A GPU-Outperforming FPGA Accelerator Architecture for Binary Convolutional Neural Networks

YIXING LI, Arizona State University
ZICHUAN LIU, Nanyang Technological University
KAI XU, Arizona State University
HAO YU, Southern University of Science and Technology
FENGBO REN, Arizona State University

FPGA-based hardware accelerators for convolutional neural networks (CNNs) have received attention 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 article, 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 fully mapped FPGA accelerator architecture tailored for bitwise convolution and normalization that features massive spatial parallelism with deep pipelines stages. A key advantage of the FPGA accelerator is that its performance is insensitive to data batch size, while the performance of GPU acceleration varies largely depending on the batch size of the data. Experiment results show that the proposed accelerator architecture for binary CNNs running on a Virtex-7 FPGA is 8.3× faster and 75× more energy-efficient than a Titan X GPU for processing online individual requests in small batch sizes. For processing static data in large batch sizes, the proposed solution is on a par with a Titan X GPU in terms of throughput while delivering 9.5× higher energy efficiency.

CCS Concepts: • **Hardware → Hardware accelerators**; • **Computer systems organization → Neural networks**;

Additional Key Words and Phrases: FPGA, hardware acceleration, deep learning, convolutional neural network, binary neural network, high-throughput, energy efficiency

ACM Reference format:
Yixing Li, Zichuan Liu, Kai Xu, Hao Yu, and Fengbo Ren. 2018. A GPU-Outperforming FPGA Accelerator Architecture for Binary Convolutional Neural Networks. *J. Emerg. Technol. Comput. Syst.* 14, 2, Article 18 (July 2018), 16 pages.
https://doi.org/10.1145/3154839

1 INTRODUCTION

Convolutional neural networks (CNNs) have become a popular machine-learning engine for many image-related data analytics [15, 16, 20, 27], such as image classification, face detection,

This work, by Arizona State University and Nanyang Technological University, was supported by Cisco Research Center (CG#594589) and Singapore MOE Tier-2 (MOE2015-T2-2-013), respectively.

Authors' addresses: Y. Li, K. Xu, and F. Ren, 699 S. Mill Avenue, # 553, Tempe, AZ 85281, USA; emails: {yixingli, kaixu, renfengbo}@asu.edu; Z. Liu, 50 Nanyang Ave, Singapore, 639798; email: zliu016@e.ntu.edu.sg; H. Yu, EE Department 1088 Xueyuan Rd., Shenzhen, 518055, China; email: yuh3@sustc.edu.cn.

ACM 1550-4832/2018/07-ART18 $15.00
https://doi.org/10.1145/3154839

object tracking, and so on. 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 CNNs 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, 26], 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 10-class classification problem by 75% [18]. Reference [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 Reference [6] demonstrate the successful use of binary and ternary $(-1, 0, +1)$ weights in a CNN, respectively. But, they both have non-binary activations. As one step forward, EBP [7], Bitwise DNNs [8], and the BCNN in Reference [9] successfully exploit both binary weights and activations. In particular, the BCNN in Reference [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 with up to 96.8% reduced network sizes with minimum accuracy loss when comparing to their full-precision counterparts. Therefore, it is believed that BCNN is a more hardware-friendly model with superior accuracy–complexity tradeoff.

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 system throughput is proportional to the computing parallelism and operating frequency, the theoretical throughput of GPU-based and FPGA-based CNN accelerators can be estimated on the first order based on device specifications. A Titan X GPU has 3072 CUDA cores, while a Virtex-7 FPGA has 3,600 DSP48 slices. For implementing a full-precision CNN, the computing parallelism of GPUs and FPGAs can be approximately the same. But, GPUs offer 5–10× higher frequency. As a result, FPGAs can hardly match up the throughput of GPUs for accelerating full-precision CNNs. Differently, for a BCNN, the operations in the convolution layers become bitwise XNORs and bit-count logic. A direct impact is that one can use LUTs instead of DSP48 slices to implement the bitwise operations on an FPGA. Hundreds of thousands of LUTs make it possible for a high-end FPGA to match up or surpass the throughput of a GPU, 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 a data center setting. Early research effort [9] shows that GPU can get 7× 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 article, we propose an optimized FPGA accelerator architecture tailored for BCNN. The proposed architecture was adopted to implement a nine-layer BCNN on a Xilinx Virtex-7 XC7VX690 FPGA, which achieves nearly state-of-the-art classification accuracy on CIFAR-10. The experiment results show that the FPGA implementation outperforms its optimized GPU counterpart with 75× higher energy efficiency and 8.3x higher throughput for processing a small batch size of 16 images (e.g., from individual online request). For processing a large batch size of 512

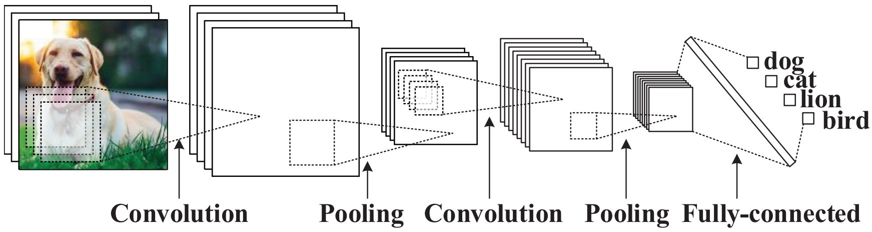


Fig. 1. Convolutional neural network.

images (e.g., from static data), the FPGA implementation achieves comparable throughput with 9.5× higher energy efficiency compared with the GPU counterpart.

The contributions of this article are summarized as follows:

- We propose a throughput optimization model for the end-to-end mapping of general BCNNs.
- We demonstrate a 7.663-TOPS 8.2W FPGA accelerator for a BCNN that highly outperforms the GPU counterpart, especially for processing individual online requests in small batch size for the first time.
- We reveal that the impact of applying binary constraints in CNN training on FPGA acceleration is the enablement of massive computing parallelism of bitwise operations based on abundant LUT resources.
- We illustrate the advantage of the fully mapped architecture over the conventional inter-layer-folded architecture due to the higher ratio of active computing units and the capability of mapping all the weights on chip.
- We optimize the accelerator architecture to fully exploit both spatial and temporal parallelism across all the layers using architectural unfolding, pipelining, and dataflow 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 AND MOTIVATION

2.1 CNN

A CNN is a trained neural network model with high-level features extracted from input images [13]. A typical CNN model contains convolutional, pooling, and fully connected layers as shown in Figure 1. 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.

2.1.1 Convolution. The convolution layer is the core layer of a CNN. Taking an RGB image as an example, the input of each convolutional layer is a three-dimensional (3D) feature map with the size of $\boldsymbol{WID'} \times \boldsymbol{HEI'} \times \boldsymbol{DEP'}$ as shown in Figure 2. Each filter has a size of $\boldsymbol{FW} \times \boldsymbol{FH} \times \boldsymbol{FD}$, where $\boldsymbol{FW}$ and $\boldsymbol{FH}$ is the width and height of the reception field, respectively, and $\boldsymbol{FD}$ is equal to the depth $\boldsymbol{DEP'}$ of the input feature maps. N filters are constructed as a 4D tensor. The output feature maps Y in the size of $\boldsymbol{WID} \times \boldsymbol{HEI} \times \boldsymbol{DEP}$ are obtained from the spatial convolution along the first and the second dimensions of the input feature maps with the 3D filter $\boldsymbol{W}[\boldsymbol{n}]$. The operation in convolutional layers is defined as

$$Y\,[n]\,[w']\,[h'] = \sum_{w=0}^{FW-1} \sum_{h=0}^{FH-1} \sum_{d=0}^{FD-1} W\,[n]\,[w]\,[h]\,[d] \times fmap\,[w'+w]\,[h'+h]\,[d]. \qquad (1)$$

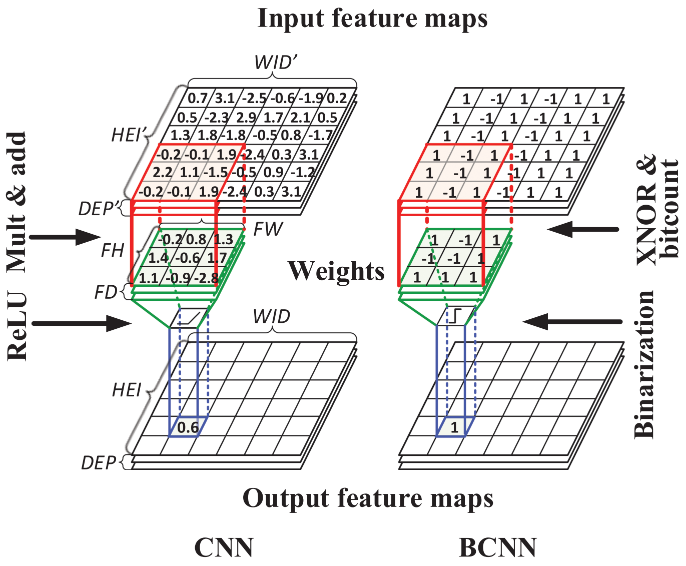


Fig. 2. A single layer in CNN and BCNN.

One should note that there is no data dependency for the calculation of each pixel across the entire output feature maps. Therefore, spatial parallelism can be applied in the hardware architecture to improve throughput. Differently, within the convolution operation for calculating each pixel, data dependency exists among the nested loops of summation in Equation (1). These data-dependent operations can be unfolded and pipelined in the hardware architecture to gain temporal parallelism and improve throughput.

2.1.2 Pooling. The pooling layer performs subsampling across a K × K contiguous region on the output feature map of convolutional layers. Pooling is used to pool out sensitive information critical to classification and eliminate insensitive information that is irrelevant. Also, pooling layers reduce an amount of trainable parameters in the network. There are two kinds of pooling methods that are commonly used in CNNs. One is max-pooling, which picks the maximum value of the pooling region. The other is average-pooling, which picks 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 needs to be applied to match the training process. Statistical reference values are counted across the whole training set as

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

where μ is the mean value and σ^2 is the variance with very a small constant ϵ to ensure a nonzero denominator. Note that γ and β scales and shifts the normalized values, respectively. Since μ, σ^2, ϵ, γ, and β are all constants in the inference stage, they can be precomputed to reduce the computation complexity of normalization.

2.1.4 Nonlinear function. Nonlinear function is an element-wise operation that performs on each neuron after the normalization in the convolutional layers and the fully connected layers. Two common nonlinear functions used in CNNs are Sigmoid and Rectified Linear Unit (ReLU) [13].

Table 1. Methods for Neural Network Compression

Methods	**Execution Stage**	**Compression Ratio**	**Inference**
Standard	Training	1×	full precision + full network
Quantizing	Post-training	Up to 3×	reduced precision + full network
Pruning	Training	Up to 5×	full precision + pruned network
BNN	Training	Up to 32×	binary + full network

2.2 BCNN

A BCNN is a CNN trained with binary constraints that results in binary weights and activations and a significant reduction in computation complexity. The convolution operation is the most time-consuming and computation intensive part of a CNN. In a BCNN, as shown in Figure 2, 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 exclusive NOR (XNOR). From a vector operation perspective, the convolution can be expressed as an XNOR dot-product operation as

$$Y\,[n]\,[w']\,[h'] = \sum_{w=0}^{FW-1} \sum_{h=0}^{FH-1} \sum_{d=0}^{FD-1} \bar{W}\,[n]\,[w]\,[h]\,[d] \oplus \overline{fmap}\,[w'+w]\,[h'+h]\,[d]\,. \tag{3}$$

Comparing to a real-valued CNN with a single−precision data format, the FPGA implementation of a BCNN requires much reduced logic and memory resources. Although, one should note that neither the inputs nor the outputs of the normalization and the pooling layers are binarized. The BCNN adopts a max-pooling scheme, which is thought to be more hardware friendly than average-pooling [14]. Since the weights and activations are constrained to either $+1$ or -1, the nonlinear function becomes an adjusted sign function, a.k.a. a Binarize function defined as

$$Binarize\,(z) = \begin{cases} 1 & if\ z \geq 0, \\ 0 & otherwise. \end{cases} \tag{4}$$

2.3 Compression Ratio and Accuracy of Compact CNNs

Table 1 shows some popular techniques for neural network compression. The baseline is a standard CNN trained by conventional techniques resulting in a 32-bit full precision network for inference. Experiment results show that simply quantizing the network parameters below 10 bits in the post-training stage will cause significant accuracy drop on CIFAR-10 classification task using the CNN model in Reference [9]. Although pruning the network has limited accuracy loss, the pruned network is still based on full-precision operations. The compression ratio achieved by pruning can be up to 5× [18], but the hardware resources needed for computing the remaining full-precision operations still have the same logic complexity.

However, the BCNN trained with binary constraints features the best compression ratio with superior accuracy performance. Reference [9] shows that BCNN can achieve same accuracy as the full-precision CNN on a 10-class classification task on CIFAR-10 dataset. Reference [19] demonstrates that with improved training technique, the BCNN only suffers from a 5% accuracy drop in terms of both top-1 and top-5 error for a 1000-class classification task based on an ImageNet dataset. In addition, the hardware resources needed for realizing the bitwise convolutions in BCNNs are just simple logic gates rather than multipliers. All of these suggest that BCNNs offer much superior tradeoff between complexity and accuracy and are ideal for efficient hardware implementation.

Table 2. BCNN Configurations

Name	**CONV-1**	**CONV-2**	**CONV-3**	**CONV-4**	**CONV-5**
Filter/weight	3 × 3 × 3	128 × 3 × 3	128 × 3 × 3	256 × 3 × 3	256 × 3 × 3
# of filters	128	128	256	256	512
Output size	128 × 32 × 32	128 × 16 × 16	256 × 16 × 16	256 × 8 × 8	512 × 8 × 8
Name	**CONV-6**	**FC-1**	**FC-2**	**FC-3**	
Filter/weight	512 × 3 × 3	8,192 × 1,024	1,024 × 1,024	1,024 × 10	
# of filters	512	—	—	—	
Output size	512 × 4 × 4	1,024	1,024	10	

2.4 Impact of Binarization on Hardware Acceleration

A Titan X GPU has 3,072 CUDA cores (one ALU per core) and can run at up to 1GHz, while a midrange Virtex-7 FPGA has 3,600 DSP48 slices and 433,200 LUTs and typically runs at around 100 to 200MHz. For mapping a full-precision or reduced-precision CNN, the two devices are on a par in terms of 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 5–10× lower frequency in general. As a result, the existing FPGA implementations of reduced-precision CNNs can hardly achieve comparable throughput to their GPU counterparts.

A BCNN offers large room for throughput improvement for both GPU-based and FPGA-based implementations. When using a tailored binary kernel on a GPU, a fully pipelined ALU in one CUDA core can process 32 bitwise operations per clock cycle. This increases the equivalent computing parallelism of a Titan X GPU to $3{,}072 \times 32 = 98{,}304$ for running a BCNN. However, for an FPGA-based BCNN, the bitwise operation can be efficiently mapped onto the abundant LUT resources. Since one six-input LUT can map 2.5 XNORs on average, the computing parallelism of a Virtex-7 FPGA is on the order of $433{,}200 \times 2.5 \approx 1{,}000{,}000$. Given the operation frequency difference, GPU- and FPGA-based BCNN implementations should have a similar level of throughput performance in a first-order estimation. The FPGA-based solution features much higher energy efficiency. It is also worth mentioning that GPUs can only achieve the theoretical peak 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 in terms of both throughput and energy efficiency. In the following sections, we present an FPGA-based BCNN accelerator and a benchmarking study that validate our hypothesis.

2.5 A BCNN ON CIFAR-10

To assess the practical performance of the proposed architecture, we use the BCNN on CIFAR-10 [9] as an example model for the FPGA implementation. The overall architecture of BCNN is shown in Table 2 [9]. It takes an RGB image with a size of $3 \times 32 \times 32$ as the input of the first layer. For each convolutional layer, the filter size is fixed as 3×3 with a stride and zero padding of 1 pixel each. The filter specification of each convolutional layer in Table 2 is denoted as the $WID \times HEI \times DEP$. Max-pooling is performed over a 2×2 window with a stride size of 2 followed by the convolutional layers of 2, 4, and 6. The last three layers are fully connected layers. Normalization is applied to all the layers, which is followed by binarization except for the last layer.

3 ALGORITHM REFORMULATION FOR EFFICIENT FPGA MAPPING

3.1 Binary-Encoded Convolution

When training the BCNN in Reference [9], the weights and activations are constrained to either $+1$ or -1. For efficient FPGA mapping, we encode $+1/-1$ as 1/0 in our design. In this way, it only takes 1 bit to store a weight 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}^{b} and the weight w_l^b, given as

$$y_l = XnorDotProduct\left(a_{l-1}^{b}, w_l^b\right). \tag{5}$$

Equation (5) sums up 1s and 0s, which is different from the original BCNN that sums up -1s and $+1$s in Equation (3). The relation between the original output feature map y_{lo} and the revised y_l in our design can be expressed as

$$y_{lo} = 1 \times y_l + (-1) \times (cnum_l - y_l) = 2y_l - cnum_l, \tag{6}$$

where $cnum_l = FW \times FH \times DEP$ is the total number of bitwise XNOR operations needed for each y_{lo}. The difference between y_{lo} and y_l is compensated in the normalization module in our design.

Note that all the layers take the binary feature map of its previous layer as the input except for the first layer. In our design, we rescale the input data within the range of $[-31, 31]$ and use a 6-bit fixed-point data format, which helps to reduce the resource utilization of non-binary operations at the cost of a limited classification accuracy loss of <0.5%, compared with its software counterpart in Theano. Since the input image size is $3 \times 32 \times 32$, 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 is denoted as

$$y_1 = FpDotProduct(a_0, w_1). \tag{7}$$

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 in Equation (4), the normalization function in Equation (2), and the value compensation in Equation (6) into a modified sign function defined as

$$NormBinarize\,(y_l, c_l) = \begin{cases} 1 & if\ y_l \geq c_l, \\ 0 & otherwise, \end{cases} \tag{8}$$

where c_l is a constant threshold derived by $c_l = (cnum_l + \mu - \beta\sqrt{\sigma^2 + \epsilon}/\gamma) \times 0.5$, and it is rounded to the nearest integer for hardware implementation.

The impact of the proposed reformulation on hardware implementation is that both the reformulated normalization and binarization functions can be efficiently implemented as a single LUT-based comparator. In addition, one only needs to store one threshold value c_l for each output value rather than a set of training parameters μ, σ^2, β, and γ.

3.3 BCNN Model Overview

We summarize the inference flow for the reformulated BCNN algorithm in Figure 3. The convolution in the first layer involves fixed-point dot product operations (*FpDotProduct*). Differently, bitwise XNOR dot product operations (*XnorDotProduct*) are used in all the other layers. Max-pooling (*MP*) is applied in layers 2, 4, and 6. Normalization and binarization are combined as a single function (*NormBinarize*), which is applied in all layers except for the output layer. The output layer ends with the normalization function *Norm* for classification.

```
{1. The first layer}
$y_1 \leftarrow FpDotProduct(a_0, w_1)$
$a_1 \leftarrow NormBinarize(y_1, c_1)$
{2. Remaining hidden layers}
for $l$ = 2 to 8 do
        $y_l \leftarrow XnorDotProduct(a_{l-1}^b, w_l^b)$
        If ($l$ = 2,4,6) then
                $y_l \leftarrow MP(y_l)$
        end if
        $a_l \leftarrow NormBinarize(y_l, c_l)$
end for
{3. Output layers}
$y_l \leftarrow XnorDotProduct(a_{l-1}^b, w_l^b)$
$a_l \leftarrow Norm(y_l, c_l)$
```

Fig. 3. Pseudo code of the BCNN algorithm.

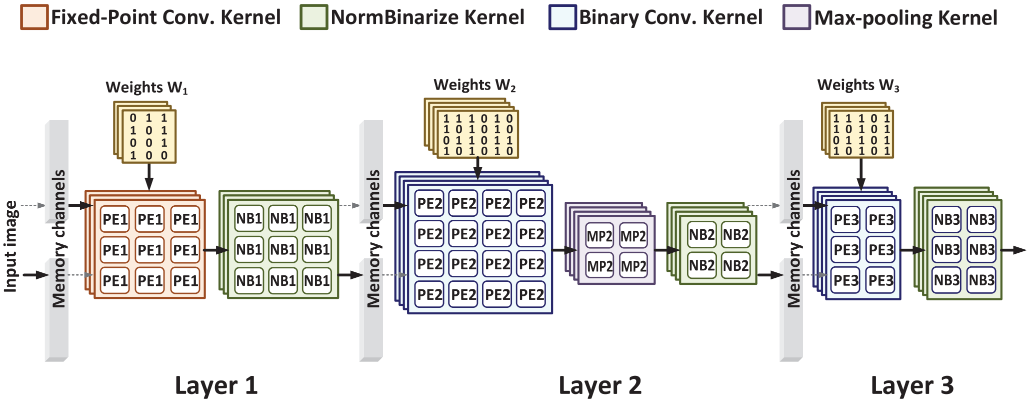


Fig. 4. Overview of the proposed accelerator architecture for BCNN.

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 comparing to the existing work relying on off-chip storage [1, 3, 12, 21].

Figure 4 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 dataflow 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.

```
for (dep=0; dep<DEP; dep++) {
 for (wid=0; wid< WID; wid++) {          Data-independent loops → unrolling factor P
  for (hei=0; hei<HEI; hei++) {
   for (d=0; d<FD; d++) {
    for (w=0; w<FW; w++) {               Data-dependent loops → unrolling factor UF
     for (h=0; h<FH; h++) {
      fmapOut[dep][wid][hei] += ~(weights[dep][w][h][d] ^ fmapIn[wid+w][hei+h][d]);
}}}}}}
```

Fig. 5. Loop unrolling in convolutional layers of BCNN.

4.2 Architectural Parameters

4.2.1 Loop Unrolling. The pseudo code in Figure 5 shows the main operations of a convolutional layer in six nested for loops. Note that three inner loops in Figure 5 accumulates the XNOR output values along the three dimensions of a convolutional filter that has loop-carried data dependency. Unrolling data-dependent loops is the same as architectural unfolding. The unfolding factor of data-dependent loops is denoted as *UF*. *UF* has a maximum value of $FD \times FW \times FH$ in each layer.

However, the three outer loops in Figure 5 are computing the pixel values along the three dimensions of an output feature map, which has no loop-carried data dependency. We denote the unrolling factor of data-independent loops as *P*. *P* has a maximum value of $DEP \times WID \times HEI$ in each layer.

From the architecture perspective of Figure 4, *UF* indicates computing parallelism inside each PE while *P* represents the number of PEs in each layer (parallelism of PE arrays). Increase *UF* improves throughput by increasing the level of temporal parallelism. This trades off more hardware resource with improved computing parallelism inside each PE. Increase *P* is equivalent to creating spatial parallelism in the architecture to improve throughput. Maximizing *P* generates a massively parallelized PE array by utilizing the abundant LUT resources on the FPGA.

4.2.2 Pipelining. Loop 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 inversely proportional to throughput, which is referred to as initial interval *I* in this article. If there is a loop existing in the data path, then the minimum initial interval will be limited by the loop latency of the recursive architecture. With loop pipelining, we can feed in the next data whenever possible with the minimum initial interval. In the case of a fully pipelined implementation, we can feed in new data every clock cycle ($I = 1$).

4.3 Throughput Modeling and Optimization

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

$$Cycle_{conv} = WID \times HEI \times DEP \times FW \times FH \times FD, \tag{9}$$

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 to an *I*-cycle initial interval for the most inner loop. Thus, the throughput of the

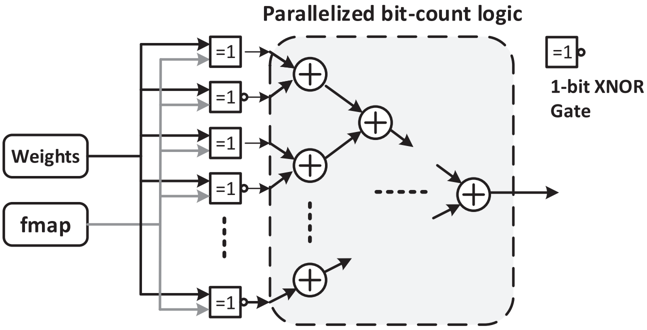


Fig. 6. Processing element (PE).

convolutional kernel with architectural optimization can be formulated as

$$throughput_{CONV} = \frac{UF \times P}{Cycle_{conv}} \times \frac{1}{I} \times freq, \tag{10}$$

where $freq$ is the system frequency. Note that $throughput_{CONV}$ is inversely proportional to the estimated cycle count $Cycle_{est}$ in a convolutional layer, defined as

$$Cycle_{est} = \frac{Cycle_{conv}}{UF \times P} \times I. \tag{11}$$

In the proposed accelerator architecture, we use a double buffering scheme to further enhance the spatial parallelism of the system as shown in Figure 4. The computation of each layer is triggered at the same time and alternates between two phases. Specifically, one channel of $fmap_{L-1}$ is used as the input of the Lth layer while the L-1th layer is writing new outputs into the other $fmap_{L-1}$channel. When both layers finish processing, the memory buffers swap, and the next processing phase is triggered. Therefore, the overall system-level throughput can be formulated as

$$throughput = \frac{freq}{max\ (C_1, C_2, C_3 \ldots, C_k)}, \tag{12}$$

where C_L is the execution time of the Lth layer in the proposed accelerator architecture. C_L can be either $Cycle_{est}$ for throughput modeling or $Cycle_r$ for evaluating real execution throughput. Equation (12) reveals that the bottleneck layer with maximum execution time determines the system throughput. Thus, the system throughput can be maximized with the optimal hardware utilization when all the layers have equal execution time ($C_1 = C_2 = C_3 = \cdots = C_k$). In the case that the Lth layer has longer execution time than other layers, one can always increase the parallelism of the Lth layer while decreasing that of 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. The fully connected layer can be easily optimized to match up the system throughput using the same principle.

5 FPGA IMPLEMENTATION

In this section, we present the strategy of mapping different computing units to maximize the FPGA resource utilization.

5.1 PE Unit

The block diagram of a PE unit is shown in Figure 6. A PE unit handles the XNOR dot product operation of a weight vector and a feature map vector from the previous layer. The vectors are fed

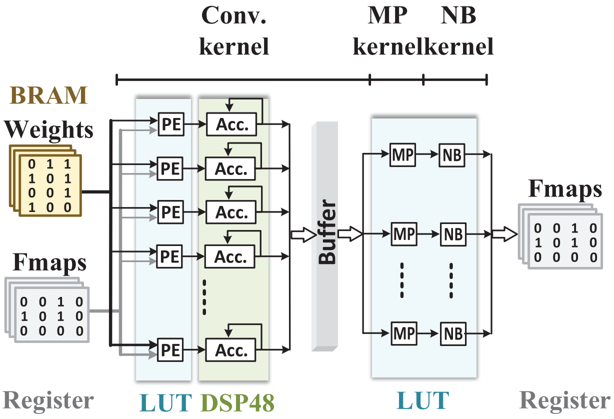


Fig. 7. The architecture of computing kernels and their FPGA mapping schemes.

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, the pixel value of an output feature map can be computed by the bit-count logic.

5.2 Computing Kernels

Figure 7 shows the architecture of the convolutional kernel followed by the Max-pooling and NormBinarize kernels. Each convolutional kernel has an array of PEs implemented using LUTs followed by an array of accumulators implemented using 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 the 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 that in Figure 7. 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.

Table 3. Optimized Parameters for Each Layer

Layer	UF	P	$Cycle_{conv}$	$Cycle_{est}$	$Cycle_r$
Conv 1	27	32	3538944	4096	5233
Conv 2	384	32	150994944	12288	12386
Conv 3	384	16	75497472	12288	12296
Conv 4	768	16	150994944	12288	13329
Conv 5	768	8	75497472	12288	12386
Conv 6	1536	8	150994944	12288	14473

Table 4. 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

6 EXPERIMENT RESULTS

We implement the proposed accelerator architecture for the BCNN in Reference [9] using the optimal architectural parameters shown in Table 3. We optimize the parameters of *UF* and *P* to make $Cycle_{est}$ of each layer approximately the same based on the throughput model in Equation (12). Each layer is also fully pipelined with an initial interval of $I = 1$. Note that the operations along the *FW* and the *FD* dimensions are fully unfolded for maximizing the throughput. By evaluating the throughput with (12), the actual throughput is 85% of the modeling throughput.

6.1 Design Environment

We use C language to describe the accelerator architecture. 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 after the implementation stage. We notice a large discrepancy of LUTs usage between the synthesis reports in Vivado HLS and Vivado Design Suite. For accurate results, the resource utilization and power consumption are reported in Vivado Design Suite after the implementation stage.

6.2 FPGA Implementation Results

As shown in Table 3, the real execution time $Cycle_r$ given by the synthesis report for each layer is well aligned with $Cycle_{est}$ estimated by our model in Equation (11). The throughput bottleneck is layer 6 in this case. Running at a system frequency of 90MHz, the FPGA-accelerated BCNN achieves an image processing throughput of 6,218 frames per second (FPS), which is the highest throughput for the same dataset reported by far. The top-1 accuracy rate is 87.8%, which is only 0.3% lower compared to the software model in Theano.

To reduce runtime, we adopt a bottom-up design strategy by synthesizing our design layer by layer in Vivado HLS and implementing the entire system in Vivado Design Suite. The overhead introduced by initialization is negligible. Table 4 shows the resource utilization summary for the entire BCNN implementation. LUTs are used for mapping all the computing kernels, including binary convolution, MP and NB kernels. Feature maps of convolutional layers are mapped onto distributed RAMs result in additional LUT consumption. The BRAM usage is mostly consumed by all the weight matrices. Flip-flops are used for storing feature maps and constructing a deep pipeline. Around 30% of the DSP slices are used by the first layer to perform fixed-point multiplication. For

Table 5. Results in Comparison with FPGA-Based and ASIC Accelerators

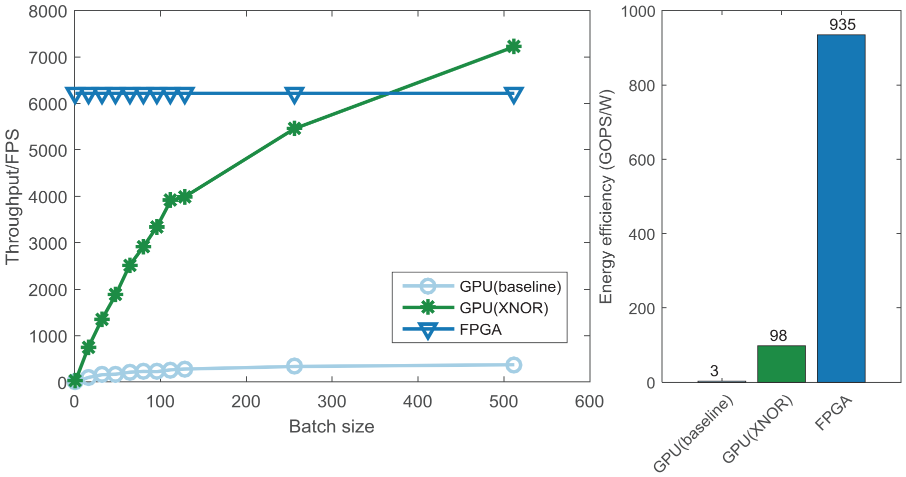
	Device	Clock (MHz)	Bit-width	GOPS	Power (W)	Energy Efficiency (GOPS/W)	Performance Density (GOPS/kLUT)	Latency (ms)
[3]	Virtex 6	200	16	147	10	14.7	0.98	-
[1]	Virtex 7	100	32 float	62	18.7	3.3	0.14	-
[12]	Zynq-7000	150	16	137	9.6	14.3	0.75	224.60
[4]	Stratix-V	120	8 ∼ 16	117.8	25.8	4.56	0.45	262.9
[22]	Arria-10	150	8 ∼ 16	645.25	21.2	30	4.01	47.94
[23]	Intel Xeon + Stratix V	200	32 float	123.48	13.18	9.37	0.62	263.27
[24]	Arria-10	385	fixed	1790	37.46	47.78	4.19	35.5
[21]	Zynq-7000	143	1 ∼ 2	207.8	4.7	44	4.43	5.94
[29]	YodaNN*	-	1	525	0.06	8600	-	-
[28]	Google TPU*	700	8	92000	40	2300	-	-
Ours	Virtex 7	90	1	7663	8.2	935	22.40	0.99

*Indicates ASIC design and results of TPU is its peak performance.

the rest of convolutional layers, DSP slices are used for accumulating PE outputs as shown in Figure 7. If the network size scales up, then the size and bandwidth of BRAMs will limit the maximum network size that can fit in our architecture. That is the limitation for a single FPGA-based implementation. However, the good thing is, if there are multiple FPGAs available, then it is feasible to use the proposed solution to map inference stage of a large network across multiple FPGAs.

Existing FPGA-based CNN implementations are compared in Table 5. To minimize the impact of different FPGA models on throughput, energy efficiency, and performance density defined as throughput normalized to resource utilization are used as the performance metrics for comparison. Compared with the FPGA implementations of floating-point or reduced-precision CNNs, our BCNN implementation achieves 4–124 × higher GOPS, 20–283 × better energy-efficiency, 5–160 × better performance density, and 36–266 × better latency. Even compared with the BCNN implementation in Reference [21], our work achieves 5 × better performance density in terms of GOPS/kLUT and 6 × better latency. Linked back to Equation (10), the throughput gains from unrolling factors *UF* and *P* is on the order of 10^3 times compared to a non-optimized design.

For the architecture maps a single layer of the BCNN at a time, we define it as inter-layer-folded architecture. Regardless of binary or non-binary CNN, all the reference work in Table 5 excluding Reference [28] (which is not clear for implementation detail) can be categorized as inter-layer-folded architecture. Take the BCNN work in Reference [21] as an example. The work in Reference [21] implements three kinds of computing kernels in hardware: floating-point convolution, binary convolution, and fully connected kernels. Since this reference work maps a single layer of the BCNN at a time, only one kind of computing kernels is active at a time. Such a time multiplexing scheme limits the system throughput due to the low hardware utilization. In our design, all the layers of the BCNN are mapped into a streaming architecture with optimized architectural parameters, and the data are flowing throughout the entire architecture in a deep pipeline. Therefore, all three kinds of kernels are simultaneously active and the ratio of active computing units is high. Besides, inter-layer-folded architecture consumes extra power for loading the weights from off-chip memory layer by layer in addition to the reported power. On the contrary, there is no such overhead in our architecture, since we fully map the network and trained parameters on chip. Roughly speaking, there will be 3 × throughput gain from fully mapped architecture. Besides, it is not practical for inter-layer-folded architecture to store all the weights on chip (even if the weights are binary). Assume all the weights are stored on chip, when the computation switches from one



* Titan X GPU has used up ~80% memory for batch size of 512

Fig. 8. Throughput and energy efficiency comparison with GPU implementations.

layer to another, it needs large MUXes to select from different memory banks (for binary weights). Since the required bit width of MUXes is of the order of magnitude of 10^5, it will result in much resource overhead in building these large MUXes. Thus, off-chip memory access is always required for the conventional inter-layer-folded architecture.

When comparing the same design between FPGA and ASIC implementation, generally speaking, the performance difference is expected to be one order of magnitude. The delay and power overhead in FPGA is mainly caused by interconnects [30]. Even without considering general performance gap between FPGA and ASIC designs, the proposed work achieves 14.6× better throughput but 9.2 × less energy efficiency than the BCNN ASIC accelerator in Reference [29]. We can see the tradeoff between throughput and energy efficiency in Reference [29]. Overall, the performance of the proposed work and Reference [29] are on the same level. If assuming a 10 × performance degradation factor from ASIC to FPGA, then the proposed work is 4 × better than Reference [28] in energy efficiency and has comparable throughput in terms of GOPS.

6.3 FPGA-Based Versus GPU-Based BCNN

Figure 8 compares the performance of the BCNN accelerated by a Titan X GPU and our FPGA-based design. For GPU acceleration, the baseline kernel is designed for floating-point computation, and the XNOR kernel is optimized for bitwise operations [9]. In the XNOR kernel, it concatenates 32 1-bit values into a 32-bit value. At the peak performance, each CUDA core can execute 32 bitwise operations per clock cycle. That is the reason why BCNN can also gain remarkable speedup on a GPU when using the XNOR kernel for compilation.

GPU acceleration is apparently sensitive to the size of workload (batch size here). One of the keys to achieving high performance in GPU computing is to hide the long latency of functional units by data-level interleaving especially when there are loop-carried data dependency existed in the algorithm. Only when the workload is large enough, a GPU is able to maintain high thread-level parallelism to achieve a high throughput. Differently, the FPGA-based solution is invariant to the batch size of data. Experiment results show that our design significantly outperforms the GPU acceleration using the baseline kernel in terms of both throughput and energy efficiency. Even compared with the GPU acceleration using the XNOR kernel, which is reported as the best

GPU-based CNN performance by far, our design achieves a 75 × better energy efficiency and an 8.3 × better throughput for processing data in a small batch size of 16. For processing data in a large batch size of 512 (the maximum size that fit into the GPU memory), our design can match the throughput of the GPU acceleration with a 9.5 × better energy efficiency.

Therefore, the FPGA-based BCNN solution is a clearly better choice for accelerating the data center applications that process online individual requests in small batch sizes. In a recent study conducted by Baidu, a dominant Internet company in China with 600 million active users, it is reported that the typical on-line prediction workload in terms of batch size is around 8 to 16 [25]. Such small workload is not enough for GPU to achieve its peak throughput performance. Thus, the FPGA-based solution is more superior in handling this kind of requests from individual users.

For processing static data in large batch sizes, the proposed solution is on a par with a Titan X GPU in terms of throughput while delivering much higher energy efficiency. This renders the FPGA-based solution a better choice for energy constrained applications, such as mobile-based advanced driver assistance systems (ADAS). In the ADAS application, a large batch of data needs to be processed for monitoring real-time road condition. In this case, both throughput and energy efficiency are essential and the FPGA-based solution can be deployed.

7 CONCLUSION

In this article, we propose an optimized accelerator architecture tailored for BCNNs. We demonstrate for the first time that the FPGA-based 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 BCNN accelerator running on a Virtex-7 FPGA is 8.3× faster and 75× more energy efficient than a Titan X GPU for processing individual online requests in small batch sizes. For processing static data in large batch sizes, the proposed solution is on a par with a Titan X GPU in terms of throughput while delivering 9.5× higher energy efficiency. Thus, BCNNs are ideal for efficient hardware implementations on FPGAs regardless of the size of workload. The bitwise operations in BCNNs allow for the efficient hardware mapping of convolution kernels using LUTs, which is the key to enable massive computing parallelism on an FPGA. Applying the optimal levels of architectural unfolding, parallelism, and pipelining based on the proposed throughput model is the key to maximizing the system throughput. Building memory channels across layers with dataflow control is the key to constructing a streaming architecture to further improve the throughput. Also, fully mapped architecture wins over the conventional inter-layer-folded architecture for better throughput due to the higher ratio of active computing units and the capability of mapping all the weights on chip.

ACKNOWLEDGMENTS

We acknowledge Mr. Skip Booth and Mr. Hugo Latapie from Cisco for fruitful research discussions. We also thank the Xilinx University Program for donating the FPGA boards.

REFERENCES

[1] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong. 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] A. Krizhevsky, I. Sutskever, and G. E. Hinton. 2012. ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*. 1097–1105.

[3] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun. 2011. Neuflow: A runtime reconfigurable dataflow processor for vision. In *Proceedings of the Conference on Computer Vision and Pattern Recognition 2011 Workshops*. 109–116.

[4] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J. S. Seo, and Y. Cao. 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] M. Courbariaux, Y. Bengio, and J. P. David. 2015. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in Neural Information Processing Systems*. 3123–3131.

[6] W. Sung, S. Shin, and K. Hwang. 2015. Resiliency of deep neural networks under quantization. arXiv:1511.06488.

[7] Z. Cheng, D. Soudry, Z. Mao, and Z. Lan. 2015. Training binary multilayer neural networks for image classification using expectation backpropagation. arXiv:1503.03562.

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

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

[10] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. 2016. XNOR-Net: ImageNet classification using binary convolutional neural networks. In *Proceedings of the European Conference on Computer Vision*. 525–542.

[11] S. Ioffe and C. Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on Machine Learning*.

[12] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, and Y. Wang. 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] Y. LeCun, Y. Bengio, and G. Hinton. 2015. Deep learning. *Nature* 521, 7553, 436–444.

[14] I. Goodfellow, Y. Bengio, and A. Courville. 2016. *Deep Learning*. MIT Press.

[15] A. Krizhevsky, I. Sutskever, and G. E. Hinton. 2012. ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*. 1097–1105.

[16] K. Simonyan and A. Zisserman. 2015. Very deep convolutional networks for large-scale image recognition. In *Proceedings of the 32nd International Conference on Learning Representations*.

[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] S. Anwar, K. Hwang, and W. Sung. 2017. Structured pruning of deep convolutional neural networks. *ACM Journal on Emerging Technologies in Computing Systems* 13, 3, Article 32, 18 pages.

[19] W. Tang, G. Hua, and L. Wang. 2017. How to train a compact binary neural network with high accuracy? In *Proceedings of the 31st AAAI Conference on Artificial Intelligence*. 2625–2631.

[20] P. Panda, A. Sengupta, and K. Roy. 2017. Energy-efficient and improved image recognition with conditional deep learning. *ACM Journal on Emerging Technologies in Computing Systems* 13, 3, Article 33, 21 pages.

[21] R. Zhao, W. Song, W. Zhang, T. Xing, J. H. Lin, M. Srivastava, R. Gupta, and Z. Zhang. 2017. Accelerating binarized convolutional neural networks with software-programmable FPGAs. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 15–24.

[22] Y. Ma, Y. Cao, S. Vrudhula, and J. S. Seo. 2017. Optimizing loop operation and dataflow in FPGA acceleration of deep convolutional neural networks. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 45–54.

[23] C. Zhang and V. Prasanna. 2017. Frequency domain acceleration of convolutional neural networks on CPU-FPGA shared memory system. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 35–44.

[24] J. Zhang and J. Li. 2017. Improving the performance of OpenCL-based FPGA accelerator for convolutional neural network. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 25–34.

[25] J. Ouyang, S. Lin, W. Qi, Y. Wang, B. Yu, and S. Jiang. 2016. SDA: Software-defined accelerator for large-scale DNN systems. In *Proceedings of the Hot Chips Conference*. 28.

[26] S. Han, J. Pool, J. Tran, and W. Dally. 2015. Learning both weights and connections for efficient neural network. In *Proceedings of the 28th International Conference on Neural Information Processing Systems (NIPS'15)*. 1135–1143.

[27] L. Wang, W. Ouyang, X. Wang, and H. Lu. 2015. Visual tracking with fully convolutional networks. In *Proceedings of the IEEE International Conference on Computer Vision*. 3119–3127.

[28] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, and R. Boyle. 2017. In-datacenter performance analysis of a tensor processing unit. arXiv:1704.04760.

[29] R. Andri, L. Cavigelli, D. Rossi, and L. Benini. 2016. YodaNN: An ultra-low power convolutional neural network accelerator based on binary weights. In *Proceedings of the 2016 IEEE Computer Society Annual Symposium on VLSI*. 236–241.

[30] D. Marković and R. W. Brodersen. 2012. *DSP Architecture Design Essentials*. Springer Science & Business Media.

Received March 2017; revised August 2017; accepted October 2017