A 7.663-TOPS 8.2-W Energy-efficient FPGA Accelerator for Binary Convolutional Neural Networks

Yixing Li¹
yixingli@asu.edu

Zichuan Liu²
zliu016@e.ntu.edu.sg

Kai Xu¹
kaixu@asu.edu

Hao Yu²
haoyu@ntu.edu.sg

Fengbo Ren¹
renfengbo@asu.edu

¹School of Computing, Informatics, Decision Systems Engineering, Arizona State University, Tempe, USA
²School of Electrical and Electronic Engineering, Nanyang Technological University, Singapore

ABSTRACT
FPGA-based hardware accelerators for convolutional neural networks (CNNs) have obtained great attentions due to their higher energy efficiency than GPUs. However, it is challenging for FPGA-based solutions to achieve a higher throughput than GPU counterparts. In this paper, we demonstrate that FPGA acceleration can be a superior solution in terms of both throughput and energy efficiency when a CNN is trained with binary constraints on weights and activations. Specifically, we propose an optimized accelerator architecture tailored for bitwise convolution and normalization that features massive spatial parallelism with deep pipelines stages. Experiment results show that the proposed architecture is 8.3x faster and 75x more energy-efficient than a Titan X GPU for processing online individual requests (in small batch size). For processing static data (in large batch size), the proposed solution is on a par with a Titan X GPU in terms of throughput while delivering 9.5x higher energy efficiency.

1. INTRODUCTION
Convolutional neural network (CNN) has become a popular machine learning engine for many image-related data analytics [13] [15-16], such as image classification, face detection, object tracking, etc. CNNs outperform traditional feature selection based approaches especially for learning from big data. For a conventional CNN, high computation complexity and large memory footprint are the two main throughput bottlenecks for hardware acceleration. Therefore, the unmet throughput need of CNN calls for the development of more efficient hardware acceleration solutions for driving real-time applications.

Several methods have been proposed to alleviate the computation complexity and memory footprint by reducing the redundancy of CNN models. These methods include pruning [18], reduced-precision CNNs [4], and binary CNNs (BCNNs) [9]. The pruning technique [18] prunes the “useless” weights of a trained network based on sensitivity analysis, which can effectively reduce the CNN weight count (usually referred to as network size) for a ten-class classification problem by 75% [18]. [4] demonstrates that reducing the numerical precision of a CNN from 32 to 16 bits has very limited impact on classification accuracy. This can result in a network size reduction of 50%. However, a numerical precision below 8 bits resulted from quantization in the post-training stage often suffers from unacceptable accuracy drop [4]. Alternatively, recent advancement in binary-constrained deep learning has opened up new opportunities for efficient hardware acceleration. BinaryConnect [5] and the work in [6] demonstrate the successful use of binary and ternary (-1, 0, +1) weights in CNN, respectively. But, they both have non-binary activations. As one step forward, EBP [7], Bitwise DNNs [8], and the BCNN in [9] successfully exploit both binary weights and activations. In particular, the BCNN in [9] shows a 0.96% classification error rate on the MNIST database [17], which is comparable to a full-precision state-of-the-art CNN. Overall, BCNNs have been shown to be able to reduce the network size by up to 96.8% with minimum accuracy loss. Therefore, it is believed that BCNN is a more hardware-friendly model with the best accuracy-complexity trade-off.

Thus far, GPU-based CNN accelerator is still dominant due to its improved throughput over CPUs. However, the high power consumption of GPUs has brought up cooling concerns in data center computing. On the other hand, FPGA-based CNN accelerator has been widely investigated due to its energy efficiency benefits. As the throughput is proportional to the parallelism and frequency, the theoretical throughput of GPU-based and FPGA-based CNN accelerators can be estimated on the 1st order based on device specification. A Titan X GPU has 3,072 CUDA cores, while a Virtex-7 FPGA has 3,600 DSP48 slices. For implementing a full-precision CNN, the parallelism of GPUs and FPGAs can be approximately the same, while GPUs offer 5-10x higher frequency. As a result, FPGAs can hardly match up the throughput of GPUs. Differently, for a BCNN, the MAC operations become bitwise XNORs and bit-count operations. A direct impact is that one can use LUTs instead of DSP48 slides to implement the bitwise operations on an FPGA. Hundred thousand of LUTs make it possible for FPGA to match up the throughput of GPUs, even considering the bitwise operation capability of CUDA cores. Moreover, FPGAs benefit from much higher energy efficiency, which makes it a superior solution for accelerating BCNN in data center settings.

Early research effort [9] shows that GPU can get 7x speedup using a binary kernel for MNIST classification task on a binary multilayer perceptron (MLP). However, there have been very few studies on exploring FPGA-based accelerator architecture for binary neural networks. In this paper, we propose an optimized FPGA accelerator architecture tailored for a 9-layer BCNN. The proposed architecture implemented on a Xilinx Virtex-7 XC7VX690 FPGA achieves nearly state-of-the-art classification accuracy on CIFAR-10. The experiment results show that our FPGA implementation outperforms its optimized GPU counterpart with 75x higher energy efficiency and 8.3x higher throughput for processing a small batch size of 16 images (e.g. online request processing). For processing a large batch size of 512 images (e.g.
static data processing), the FPGA implementation achieves comparable throughput with 9.5x higher energy efficiency compared with GPU counterpart.

The contributions of this paper are summarized as follows:

- We demonstrate a 7.663-TOPS 8.2-W FPGA accelerator for BCNN that highly outperforms the GPU counterpart especially for processing online individual requests (in small batch size).
- We reveal the impact of applying binary constraints in CNN training on FPGA implementations is the enablement of massive computing parallelism of bitwise operations based on abundant LUT resources.
- We optimize the accelerator architecture to fully exploit both spatial and temporal parallelism across all the layers using architectural unfolding, pipelining, and data-flow control with memory channels. Compared with GPU implementations that only have spatial parallelism, the proposed architecture offers superior throughput and energy efficiency performance regardless of the size of workload.

2. BACKGROUND & MOTIVATION

2.1 Binary CNN (BCNN)

A CNN is a trained neural network model with high-level features extracted from input image [13]. A typical CNN model contains convolutional, pooling, and fully-connected layers. The first few layers usually capture regional information such as edges and curves and the last few layers interpret these low-level features into high-level abstractions with the posterior probability assigned for classification. A BCNN is a CNN trained with binary constraints resulting in binary weights and activations, and a huge reduction in computation complexity.

2.1.1 Convolution

The convolution layer is the core layer of a BCNN. Same as typical CNN, the input of each convolutional layer is a 3D feature map with a size of $\text{WID}' \times \text{HEI}' \times \text{DEP}'$ as shown in Figure 1. Each filter has a size of $\text{FW} \times \text{FH} \times \text{FD}$, where $\text{FW}$ and $\text{FH}$ is the width and height of the receptor field, respectively, and $\text{FD}$ is equal to the depth $\text{DEP}'$ of the input feature maps. N filters are constructed as a 4D tensor. The output feature maps $\text{Y}$ in the size of $\text{WID} \times \text{HEI} \times \text{DEP}$ are obtained from the spatial convolution along the 1st and the 2nd dimensions of the input feature maps with 3D-filter $\text{W}[n]$. In a BCNN, both the weights and activations are constrained to a binary set of values, e.g. [+1, -1]. As such, the multiplications in convolution is simplified to a bitwise operation and summation becomes bit-count operation. The operation in BCNN convolutional layers is defined as

$$Y[n][w'][h'] = \sum_{w=0}^{\text{FW}-1} \sum_{h=0}^{\text{FH}-1} \sum_{d=0}^{\text{FD}-1} \text{W}[n][w][h][d] \oplus \text{fmap}[w'+w][h'+h][d].$$

(1)

Comparing to a real-valued CNN with a single-precision data format, both the logic and memory resources can be greatly reduced in the hardware implementation of a BCNN.

2.1.2 Pooling

The pooling layer performs subsampling across a K x K contiguous region on the output feature map of convolutional layers, which pools out sensitive information regarded to classification and eliminated insensitive one. There are two kinds of pooling methods commonly used in BCNNs. One is max-pooling, which takes the maximum value of the pooling region. The other is average-pooling, which takes the mean value of the pooling region.

2.1.3 Normalization

Normalization is a powerful technique that stabilizes and accelerates the training process [11]. In the inference stage, normalization is also needed to match the training process. Statistical reference values are counted across the whole training set as

$$z = \frac{Y - \mu}{\sqrt{\sigma^2 + \epsilon}} Y + \beta,$$

(2)

where $\mu$ is the mean value and $\sigma^2$ is the variance with very a small constant $\epsilon$ to ensure a non-zero denominator. Note that $\gamma$ and $\beta$ scales and shifts the normalized values. Since $\mu$, $\sigma^2$, $\epsilon$, $\gamma$ and $\beta$ are all constants in inference stage, it enables the possibility to reduce the computation complexity of normalization.

2.1.4 Nonlinear function (Binarization)

Nonlinear function is the element-wise operation that performs on each neuron after normalization in convolutional layers and fully-connected layers [13]. Since the weights and activations are constrained to either +1 or -1, the nonlinear function of BCNN becomes an adjusted sign function, a.k.a. a Binarize function defined as

$$\text{Binarize}(z) = \begin{cases} 1 & \text{if } z \geq 0, \\ -1 & \text{otherwise,} \\ \end{cases}$$

(3)

2.2 A BCNN on CIFAR-10

The overall architecture of BCNN is shown in Table 1 [9]. It takes an RGB image with a size of 3 x 32 x 32 as the input of the

Table 1. BCNN configurations

| Name | CONV-1 | CONV-2 | CONV-3 | CONV-4 | CONV-5 |
|------|--------|--------|--------|--------|--------|
| Filter/weight | 3x3x3 | 128x3x3 | 128x3x3 | 256x3x3 | 256x3x3 |
| # of filters | 128 | 128 | 256 | 256 | 512 |
| Output size | 128x32x32 | 128x16x16 | 256x16x16 | 256x8x8 | 512x8x8 |
| Name | FC-1 | FC-2 | FC-3 |
| Filter/weight | 512x3x3 | 8192x1024 | 1024x1024 |
| # of filters | 512 | - | - |
| Output size | 512x4x4 | 1024 | 1024 |

Figure 1. BCNN overview.
first layer. For each convolutional layer, filter size is fixed as 3 × 3 with a stride of 1 pixel and zero padding of 1 pixel. The filter information of each convolutional layer in Table 1 is denoted as the \( W \times H \times E \times D \). Max-pooling is performed over a 2 × 2 window with stride 2 followed by convolutional layer 2, 4 and 6. Last three layers are fully connected layers. Normalization is applied to all the layers, which is followed by binarization except the last layer.

Figure 2 shows the accuracy comparison between the BCNN and a reduced-precision CNN with the same configuration in Table 1. It is shown that simply quantizing the network parameters below 10 bits in the post-training stage will cause significant accuracy drop. Differently, the BCNN trained with binary constraints can achieve almost the same accuracy as the full-precision CNN. This indicates that BCNN offers much superior trade-off between complexity and accuracy and is ideal for efficient hardware implementation.

2.3 BCNN’s impact on accelerators

A Titan X GPU has 3,072 CUDA cores (one ALU per core) and can run at 1 GHz, while a midrange FPGA Virtex-7 has 3,600 DSP48 slices, 433,200 LUTs and typically runs at around 100-200 MHz. As a 1st order estimation for a full-precision or reduced-precision CNN, two devices are barely on a par with the level of computing parallelism considering that a CUDA core and a DSP48 slice can map a floating- and a fixed-point multiplication accumulator (MAC), respectively. But, FPGAs run at a 5-10x lower frequency in general. As a result, the existing FPGA implementations of reduced-precision CNNs can hardly get comparable throughput to their GPU counterparts.

Differently, BCNN offers large room for throughput improvement for both GPU-based and FPGA-based implementations. For a GPU-based BCNN with a tailored binary kernel, one CUDA core can process 32-bit bitwise operation per clock cycle in a fully-pipelined ALU, which increase the equivalent parallelism of a Titan X GPU to 3,072×32=98,304. On the other hand, for an FPGA-based BCNN, the bitwise operation can be efficiently mapped onto the abundant LUT resources. Since one 6-input LUT can map 2.5 MACs on average, the computing parallelism of a Virtex-7 FPGA is on the order of 433,200×2.5=1,083,000. Taken the operation frequency into consideration, the throughput of GPU- and FPGA-based BCNN implementations should reach similar levels. The FPGA-based solution also enjoys much higher energy-efficiency. It is worth mentioning that GPUs can only achieve theoretical throughput when the data batch size is large enough to hide the computation and memory access latency. Thus, in the application scenarios such as processing online classification requests from individual users where small batches of data must be processed on the fly, FPGA-based solution will keep the promise to outperform GPU counterparts on both throughput and energy efficiency. In the following sections, we present an FPGA-based BCNN accelerator and benchmarking studies that validate our hypothesis.

3. Algorithm Reformulation for FPGA Mapping

For the best quality of implementation results on an FPGA, we reformulate BNN model to our BCNN model to further improve the hardware-friendliness of the BNN model [9].

3.1 Binary-encoded Convolution

When training a BCNN in [9], the weights and activations are constrained to either +1 or -1. For efficient FPGA mapping, we encode +1 and -1 as 1 and 0, respectively, in our design. In this way, it only takes 1-bit word length to store a weight value or an activation value. Moreover, the convolution operation in layer \( l \) is simplified into an XNOR dot product of the input feature map \( a_{l-1}^p \) and weight values \( w_{l}^p \), given as

\[
y_{l} = XnorDotProduct(a_{l-1}^p, w_{l}^p).
\]

Equation (4) shows that we are summing up 1s and 0s which is different from the original BCNN that sums up -1s and +1s as shown in Equation (1). The relationship between original output feature map pixel value \( y_{l0} \) and the revised \( y_{l} \) in our design can be expressed as Equation (5)

\[
y_{l0} = 1 \times y_{l} + (-1) \times (cnum_{l} - y_{l}) = 2y_{l} - cnum_{l},
\]

where \( cnum_{l} = FW \times FH \times DEP \), which is the total number of bitwise XNOR or multiplication for each output pixel. The difference between \( y_{l0} \) and the revised \( y_{l} \) can be compensated in normalization module, which will be discussed in Section 3.3.

Note that all the layers take a binary feature map of its previous layer as the input, except for the first layer. In our design, we rescale the input data to the range of \([-31,31]\) and use a 6-bit representation, which only results in a classification accuracy loss of <0.5%. Since the input image size is 3 x 32 x 32, the computational complexity of the first layer is not a dominating factor. The fixed-point dot product of a 6-bit signed input \( a_{0} \) and a 2-bit signed weight \( w_{1} \), denoted as \( FpDotProduct \) is implemented for the first layer.

\[
y_{1} = FpDotProduct(a_{0},w_{1})
\]

3.2 Comparator-based Normalization

The parameters subject to training can be considered as constant values in the inference stage. Therefore, we can combine the binarization (Equation (3)), normalization function (Equation (2)) and Equation (5) and simplify them into a modified sign function defined as

\[
NormBinarize(y_{l0},c) = \begin{cases} 1 & \text{if } y_{l0} \geq c, \\ 0 & \text{otherwise}, \end{cases}
\]

where \( c \) is a constant threshold derived by \( c = \sqrt{(cnum_{l} + \mu - \beta \sigma^{2} + \varepsilon y)} \times 0.5 \). Then we round \( c \) to the nearest integer for hardware implementation.

The impact of this reformulation on hardware implementation is that we now only need an LUTs-based comparator to implement both the normalization and the binarization functions. In addition, we only need to store one constant \( c \) for each output value rather than a set of parameters \( \mu, \sigma^{2}, \beta \) and \( \gamma \).
4. Architecture Design and Optimization

4.1 Architecture Overview

The binary nature of the BCNN enables us to map all the weights, feature maps, and reference values (for normalization) onto the on-chip block RAMs (BRAMs) in a single FPGA. This eliminates any DRAM access latency and dramatically reduces the energy consumption of the system compared to the existing work relying on off-chip storage [1] [3] [12].

Figure 3 shows the overall architecture of the proposed BCNN accelerator. The binary convolutional kernel in each layer is followed by a NormBinarize (NB) kernel with or without a max-pooling (MP) kernel. All of the kernels are highly parallelized with an optimized number of processing elements (PEs) and operate in a single instruction multiple data (SIMD) fashion. A streaming architecture is enabled by using double-buffering-based memory channels to handle the data flow between adjacent layers. Each PE in the binary convolutional kernel handles an XNOR dot product operation, which is the core operation in both convolutional and fully-connected layers. The PEs interface with the BRAMs in parallel to read the weights concurrently.

4.2 Loop Unrolling

Note that the three nested loops in (1) that accumulate the XNOR output values along the three dimensions of a convolutional filter has loop-carried data dependency. Unrolling data-dependent loops is the same as architectural unfolding, which will improve throughput by increasing the level of temporal parallelism. This trades off more hardware resource with reduced loop latency. The unfolding factor is a critical architectural parameter in our design, denoted as $UF$. $UF$ has a maximum value of $WID \times HEI \times DEP$ in each layer.

Differently, the calculation of the pixel values along the three dimensions of an output feature map has no loop-carried data dependency. Unrolling independent loops is equivalent to creating spatial parallelism in the architecture to improve throughput. In our design, we fully unroll these independent loops to maximize throughput. We denote the unrolling factor of independent loops as $P$. Maximizing $P$ generates a massively parallelized PE array by utilizing the abundant LUT resources on the FPGA. Note that the PEs in the same layer are identical, but they could be different in size across layers.

4.3 Pipelining

Deep pipelining is applied in the proposed architecture to further enhance the temporal parallelism and maximize the system throughput. Note that the queuing time to feed in the next data is the inverse throughput, which is referred to as initial interval $I$ in this paper. If there is a loop existing in the data path, the minimum initial interval will be limited by the physical loop latency in the hardware. With pipelining, we can feed in next data whenever it is possible with less queuing time. In the case of fully pipelining ($I = 1$), we can feed in new data every clock cycle.

4.4 Throughput Modeling

If we only perform one XNOR operation and one accumulation in each clock cycle, the total execution time $Cycle_{conv}$ in terms of clock cycles of a convolutional layer can be formulated as

$$Cycle_{conv} = WID \times HEI \times DEP \times FW \times FH \times FD,$$  

(8)

where $WID, HEI$, and $DEP$ denotes the width, height, and depth of a convolutional filter, and $FW, FH$, and $FD$ denotes the width, height and, depth of an output feature map, respectively.

When architectural unfolding is applied in performing the XNOR dot product operation in each PE, $Cycle_{conv}$ will be divided by $UF$. Similarly, when spatial parallelism is applied to create PE arrays for processing $P$ output pixels in parallel, $Cycle_{conv}$ will be further reduced by $P$ times. The same PE array is reused to calculate the output feature maps with pipelining applied, which contributes $I$ cycles for the most inner loop. Thus the throughput of the convolutional layer can be formulated as

$$throughput_{conv} = \frac{UF \times P}{Cycle_{conv}} \times \frac{1}{I} \times freq.$$  

(9)

where $freq$ is the system frequency. The part in Equation (9) unrelated to $freq$ is the estimated cycle count $Cycle_{est}$ in a convolutional layer.

$$Cycle_{est} = \frac{Cycle_{conv}}{UF \times P} \times I.$$  

(10)

In the proposed accelerator architecture, we use the double buffering scheme to further enhance the spatial parallelism of the system as shown in Figure 3. The computation of each layer is triggered at the same time and switches between two phases. Specifically, one channel of $fmap_{l-1}$ is used as the input of the $L^{th}$ layer while the $L^{-1}^{th}$ layer is writing new outputs into the other $fmap_{l-1}$ channel. When all the layers finish processing, the memory buffers switch and the next processing phase is triggered.

Therefore, the overall system-level throughput can be formulated as

$$throughput = \frac{\max(C_1, C_2, C_3,..., C_k)}{freq},$$  

(11)

where $C_k$ is the execution time of the $L^{th}$ layer in the proposed accelerator architecture. The system throughput can be maximized with optimal hardware utilization when all the layers have equal execution time. In the case that the $L^{th}$ layer has longer execution time than other layers, one can always increase the parallelism of the $L^{th}$ layer while decreasing it in other layers to gain throughput with minimum overhead in resource usage. Since the convolutional layers take up over 95% of the computation, we only emphasize the optimization of convolutional layers in this section. Fully-connected is easy to match up the system throughput using the same techniques.
5. FPGA Implementation

In this section, we specify our strategy of mapping different computing units to optimize the FPGA resource utilization.

5.1 PE Unit

The block diagram of a PE is shown in Figure 4. A PE unit handles the XNOR dot product computation of a weight vector and a feature map vector from the previous layer. The vectors are fed into an array of 2-input XNOR gates followed by a parallelized bit-count logic for accumulation. Since both the XNOR gates and the bit-count logic take binary values as input, the PEs can be efficiently implemented using the abundant LUT resources. This is the key to enabling massive computing parallelism on an FPGA. Note that the number of XNOR gates in each PE is the same as the unfolding factor $UF$ of the current layer. By accumulating the PE output, a pixel value of the output feature map can be computed by the bit-count logic.

5.2 Computing Kernels

Figure 5 shows the architecture of the convolutional kernel followed by the MP and NB kernel. Each convolutional kernel has an array of PEs implemented by LUTs followed by an array of accumulators implemented by DSP48 slices. The number of PEs and DSP slices is equal to the spatial parallelism factor $P$. Each convolutional kernel thereby computes $P$ pixel values of the output feature map in parallel. Besides weight arrays, only intermediate results of the accumulator outputs (bit-count results) within a single feature map are stored in BRAMs. Feature maps are mapped onto distributed RAMs.

For the convolutional layers 1, 3 and 5 without max-pooling, the outputs of accumulators are directly connected to the NB kernels. The hardware kernel of fully-connected layers is similar to Figure 5. Note that the max-pooling is performed in pipeline with the computation of feature maps in our implementation.

5.3 Memory

To read and write a large number of bits in the same clock cycle, we have to partition and reshape the memory arrays in the BCNN model. Partition essentially breaks down a large data array into smaller ones to fit in multiple BRAMs for parallel access. Reshaping basically redefines the depth and width of a single BRAM by grouping multiple words into a wider one. In our design, the weight and fmap arrays are mapped onto BRAMs and distributed RAMs (registers), respectively. Since the maximum word length of a BRAM in a Virtex-7 FPGA is limited to 32 bits, we first reshape the weight array by 32 and then partition the weight arrays into several BRAMs to guarantee enough memory bandwidth for the required system throughput.

6. Experiment Results

We implemented the proposed accelerator architecture for the BCNN in [9] using the architectural parameters shown in Table 2. Regarding to Equation (10), we tune the parameters of $UF$ and $P$ to make $Cycle_{est}$ of each layer be around the same and assume fully pipeline is applied in each layer ($I = 1$). Specifically, computation along $FW$ and $FD$ is fully unfolded.

6.1 Design Environment

For this work, we use C-language to describe the accelerator architecture and Vivado HLS is used to produce the RTL codes. The Vivado Design Suite is used to map the design onto a Xilinx Virtex-7 XC7VX690 FPGA. The execution time in terms of clock cycles is reported by Vivado HLS and the system frequency is reported by Vivado Design Suite. We notice a large discrepancy of LUTs usage between the synthesis reports in Vivado HLS and Vivado Design Suite. The resource utilization and power consumption are reported in Vivado Design Suite after implementation.

6.2 Implementation results

As shown in Table 2, the real execution time $Cycle_{r}$ given by synthesis report for each layer is very close to formulated $Cycle_{est}$.

Table 2. Optimization parameter for each layer

| Layer | $UF$ | $P$ | $Cycle_{comp}$ | $Cycle_{est}$ | $Cycle_{r}$ |
|-------|------|-----|----------------|--------------|-------------|
| Conv 1 | 27   | 32  | 3538944        | 4096         | 5233        |
| Conv 2 | 384  | 16  | 150994944      | 12288        | 12386       |
| Conv 3 | 768  | 8   | 150994944      | 12288        | 13329       |
| Conv 4 | 1536 | 8   | 150994944      | 12288        | 14473       |

Table 3. FPGA resource utilization summary

| Resource | LUTs | BRAMs | Registers | DSP |
|----------|------|-------|-----------|-----|
| Used     | 342126 | 1007  | 70769     | 1096|
| Available| 433200 | 2060  | 607200    | 2800|
| Utilization/% | 78.98 | 48.88 | 14.30     | 39.14|
The bottleneck layer is layer 6. With maximum system frequency of 90 MHz, the throughput of our BCNN with FPGA accelerator is 6218 fps. Top-1 accuracy rate is 87.8%, with only 0.3% accuracy drop compared with its counterpart on Theano.

We use the IP cores generated by Vivado HLS to implement our design layer by layer in Vivado Design Suite. For each layer, it contains the initialization of feature map from the previous layer and all the computation in the current layer. This scheme can make sure that we only count the resource utilization of each feature map once in the whole network. The overhead introduced by initialization can be negligible. Table 3 shows the resource utilization summary for the whole network. LUTs for logics are used for mapping PEs, max-pooling, normalization, and binarization. The feature maps of convolutional layers are mapped to distributed RAMs, resulting in LUTs consumption. This approach is much more efficient than mapping feature maps onto RAMs, which approximately consume 50% of RAMs to retain enough word-length for highly parallel read and write. The BRAMs usage is mostly consumed by all weight matrices. FFs are used by partial feature maps and constructing a deep pipeline. Around 30% of DSP slices are used by 1st layer to do multiplication for 6-bit signed inputs. For the other convolutional layers, DSPs are used for accumulation after PEs as shown in Figure 5.

Compared with FPGA implementations of floating-point or reduced-precision CNNs in Table 4, our implementation of BCNN results in 24x to 124x better performance in GOPS and 29x to 283x better in energy-efficiency.

In Figure 6, it compares the performance of the same BCNN by using Titan X GPU and our FPGA-based design. For GPU-based ones, the baseline kernel is non-optimized one for floating-point and the XNOR kernel is optimized for BCNN [9]. GPU accelerator is apparently sensitive to different workload (batch size here), but FPGA-based won’t. Our design outperforms GPU baseline both on throughput and energy efficiency. More importantly, even compared with an optimized XNOR kernel which is reported as the best GPU performance by far, it can outperform GPU counterpart both on energy efficiency and throughput by 75x and 8.3x for a small batch size of 16. On the other hand, our design can match up the throughput with 9.5x better energy-efficient processing a large batch size of 512.

Thus, in terms of applications such as processing online individual requests (in small batch size), FPGA-based solution can outperform GPU-based one for both throughput and energy efficiency. For processing static data (in large batch size), the proposed solution is on a par with a Titan X GPU in terms of throughput while delivering much higher energy efficiency.

7. Conclusion

In this paper, we propose an optimized accelerator architecture tailored for BCNNs and demonstrated for the 1st time that the FPGA-accelerated BCNN solution can greatly outperform a Titan X GPU in terms of both throughput and energy efficiency for processing accurate image classification tasks. The proposed FPGA-accelerated BCNN is 8.3x faster and 75x more energy-efficient than a Titan X GPU for processing online individual requests (in small batch size). For processing static data (in large batch size), the proposed solution is on a par with a Titan X GPU in terms of throughput while delivering 9.5x higher energy efficiency. Thus, BCNNs are ideal for efficient hardware implementations on FPGAs regardless of different workload. Bitwise operations in a BCNN enable the efficient hardware mapping of convolution kernels onto LUTs. Architectural unfolding, parallelism, and pipelining are the keys to enable massive parallelism and a high computing throughput. Building memory channels across layers with data-flow control allows for a streaming architecture with further enhancement on spatial parallelism.

8. Reference

[1] Zhang, C., Li, P., Sun, G., Guan, Y., Xiao, B., & Cong, J. 2015. Optimizing fpga-based accelerator design for deep convolutional neural networks. In Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 161-170.

[2] Krizhevsky, A., Sutskever, I., & Hinton, G. E. 2012. ImageNet classification with deep convolutional neural networks. In Advances in neural information processing systems, 1097-1105.

[3] Farabet, C., Martini, B., Corda, B., Akselrod, P., Culurciello, E., & LeCun, Y. 2011. Neuflow: A runtime reconfigurable dataflow processor for vision. In Cypr 2011 Workshops, 109-116.

[4] Suda, N., Chandra, V., Dasika, G., Mohanty, A., Ma, Y., Vrudhula, S., Seo, J.S., and Cao, Y. 2016. Throughput-Optimized OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks. In Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 16-25.

[5] Courbariaux, M., Bengio, Y., and David, J. P. 2015. Binaryconnect: Training deep neural networks with binary weights during propagations. In Advances in Neural Information Processing Systems, 3123-3131.

[6] Sung, W., Shin, S., and Hwang, K. 2015. Resiliency of Deep Neural Networks under Quantization. arXiv preprint arXiv:1511.06488.
[7] Cheng, Z., Soudry, D., Mao, Z., and Lan, Z. 2015. Training Binary Multilayer Neural Networks for Image Classification using Expectation Backpropagation. *arXiv preprint arXiv:1503.03562*.

[8] Kim, M. and Smaragdis, P. 2016. Bitwise neural networks. *arXiv preprint arXiv:1601.06071*.

[9] Courbariaux, M. and Bengio, Y. 2016. Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1. *arXiv preprint arXiv:1602.02830*.

[10] Rastegari, M., Ordonez, V., Redmon, J., and Farhadi, A. 2016. XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. *arXiv preprint arXiv:1603.05279*.

[11] Ioffe, S., & Szegedy, C. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*.

[12] Qiu, J., Wang, J., Yao, S., Guo, K., Li, B., Zhou, E., Yu, J., Tang, T., Xu, N., Song, S. and Wang, Y. 2016. Going deeper with embedded fpga platform for convolutional neural network. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 26-35.

[13] LeCun, Y., Bengio, Y., & Hinton, G. 2015. Deep learning. *Nature*, 521(7553), 436-444.

[14] Bengio, Y., Goodfellow, I. J., & Courville, A. 2015. Deep learning. *An MIT Press book in preparation. http://www.iro.umontreal.ca/~bengioy/dlbook*.

[15] Krizhevsky, A., Sutskever, I., & Hinton, G. E. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, 1097-1105.

[16] Simonyan, K., & Zisserman, A. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

[17] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. 1998. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278-2324.

[18] Anwar, S., Hwang, K., and Sung, W. 2015. Structured Pruning of deep convolutional neural networks. *arXiv preprint arXiv:1512.0857*