBPNet for multiple datasets on FPGA targets to characterize its efficiency. Based on these experiments, we propose an efficient architecture for LBPNet and the critical optimization strategies. We implement and evaluate the complete system in terms of classification accuracy, latency, resource utilization, and energy efficiency.

II. PRELIMINARY

Since the LBPNet [20] was proposed to be an alternative of the prevailing deep learning method CNN, we start from the preliminary knowledge of CNNs.

A. Convolutional Neural Networks

Fig. 1 illustrates the operation in a convolutional layer. A convolutional layer (Conv layer) performs a 2-D spatial convolution on the input images or feature maps with kernels composed of multi-channel dense filters. Stacking multiple Conv layers up means taking the output feature maps of previous Conv layer as the input of the current concerned Conv layer. Deepening network structure can extract more abstract representations embedded in the images for classification.

B. Local Binary Pattern Network

LBPNet [20] operates based on the optical flow theory, more specifically in addressing the aperture problem. The image gradient of the input image/feature map guides the training of patterns. The local binary patterns are trained to minimize the cross-entropy of softmax cost function. The learning process deforms the sampling points in a pattern to a better set of locations for discriminative features. The channel fusion process is done with random projection, which has been proven to be an effective distance preserving method [22].

Fig. 2 illustrates the operations in an LBP layer; three local binary patterns are visualized as black masks with certain apertures on them. Analogous to a Conv layer, there are multiple local binary patterns, which record the sampling positions for the comparison with a pivot sampling. For each comparison pair, the pivot (a star-shaped aperture) and a sampling point (a round aperture) are used to index two values from the input features. The results of comparisons are allocated to predefined locations in a bit array. If multiple input channels presents, results of LBP operations on all channels are fused together according to a random projection map. There is no MAC operation or convolution in an LBP Layer. The low bit comparison can be implemented in combinational logic, and bit allocations require only a good buffer design.

The benefit of LBPNet is multi-fold. One, the sparse sampling pattern greatly reduces the model size. An LBP pattern contains $N_{sampling}$ sampling points’ locations on a window. Assuming the number of input channel is $N_{in}$, and the number of output channel is $N_{out}$, the number of sampling locations is $2 \times N_{out} N_{in} N_{sampling}$, where the number 2 means the two dimension locations. However, the presence of random projection instructs us to compare only a part of the sampling pairs and drop the unused pairs. Therefore, we only need store $2 \times N_{out} N_{sampling}$ sampling positions and a mapping table of size $N_{out} \times N_{Sampling}$. All of the them are typically in Kbits.

Second, the convolution-free design of LBP layers unleashes the computation latency from the system pipelined cycles. We can design a customized comparator module for the data parallelism in an LBP Layer. The speedup of an LBP layer over a Conv layer with massive MAC operation is, therefore, guaranteed.

Third, LBPNets reduce the hardware cost. In FPGA, it takes 62 LUTs to implement an 8-bit multiplier, and 8 LUTs for 8-bit adder, while a boolean comparator requires only 4 LUTs, which implies we can either use cheaper FPGA with less computation capability or implement more data parallelism within the same FPGA compared with a CNN-based accelerator.

Last but not least, the energy efficiency is expected to be higher than CNN-based accelerator because we only need comparison and buffering to implement LBPNets. For many application, such as unmanned aerial vehicles or hearing aided devices, the short battery life is usually a critical issue. The hardware accelerated LBPNets can be used on the energy
efficiency concerning applications to boost user experience.

III. ANALYSIS AND MODIFICATIONS OF LBPNETS

The multi-layer perceptron (MLP) classifier in LBPNet [20] was not the main focus, and hence the floating arithmetics were adopted. To fill the gap between theory and practice, we modify the MLP classifier with two advanced techniques and train the networks from scratch for FPGA implementation. In this section, we start with an overview of the network structure and then dive into the modifications for hardware.

A. Structures of LBPNets

We implemented multiple LBPNets for different datasets. Despite the numbers of kernels and depths, all network structures share the same characteristics. In this section, we use the network for the MNIST dataset as an example to analyze the structure before describing the FPGA architecture.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Input ch ( N_{in} )</th>
<th>output ch ( N_{out} )</th>
<th>Fmap dim of</th>
<th>Fmap size (Kbit)</th>
<th>Param (Kbit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LBP1</td>
<td>1 39 39 32 x 32</td>
<td>40 40 32 x 32</td>
<td>163.84</td>
<td>-</td>
<td>2.18</td>
</tr>
<tr>
<td>Joint1</td>
<td>40 40 32 x 32</td>
<td>80 80 32 x 32</td>
<td>327.68</td>
<td>-</td>
<td>4.48</td>
</tr>
<tr>
<td>LBP2</td>
<td>80 80 32 x 32</td>
<td>160 160 32 x 32</td>
<td>655.36</td>
<td>-</td>
<td>8.19</td>
</tr>
<tr>
<td>Joint2</td>
<td>160 160 32 x 32</td>
<td>320 320 32 x 32</td>
<td>1,230.72</td>
<td>-</td>
<td>16.32</td>
</tr>
<tr>
<td>AvgPool</td>
<td>160 160 5 x 5</td>
<td>512 512 1</td>
<td>32.00</td>
<td>-</td>
<td>0.51</td>
</tr>
<tr>
<td>BatchNorm</td>
<td>512 512 1</td>
<td>512 512 1</td>
<td>0.08</td>
<td>-</td>
<td>5.28</td>
</tr>
<tr>
<td>FC1</td>
<td>512 512 1</td>
<td>1024 1024 1</td>
<td>4.10</td>
<td>2,056.19</td>
<td>8.90</td>
</tr>
<tr>
<td>FC2</td>
<td>1024 1024 1</td>
<td>1024 1024 1</td>
<td>4.69</td>
<td>2,069.66</td>
<td>8.90</td>
</tr>
</tbody>
</table>

TABLE I

The architecture of our modified LBPNets on MNIST. The binarized MLP classifier is composed of two binarized FC layers and one modified Batch Normalization Layer [19]. Although binarized, the FC parameters can only be stored in an off-chip DRAM due to the limitation of FPGAs’ typical on-chip BRAM size [18].

For the MLP classifier part, an average pooling layer is then used to reduce 2-D images. The two binarized FC layers and one modified batch normalization layers are designed to reduce the dimension of data further and extract features for the 10 classes of MNIST dataset.

B. Binarization of Weights

We binarize the weights of both the two fully connected layers to either -1 or 1. Then, we set all -1 to 0 for digital circuitry. The input of the first layer is the averaged value from the AvgPool Layer, which is in floating or fixed numbers. Although we cannot use an XNOR gate to replace the multiplication between the input and a weight, binarized weights enable us to use a multiplexer to select whether to add or subtract the input from an accumulator. The second binarized fully connected layer takes binarized input from the BatchNorm layer. Therefore, we can replace the multiplication with an XNOR operation in the dot-product. However, binarization has proven to be lossy [23]. We must expect inferior classification accuracy and evaluate the difference for the trade-off between binary and floating arithmetic operations.

C. Simplification of Batch Normalization Layer

To avoid on-chip floating point arithmetic operations, we also introduce the method for the batch normalization layer mentioned in FINN [19]. This consists of methods to combine the binarization activation function with the linear transform and calculating a threshold for each input activation off-line as shown in Eq. 1 and Eq. 2.

\[
\gamma (x - \mu) / \sigma + \beta > 0,
\]

where \( x \) is the input, \( \mu \) is the mean over a mini-batch, \( \sigma \) is the standard deviation over a mini-batch, \( \gamma \) is the learned scaling factor, and \( \beta \) is the learned shifting factor.

\[
\begin{cases} 
  x > \text{threshold}, & \gamma \sigma > 0 \\
  x < \text{threshold}, & \text{otherwise}
\end{cases}
\]

\[
\mu - \beta \sigma
\]

where \( \text{threshold} = \mu - \beta \sigma \) is calculated off-line after the modification from Eq. 1 to Eq. 2 and is lossless due to the mathematical equivalence.

The modified LBPNets are trained on a GPU machine with NVIDIA Tesla K40, and the training achieves 100.0% accuracy. The test accuracy is 99.34% on MNIST. Compared with the LBPNet paper [20], as mentioned earlier, we have sacrificed some classification accuracy to make LBPNet hardware-friendly through binarizing the MLP classifier.

IV. FPGA ACCELERATOR DESIGN

A. Accelerator Architecture

An overview of the accelerator architecture is shown in Figure 4. The accelerator consists of four compute units as shown for each different type of layer, data and weight buffers, a memory access controller for off-chip memory transfers, and a controller. The operations of the LBP, Average Pool, Fully-Connected, and BatchNorm layers are performed through the four compute units LBP, AvgPool, FC, and BatchNorm, respectively. The LBP unit – dedicated to perform the compare
operations in the LBP layer – consists of a set of logical elements, while the other compute units are made up of arithmetic units to perform operations such as addition, multiplication, and division for the other layers. The input data and the parameters of the layers loaded from the off-chip memory are stored into the on-chip Data buffer and Parameters buffer, respectively. In addition to storing the input data, the Data buffer can accommodate the intermediate results of the layers which are necessary for the computations of their next layer. Because the size of the intermediate outputs of the layers is small, they can easily be stored on the available on-chip memories. This eliminates the need to transfer intermediate outputs between the accelerator and the off-chip memory. Thus, off-chip memory transfers are only needed for the input image, loading each layers weights, and sending back the final prediction output. This is one of the benefits of LBPNet compared to most other CNN-based accelerators where the size of intermediate results typically exceeds the available on-chip storage. On the other hand, the Parameter buffer can store all the weights for all the LBP layers at once. So only one time data and weight load is required for all the LBP layers and AvgPool layer to compute the input of the first FC layer. On the other hand, there is only enough space to store a portion of the FC layers weights. Each time a new set of weights are loaded into the parameter buffer and a new set of intermediate result is generated. This continues until all FC layer outputs are generated and stored in the on-chip Data buffer. To accelerate the communication and parallelize computations, we pack our 8-bit weights and generate to 64-bit words, store these words in the buffers, and unpack them to perform parallel computations.

B. Execution Flow of the Accelerator

At the beginning, input image and parameters of the three LBP layers are loaded from the off-chip memory to the Data buffer and Parameters buffer. Afterwards, the LBP compute unit performs the corresponding comparison operations starting from the layer LBP1. Accordingly, the output of LBP1 is stored in Data buffer on top of the input data. The process continues until all the LBP layers are performed. Then, the AvgPool unit starts performing a quantized version of average pooling operations on the data to reduce its dimensions. At this point, the parameters stored in the Parameters buffer are not needed any longer, and the space can be freed to store the parameters of other layers.

The next pass operates on layer FC1. Since the parameters of this layer exceed the size of the Parameters buffer, a portion of the parameters are loaded into the on-chip buffer, and the corresponding multiply-and-accumulate (MAC) operations are performed, and the partial outputs are stored in the Data buffer. Then, the process moves to the computations with the next part of parameters by loading them into Parameters buffer and performing the corresponding FC computations. After all the computations of the layer FC1 are completed, the parameters of the BatchNorm layer are brought into the Parameters buffer and overwrite the parameters of FC1. Then, the compute unit BatchNorm performs the batch normalization on the results of FC1 which are available in the Data buffer. The new outputs generated by the BatchNorm unit are stored in the Data buffer. Finally, the computations of the last layer are performed similarly to executing the layer FC1. The last FC unit generates prediction output values. The final label is computed using ArgMax operation on the results of the last FC layer and is written back to off-chip memory.

C. Compute Units Architecture

**LBP Layers**: The LBP unit is the most critical component of the accelerator responsible for a number of repeated LBP layers. Each unit in the LBP layer is responsible for reading eight input pixels from the data buffer and performing four comparison operations to generate one output. The position of these eight points are read from the weight buffer; then we can access the corresponding locations in the data buffer, compare every two of them together, and generate the corresponding output pixel by concatenating the four comparison results. As the weights are 8-bits for these layers, and they are stored in 64-bit words, we only need to read two words from the weight buffer which can be done in one cycle. These values indicate the position of points that should be accessed from a tensor in the data buffer. After reading each two-pixel values, a comparison is performed, and 1 bit of the output pixel is generated. This process is performed for every input channel in a pipeline fashion. This process is repeated in a sliding window pattern for the whole image. To improve the latency of the LBP computations, the operations inside the LBP can be parallelized. In this case, we partition the tensor input horizontally, and each processing element performs the aforementioned operations on one part. In order for the processing elements to access to data buffer at the same time, we partition the data buffer BRAM horizontally. There is clearly a trade-off between resource utilization and performance as we change the level of parallelism.

**FC Layer**: Each cycle we read in \( N \) data words and an equal number of weight words. \( N \) here is the input parallelization factor (we used 8 in our implementation). We apply appropriate memory partitioning to be able to access to \( N \) data words in one cycle. \( N \) multiplications are done in parallel, and this process is pipelined until an output is generated. As
we perform quantization on FC layers, we only have integer MAC units. After the computations on the available set of weights are done, a new set of weights are loaded from the off-chip memory, and the next set of outputs are generated. Note that the level parallelism in FC layer is typically bound by memory bandwidth of the off-chip connection, rather than the throughput of the accelerator.

**BatchNorm Layer:** We implement batch normalization layer using a parallel comparison between the data and weights, and multiplexers to generate a binarized output for the next fully-connected layer. In each cycle, eight parallel comparisons are made to generate eight outputs, and this process is performed in a fully pipelined fashion.

**AvgPool Layer:** This layer is relatively simple. It averages over 5-by-5 windows from input channels. The required memory read, computation, and memory write is fully pipelined.

**V. EXPERIMENTAL RESULTS**

**A. Dataset**

<table>
<thead>
<tr>
<th>Dataset</th>
<th>#Class #Examples Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MNIST</td>
<td>10 72.2577-26.032 Handwritten number</td>
</tr>
<tr>
<td>SVHN</td>
<td>10 73.462,000 Handwritten Devanagari characters</td>
</tr>
<tr>
<td>ICDAR-DIGITS</td>
<td>10 988 Photos of numbers</td>
</tr>
<tr>
<td>ICDAR-UpperCase</td>
<td>26 5,288 Photos of lower case Eng. char.</td>
</tr>
<tr>
<td>ICDAR-LowerCase</td>
<td>26 5,453 Photos of upper case Eng. char.</td>
</tr>
<tr>
<td>Char54K-EnglishHnd</td>
<td>62 7,705 Photos, Alphanumeric</td>
</tr>
<tr>
<td>Char54K-EnglishImg</td>
<td>62 3,410 Handwritten, Alphanumeric</td>
</tr>
<tr>
<td>Char54K-EnglishFnt</td>
<td>62 62,992 Printed Fonts, Alphanumeric</td>
</tr>
<tr>
<td>Char54K-EnglishImg</td>
<td>62 26,032 Photos of house number</td>
</tr>
</tbody>
</table>

**TABLE II**

<table>
<thead>
<tr>
<th>Dataset</th>
<th>#Class #Examples Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MNIST</td>
<td>10 72.2577-26.032 Handwritten number</td>
</tr>
<tr>
<td>SVHN</td>
<td>10 73.462,000 Handwritten Devanagari characters</td>
</tr>
<tr>
<td>ICDAR-DIGITS</td>
<td>10 988 Photos of numbers</td>
</tr>
<tr>
<td>ICDAR-UpperCase</td>
<td>26 5,288 Photos of lower case Eng. char.</td>
</tr>
<tr>
<td>ICDAR-LowerCase</td>
<td>26 5,453 Photos of upper case Eng. char.</td>
</tr>
<tr>
<td>Char54K-EnglishHnd</td>
<td>62 7,705 Photos, Alphanumeric</td>
</tr>
<tr>
<td>Char54K-EnglishImg</td>
<td>62 3,410 Handwritten, Alphanumeric</td>
</tr>
<tr>
<td>Char54K-EnglishFnt</td>
<td>62 62,992 Printed Fonts, Alphanumeric</td>
</tr>
</tbody>
</table>

**B. Experiment Setup**

We implemented our design in C++ and used Xilinx Vivado HLS and Vivado Suite 2015.4 as the primary tool for synthesizing the accelerator. We evaluate resulting designs on a low-cost Xilinx Zynq-7000 series (XC7Z020 FPGA) target. We performed HLS design space exploration to select the design options that strike a balance between resource utilization and latency. In our final design, we use 64-bit words, and the LBP compute unit consists of four parallel processing units. The resource utilization and power numbers are reported by Vivado tool after placement and route.

**C. Results**

<table>
<thead>
<tr>
<th>Dataset</th>
<th>#Class #Examples Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MNIST</td>
<td>10 72.2577-26.032 Handwritten number</td>
</tr>
<tr>
<td>SVHN</td>
<td>10 73.462,000 Handwritten Devanagari characters</td>
</tr>
<tr>
<td>ICDAR-DIGITS</td>
<td>10 988 Photos of numbers</td>
</tr>
<tr>
<td>ICDAR-UpperCase</td>
<td>26 5,288 Photos of lower case Eng. char.</td>
</tr>
<tr>
<td>ICDAR-LowerCase</td>
<td>26 5,453 Photos of upper case Eng. char.</td>
</tr>
<tr>
<td>Char54K-EnglishHnd</td>
<td>62 7,705 Photos, Alphanumeric</td>
</tr>
<tr>
<td>Char54K-EnglishImg</td>
<td>62 3,410 Handwritten, Alphanumeric</td>
</tr>
<tr>
<td>Char54K-EnglishFnt</td>
<td>62 62,992 Printed Fonts, Alphanumeric</td>
</tr>
</tbody>
</table>

**TABLE III**

**LBPNet structure and Accuracy. We use the same binarized MLP classifier throughout the experiment. The CNN baseline results are listed as well.**

Table III lists the LBPNet structure and the accuracy for every dataset as well as the CNN baseline. For those baseline results without references, we build CNNs with the same network structures like LBPNet.

The resource utilization for our design is 7954 LUT, 7188 FF, 68 BRAM, and 16 DSP. Our FPGA implementation works at 200 MHz. We evaluate performance of our accelerator for different datasets. The latency break-down for different layers and total execution time is summarized in Table IV. The last column in this table (labeled as Total Runtime), shows the execution time (in millisecond) per image for the optimized design for different datasets. This table also shows the breakdown of latencies (number of cycles) per layer in columns 2-5. Column 2 and 4 shows the sum of latencies for all LBP layers and FC layers. Since different LBP and FC layers work on different data sizes, their latency is different. For example, for the MNIST dataset, the latency of three LBP layers are 51745, 78404, 156804 cycles respectively. For the same dataset, the latencies in the two FC layers are 259588 and 811 cycles respectively. Similarly, there are difference in the runtime for different datasets because they use different numbers of LBP or different numbers of kernels.

**TABLE IV**

**Latency (number of clock cycles) break-down for different layers and total run time for different datasets. The runtime is in millisecond.**

We compare our design with off-the-shelf CNN FPGA implementations. Table V compares the resource utilization for different FPGA implementations of LeNet with LBPNet. As show, our accelerator achieves highest accuracy among all implementations. It utilizes only 48.6% of BRAMs, 7.3% of DSP units, 15.2% of LUTs, and 6.8% of Flip flops on our target FPGA. Comparing to CNN architectures, we mostly have better resource utilization. We also compare our throughput to other works. Throughput is shown in gigaroperations-per-second (GOPS). Our accelerator achieves better throughput that CNN-based accelerators. We also have better power consumption when compared to CNN implementations. For example, [28] utilizes 3.32 W power, while our accelerator consumes only 0.5 W to perform classification. Our accelerator is more energy efficient than CNN due to replacing expensive convolution operations with simple logical operations. In general, LBPNet enables us to achieve a good balance between resource utilization and throughput, while maximizing accuracy. LBPNet’s inference operations on a CUDA-supported GPU presents a latency of 0.7 ms per image for MNIST dataset, the average power consumption of 130, and the memory consumption of 44 MBytes. By comparison, FPGA implementation of LBPNet here is 4.4X slower than a Tesla K40 GPU, but 52X more energy efficient compared to the GPU implementation.
VI. RELATED WORK

Our work is the first to approach design of hardware-accelerated LBPNets. While it is hard to conduct a direct comparison with existing hardware accelerators for CNNs because of the diversity of implementation choices, the effect on computation and memory size can be examined. Here we provide three published works regarding the compression of CNNs as a reference as these use common essential techniques.

Deep Compression [15] utilized multiple techniques to achieve a compression rate of 35X: pruning, quantization, customized weight encoding, Huffman encoding. However, owing to that the pruning and quantization retraining loops were not combined to minimize the interactive effects, there was no guarantee to the global minimum of the training result. FINN [19] fully exploited the critical characteristics of BNN: 1) a popcount module was synthesized to count the number of 1’s. 2) redesigning BatchNorm to threshold the popcount results with different values, which can be calculated off-line. Although FINN provided a complete synthesizing flow for trained BNNs, it degraded the classification accuracies because the padding subroutine was not correctly imposed.

BCNN Accelerator [18]. The authors adopted a partially shared streaming architecture and managed to process subword buffering and storing efficient. Due to the retraining of the network, BCNN Accelerator solved the zero-padding issue as FINN encountered.

All the three works were based on CNN, and the smallest model size they achieved was still a couple of Mbits. The most efforts in those work were the quantization of CNNs because CNNs were not designed to be hardware-friendly. Instead, our accelerated LBPNets are designed for bit-wise operation since the development of its algorithm [20].

REFERENCES