Neurocomputing 000 (2017) 1–15

Contents lists available at ScienceDirect

Neurocomputing

journal homepage: www.elsevier.com/locate/neucom

FP-BNN: Binarized neural network on FPGA

Shuang Liang[a], Shouyi Yin[a,*], Leibo Liu[a], Wayne Luk[b], Shaojun Wei[a]

[a] Institute of Microelectronics, Tsinghua University, Beijing, China
[b] Department of Computing, Imperial College London, UK

ARTICLE INFO

Article history:
Received 10 December 2016
Revised 10 August 2017
Accepted 17 September 2017
Available online xxx

Communicated by Dr. Deng Cheng

Keywords:
Binarized neural network
Hardware accelerator
FPGA

ABSTRACT

Deep neural networks (DNNs) have attracted significant attention for their excellent accuracy especially in areas such as computer vision and artificial intelligence. To enhance their performance, technologies for their hardware acceleration are being studied. FPGA technology is a promising choice for hardware acceleration, given its low power consumption and high flexibility which makes it suitable particularly for embedded systems. However, complex DNN models may need more computing and memory resources than those available in many current FPGAs. This paper presents FP-BNN, a binarized neural network (BNN) for FPGAs, which drastically cuts down the hardware consumption while maintaining acceptable accuracy. We introduce a Resource-Aware Model Analysis (RAMA) method, and remove the bottleneck involving multipliers by bit-level XNOR and shifting operations, and the bottleneck of parameter access by data quantization and optimized on-chip storage. We evaluate the FP-BNN accelerator designs for MNIST multi-layer perceptrons (MLP), Cifar-10 ConvNet, and AlexNet on a Stratix-V FPGA system. An inference performance of Tera opartions per second with acceptable accuracy loss is obtained, which shows improvement in speed and energy efficiency over other computing platforms.

1. Introduction

As the computational ability of processors rapidly grows, training and testing deep neural networks (NNs) become much more feasible, which substantially boost the design of various models targeting applications such as computer vision [1–3], speech recognition [4,5], and even artificial intelligence (AI) for games against human beings [6,7]. Higher accuracy typically demands more complex models. Take ImageNet Large-Scale Vision Recognition Challenge (ILSVRC) as example, Krizhevsky et al. [8] achieved 84.7% top-5 accuracy in classification task in 2012 with a model including 5 convolution (CONV) layers and 3 fully-connected (FC) layers; He et al. [9] got a 95.1% result surpassing human-level classification performance (94.9% [3]) with a 22-layer model, and they won the 2015 competition for achieving an accuracy of 96.4% with a model depth of 152 [10]. Such model can take over 11.3 billion floating-point operations (GFLOPs) for the inference procedure, and even more for training.

These convolutional neural networks (CNNs) mostly consist of intensive multiplication and accumulation (MAC) operations. General-purpose processors execute these operations mostly sequentially, which leads to low efficiency. Graphics processing units (GPUs) can offer Giga to Tera FLOPs per second's (FLOP/s) computing speed due to their single-instruction-multiple-data (SIMD) architecture and high clock frequency. Therefore, researchers tend to use one or several GPUs to meet the model training demand [11] for quick development iterations. However, GPUs also suffer from a high energy cost – for a NVIDIA Tesla K40 GPU, the thermal design power (TDP) is 235 W [12]. Such power consumption can be tolerable for high-performance servers, but for embedded systems such as mobile devices, robots, etc., which are mostly powered by batteries, low power consumption becomes essential.

Field Programmable Gate Arrays (FPGAs) usually consume one order-of-magnitude less power than GPUs, while offering considerable speed-up over CPUs. Moreover, FPGAs offer more flexibility, since they are reconfigurable and support customizable data types, which can be useful in reducing resource utilization. There is much research on accelerating state-of-the-art NN models with FPGAs [13–15]. However, since most current FPGAs have limited resources (several dozen *M* bits of on-chip memory, several hundred to thousand digital signal processors (DSPs)), designers have to adopt techniques such as tiling to support many NN models, since most models have a large number of weights and MAC operations (Table 1). Furthermore, memory bandwidth can be a bottleneck during the data loading stage for some wide data-dependency pattern such as FC layers [15].

* Corresponding author.
E-mail addresses: s-liang11@mails.tsinghua.edu.cn (S. Liang), yinsy@tsinghua.edu.cn (S. Yin), liulb@tsinghua.edu.cn (L. Liu), w.luk@imperial.ac.uk (W. Luk), wsj@tsinghua.edu.cn (S. Wei).

https://doi.org/10.1016/j.neucom.2017.09.046
0925-2312/

Table 1
Summary of weight and MAC number of popular CNNs [21].

Model	LeNet-5	AlexNet	VGG-16	GoogLeNet v1	ResNet-50
Weights	60 K	61 M	138 M	7 M	25.5 M
MACs	341 K	724 M	15.5 G	1.43 G	3.9 G

To improve resource usage, there are several ways of compressing models to smaller sizes, such as gaining sparsity of network connections and narrowing data bit-width [15–17]. Binarization is a promising method to compress the NN models, which can directly shrink the bit-width of inputs and weights from 32 bit (single-precision floating-point) to a single bit. Recently, Courbariaux et al. [18] introduced a method to train binarized neural networks (BNNs) over MNIST, Cifar-10 and SVHN [19] datasets, with near state-of-the-art accuracy. Shortly after that, Rastegari et al. [20] announced they successfully trained ImageNet models with BNN-based XNOR-Net method with an accuracy of 12.4% below the full precision AlexNet, and provides a 58 times speed-up and 32 times model size compression. The emergence of binarized models makes it feasible to implement a system on FPGAs with much higher performance than floating-point versions. This motivates us to design a method to take a given BNN model and generate the datapath logic and data management pattern on FPGA based to an optimization metric, which forms an accelerator system targeting Tera operations per second's(TOP/s) throughput speed.

In this paper, we introduce FP-BNN, a BNN acceleration system design on FPGA, with related optimizations. The contributions of this paper are as follows:

- An analytical resource aware model analysis (RAMA) to assess the resource cost, to help on-chip system architecture design.
- A datapath design with multipliers replaced by XNOR, popcount and shifting operations for BNNs, and a compression tree generation method for more efficient popcount.
- An optimized data managing pattern with parameter quantization and on-chip storage strategy.
- A demonstration with popular small (MNIST MLP and Cifar-10 ConvNet) and large (AlexNet) models implemented on FPGA in binarized style, achieving a performance of TOP/s with high power efficiency.

The rest of the paper is organized as follows. Section 2 reviews the basic concepts of CNN and BNN and discuss on the related works. Section 3 describes the RAMA method. Section 4 presents the system design and the details of each processing element (PE). Section 5 explains how we tile and schedule the large computing task onto our system. Section 6 covers a data quantization to compress the model, and introduces the on-chip design of the memory system. Evaluation will be discussed in Section 7, and conclusion will be given in Section 8.

2. Background

In this section, we will first provide an overview of the basic concepts of CNN, and then explain how a binarized NN works. Based on these concepts, we take a brief overview of related efforts and discuss them.

2.1. Basics of CNN

Fig. 1 shows a typical CNN model structure [22]. A CNN model usually consists of CONV layer, FC layer and Pooling (POOL) layer, forming a trainable network. *CONV layer*: The CONV layer realizes a filter-like process, which uses a $K \times K$ weight kernel W to convolve the input feature-map (fmap) I in a sliding-window manner with a stride of S. This can be expressed as:

$$\mathbf{A}_n^{(l)}(i,j) = \mathbf{B}^{(l)}(n) + \sum_{m=1}^{N_{in}^{(l)}} \mathbf{W}^{(l)}(m,n) \otimes \mathbf{I}_m^{(l)}(i,j), \tag{1}$$

where $\otimes$ is defined as convolution, which equals to K^2 element-wise multiplications with accumulation (K stands for the kernel size):

$$\mathbf{X} \otimes \mathbf{Y} = \sum_{i=1}^{K} \sum_{j=1}^{K} \mathbf{X}(i,j) \cdot \mathbf{Y}(i,j) \tag{2}$$

FC layer: The FC layer will operate a linear transformation on the input 1-D vectors with a weight matrix. The pattern of the input-output network is fully-connected, which is how it got its name. This process can be shown as:

$$\mathbf{A}^{(l)}(n) = \mathbf{B}^{(l)}(n) + \sum_{m=1}^{N_{in}^{(l)}} \mathbf{I}^{(l)}(m) \cdot \mathbf{W}^{(l)}(m,n) \tag{3}$$

POOL layer: The POOL layer realizes a "down-sampling" operation, which compresses the input images into smaller scales. We take the most common max-POOL as an example, which extracts the maximum value from the $K \times K$ kernel window as the output:

$$\mathbf{A}^{(l)}(i,j) = \max[\mathbf{I}_{K \times K(i,j)}^{(l)}] \tag{4}$$

Activation Layer: Just like biological neurons, we say they are "firing" once the key value exceeds the threshold and are "silent" if not. Various activation functions are implemented in neural network designs to imitate the neurological behaviour such as ReLU, tanh, sigmoid, etc., which also introduce non-linearity to the networks.

Batch Normalization (BN) layer: Since the distribution of each layer's input can fluctuate during training, Batch Normalization [23] is introduced to speed up training. For a d-dimensional input vector $x = (x^{(1)}, x^{(2)}, \ldots, x^{(d)})$, we can normalize each dimension with:

$$\widehat{x}^{(k)} = \frac{x^{(k)} - E[x^{(k)}]}{\sqrt{Var[x^{(k)}]}} \tag{5}$$

After that, for each activation $x^{(k)}$, we should scale and shift the normalized value to achieve an identity transform:

$$y^{(k)} = \gamma^{(k)} \widehat{x}^{(k)} + \beta^{(k)} \tag{6}$$

where $\gamma^{(k)}$ and $\beta^{(k)}$ are to be learned during the training process. The whole process is described in Algorithm 1.

Algorithm 1 Batch Normalization [23].

Require: A mini-batch of input values: $\mathcal{B} = \{x_i\}, i = 1 \sim m$; Initialized parameters: γ, β.
Ensure: Updated γ, β; Output $y_i = BN_{\gamma,\beta}(x_i), i = 1 \sim m$.
$\mu_{\mathcal{B}} = \frac{1}{m} \sum_{i=1}^{m} x_i$; //Get mini-batch's mean
$\sigma_{\mathcal{B}}^2 = \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2$; //Get mini-batch's variance
$\hat{x}_i = \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \varepsilon}}$; //Normalize
$y_i \equiv BN_{\gamma,\beta}(x_i) = \gamma \hat{x}_i + \beta$; //Scale and shift

2.2. Training a CNN

A given CNN model with initialized parameters should be trained on a certain dataset in order to approximate the ideal

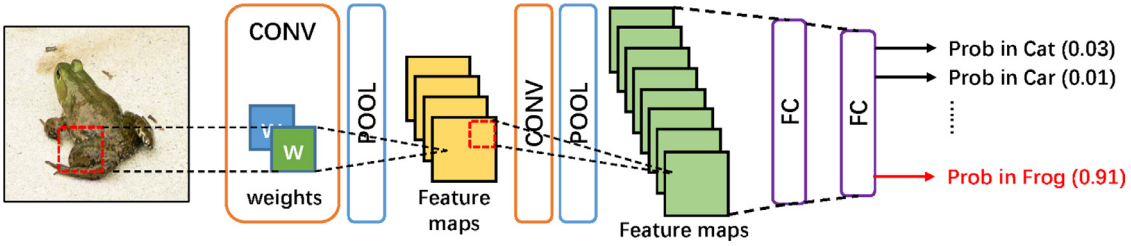
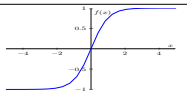
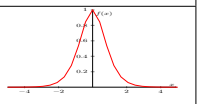
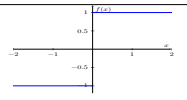
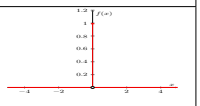
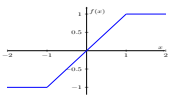
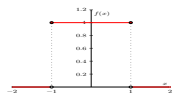


Fig. 1. A typical CNN model structure.

Table 2
Comparison between activation function Tanh, sign and HTanh.

Operation	Function plots	Derivative plots
$Tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$		
$sign(x) = \begin{cases} +1 & x \geq 0 \\ -1 & x < 0 \end{cases}$		
$HTanh(x) = \begin{cases} +1 & x > 1 \\ x & -1 \leq x \leq 1 \\ -1 & x < -1 \end{cases}$		

model for ground-truth results. The most commonly used training method is Back-Propagation (BP) training, which consists of two stages:

(1) *Forward propagation (Inference)*, which leads the input data going through the network to get an output result;
(2) *Back propagation*, which calculates the error between output and ground-truth labels with a defined *loss function C*, and then propagates the gradient of each layer's output function backwards to update the weights in order to minimize the loss function for the next training iteration.

Detailed derivation can be found in [24]. Since the overall process is compute-intensive, high-performance servers with accelerators such as GPUs are often used in training. Then the pretrained models can be used in many real-time scenarios by going through inference process only with minor changes, which can be implemented on many embedded hardware platforms.

2.3. How BNN works

The essential idea of BNN is to constrain both weights and activations to +1 and −1 [18]. The binarization method can be done in either stochastic or deterministic way, and the latter is often realized by the Sign function:

$$x_b = Sign(x) = \begin{cases} +1 & x \geq 0 \\ -1 & x < 0 \end{cases} \tag{7}$$

The problem is that during the training process, the derivative of the Sign function is almost *zero* everywhere (as shown in Table 2), resulting in an incompatibility with the BP training process. Hinton [25] introduced a "straight-through estimator" to cope with this problem. Courbariaux et al. [18] used a similar estimator in a deterministic way, which can be seen as a *hard tanh (HTanh)* function:

$$est\,(Sign(x)) = H\mathrm{Tanh}(x) = \begin{cases} +1 & x > 1 \\ x & -1 \leq x \leq 1 \\ -1 & x < -1 \end{cases} \tag{8}$$

Assume the required gradient is $\frac{dC}{du}$, and $a = Sign(u)$, then we will have the estimator of the gradient:

$$est\left(\frac{dC}{du}\right) = \frac{dC}{da} \cdot \frac{dSign(u)}{du} = \begin{cases} \frac{dC}{da} & -1 \leq u \leq 1 \\ 0 & otherwise \end{cases} \tag{9}$$

Since BN layers have the effect of avoiding internal covariate shift, which can accelerate the training process and reduce the impact of binarization, [18] introduces BN layers in their BNN models. To deal with the large amount of multiplications in BN, they replace them with shift operations to get a Shift-Based BN (SBN). This can largely reduce the computing resource cost with only a small loss of precision – which actually can be healed through the training process. The SBN replacement can be described as Eq. (10) where *sal*(*x*, *y*) means an arithmetic left shift to *x* by *y* bits:

$$x \cdot y \approx sal[x, round(\log_2|y|)] \cdot sign(y) \tag{10}$$

2.4. Related work

To accelerate an NN model in embedded hardware, spade husbandry should be taken. There has been many efforts deploying CNN models in hardware. Farabet et al. [26] designed a 3 CONV layers +5 FC layers simple face detection system on FPGA with 10 frames (512 × 384) per second's performance. Zhang et al. [14] proposed a nested-loop model to describe CNN, and accelerates CONV layers only under the guidance of a roofline model. Qiu et al. [15] realized an even deeper VGG model on FPGA. Most of these previous designs store weights and fmaps off-chip since their size is too large for on-chip storage. As a result, the dataflow bandwidth is limited and frequent off-chip memory access happens. So some

Please cite this article as: S. Liang et al., FP-BNN: Binarized neural network on FPGA, Neurocomputing (2017), https://doi.org/10.1016/j.neucom.2017.09.046

designs support dedicated memory cache for on-chip data reuse [27–29], but the increase of memory placement means fewer arithmetic resources since chip area is limited.

Clearly a small model that supports high accuracy and high performance is ideal. One method is to exploit the sparsity inside the model by pruning off connections [16,30,31]. Another method is to reduce bit-width of operations. Much previous work took a quantized fixed-point strategy to the on-chip data [15,27,28,32,33] presented a detailed analysis pointing out that for small models such as MNIST and Cifar-10, the weights can be quantized to 4 bits, while for large models such as AlexNet, 8 bits would be necessary.

Recently, some efforts successfully reduced the bit-width of weights to 2 bits such as ternarized weight NN (TWN) [34,35], or even 1-bit binarized weight NN (BWN) [20]. Moreover, activations can be reduced to 2 bits [36–38] or even 1 bit (BNN) with little loss for small datasets [18,20]. These results stimulate hardware development. YodaNN [39] designed a UMC 65-nm ASIC targeting BWN with 1.5 TOP/s. Alemdar et al. [37] implemented ternarized NN (TNN) on FPGA with a speed of 600 GOP/s for MNIST MLP and 200 GOP/s for Cifar-10 ConvNet under 250 MHz clock, and on ST 28 nm ASIC with doubled throughput and around 300 mW power consumption under 500 MHz clock. Zhao et al. [40] implemented a BNN on FPGA with the help of high-level synthesis (HLS) tool, and get a 200 GOP/s performance for Cifar-10 ConvNet. In addition, Umuroglu et al. [41] also proposed a BNN design targeting small datasets MNIST and Cifar-10 and reached a performance of TOP/s.

We should notice that since the bit-width of data has been reduced by 32 times in BNN, an execution speed of TOP/s is expected since many recent non-BNN designs have already reached a performance of several hundred GOP/s. The key optimizations include: (1) single-bit based MAC operation, which can be replaced by efficient XNOR and popcount operations and can be free from conventional multiply and add operations; (2) small size for both parameters and intermediate results, which would enable on-chip caching; (3) broaden bandwidth for on-chip BRAMs, which would reduce the bottleneck of data dependency with wide data-access patterns such as those in FC layers. Our FP-BNN design is developed based on the above motivations. Furthermore, FP-BNN supports large models such as XNOR-Net version AlexNet.

3. Resource-Aware Model Analysis (RAMA)

To design an NN accelerator on chip, we should consider how to tile the overall task onto limited resources, which can be classified into two classes: arithmetic units and memory units. To help choosing the size of task tiles, we need to estimate the resource cost beforehand. The RAMA method is introduced to address this need.

In modern FPGA platforms, four kinds of resources are provided: look-up tables (LUTs), flip-flops (FFs), block RAMs (BRAMs) and digital signal processing units (DSPs). LUTs and DSPs are the key to form arithmetic and control logic, while BRAMs are usually used as on-chip storage for fast data access. From the arithmetic perspective, MACs are the key operations which cost most resources. DSPs have hard-wired multipliers and can be configured to quickly deliver results under high clock frequency – and one can choose LUTs to implement a customized multiplier. We compare the resource cost of these two ways on a Stratix V FPGA synthesized with Altera Quartus v13.1, and the result is shown in Table 3.

With the resource cost of one single MAC operation in hand, we need to further count the number of MACs in each layer, which can be represented as $\mathcal{N}_{\text{layer}}(MAC)$. For CONV layers we have (FC layers can be seen as $K = R_{\text{out}} = C_{\text{out}} = 1$):

$$\mathcal{N}_{CONV/FC}(MAC) = N_{\text{in}} \times N_{\text{out}} \times K^2 \times R_{\text{out}} \times C_{\text{out}} \tag{11}$$

Table 3
Resource cost of MACs on Stratix V FPGA.

Operation	LUT	FF	DSP
32-bit float add(+)	581	525	0
x-bit fixed add(+)	x	x+1	0
32-bit float mult(×)	147	363	1
x-bit fixed mult(×)	0	1	$\lceil \frac{x}{18} \rceil$

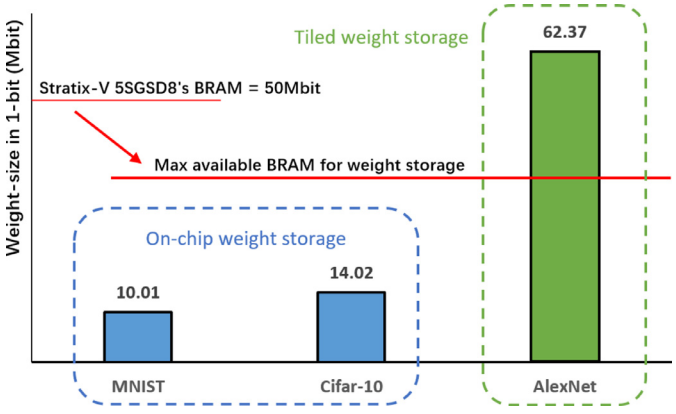


Fig. 2. Weight storage strategy selection for small (MNIST & Cifar-10) and large (AlexNet) models.

For operations in BN layers, the number of operations (NOP) has a linear relationship with the number of output channels N_{out}. Notice that the shift-based transformation can change multiplications into cheap sum and shift operations. To get NOP after tiling, we just need to replace the original dimensions with tiled ones, and then we can estimate the resource cost for a certain type $C_{\text{res\_type}}(layer)$ by summing up the product of tiled NOP and resource cost of one operation, which in tern help us determine the tiling factor.

Next, from the memory perspective, we should concentrate on the size of parameters and the activation outputs of each layer. Given that $\mathcal{N}_{\text{layer}}(data)$ denotes the size of a certain kind of data in one layer, then for the weights we have

$$\mathcal{N}_{CONV}(W) = N_{\text{in}} \times N_{\text{out}} \times K^2 \tag{12}$$

For other parameters, such as biases, normalization parameters, they are given by the number of output channels, that is

$$\mathcal{N}_{CONV}(Other) = N_{\text{out}} \tag{13}$$

For activations, we have

$$\mathcal{N}_{CONV}(A) = N_{\text{out}} \times R_{\text{out}}^2 \tag{14}$$

The overall memory cost of each type of data is the product of the bit-width and the amount of data. For weights, from Fig. 2 we can see that in an ideal binarized condition, small models can completely be stored in on-chip BRAMs, while large models' amount of weight can exceed the upper limit of available BRAMs. We use a tiled weight storage strategy that takes only one portion of weights required for the current tile from off-chip memories. For activations (feature maps (fmaps)), since data adjacency will be needed in both vertical and horizontal axis, BRAM will not be a suitable choice since it can only be configured into fixed shapes, and the maximum width of one BRAM is often no more than 40 and accordingly the minimum depth is 512.

4. Hardware logic design

In this section, we present the hardware logic design of our FPGA accelerator system.

Please cite this article as: S. Liang et al., FP-BNN: Binarized neural network on FPGA, Neurocomputing (2017), https://doi.org/10.1016/j.neucom.2017.09.046

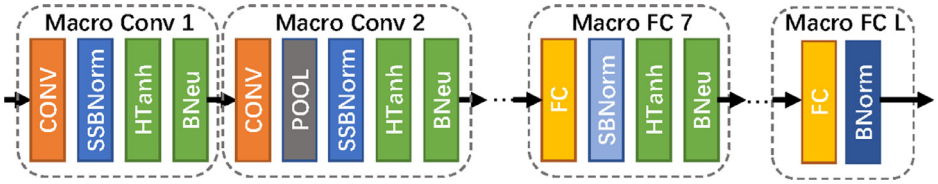


Fig. 3. A normal structure of a BNN model.

Table 4
RAMA-based topology analysis of MNIST MLP, Cifar-10 CONV-Net and AlexNet.

	Macro Layer	Structure[a]	$N_{in} \times R_{in}^2$	K	S	K_{POOL}[b]	$N_{out} \times R_{out}^2$	$\mathcal{N}(W)$	$\mathcal{N}(Others)$	$\mathcal{N}(MAC)$	$\mathcal{N}(A)$
MNIST MLP	1	F-B-A	784	–	–	–	2048	1.61 *M*	10240	1.61 *M*	2048
	2	F-B-A	2048	–	–	–	2048	4.19 *M*	10240	4.19 *M*	2048
	3	F-B-A	2048	–	–	–	2048	4.19 *M*	10240	4.19 *M*	2048
	4	F-B	2048	–	–	–	10	2048	50	20.48 *K*	10
	Total							**10.01 M**	**30.77 K**	**10.01 M**	–
CIFAR10 ConvNet	1	C-B-A	3×32^2	3	1	–	128×32^2	3456	640	3.54 *M*	131.07 *K*
	2	C-P-B-A	128×32^2	3	1	2	128×16^2	147.46 *K*	640	150.99 *M*	32.77 *K*
	3	C-B-A	128×16^2	3	1	–	256×16^2	294.91 *K*	1280	75.50 *M*	65.54 *K*
	4	C-P-B-A	256×16^2	3	1	2	256×8^2	589.82 *K*	1280	150.99 *M*	16.38 *K*
	5	C-B-A	256×8^2	3	1	–	512×8^2	1.18 *M*	2560	75.50 *M*	32.77 *K*
	6	C-P-B-A	512×8^2	3	1	2	512×4^2	2.36 *M*	2560	150.99 *M*	8192
	7	F-B-A	8192	–	–	–	1024	8.39 *M*	5120	8.39 *M*	1024
	8	F-B-A	1024	–	–	–	1024	1.05 *M*	5120	1.05 *M*	1024
	9	F-B	1024	–	–	–	10	10.24 *K*	50	10.24 *K*	10
	Total							**14.02 M**	**19.25 K**	**61.69 M**	–
AlexNet ConvNet	1	C-P-B-A	3×224^2	11	4	3	96×27^2	34.85 *K*	480	105.42 *M*	69.98 *K*
	2	C-P-B-A	96×27^2	5	1	3	256×13^2	614.40 *K*	1280	447.90 *M*	43.26 *K*
	3	C-B-A	256×13^2	3	1	–	384×13^2	884.74 *K*	1920	149.52 *M*	64.90 *K*
	4	C-B-A	384×13^2	3	1	–	384×13^2	1.33 *M*	1920	224.28 *M*	64.90 *K*
	5	C-P-B-A	384×13^2	3	1	3	256×6^2	884.74 *K*	1280	149.52 *M*	9216
	6	F-B-A	9216	–	–	–	4096	37.75 *M*	20480	37.75 *M*	4096
	7	F-B-A	8192	–	–	–	4096	16.78 *M*	20480	16.78 *M*	4096
	8	F-B	4096	–	–	–	1000	4.10 *M*	5000	4.10 *M*	1000
	Total							**62.37 M**	**52.84 K**	**1.14 G**	–

[a] F = FC, C = CONV, P = POOL, B = BN, A = Activation (HTanh+BNeu).
[b] All pooling layers' stride is 2.

4.1. Overall architecture

A normal structure of a BNN model is given in Fig. 3. We can divide the model into several macro-layers with similar structures, each including a convolution or fully-connected (*C/F*) layer, a batch normalization (BN) layer and an activation layer which consists of a Hard Tanh (HTanh) layer and a Binarized Neuron (BNeu) layer. For some macro layers, pooling is introduced for down-sampling. Here we choose MNIST MLP and Cifar-10 ConvNet as small dataset examples, and AlexNet for large dataset ImageNet. The topology of each model is described in Table 4, and the key features of each layer are extracted based on RAMA, in which R_{in} and R_{out} are respectively the input and output image size, K is the convolution kernel (window) size, S is the stride of the moving window, and K_{POOL} is the pooling window size.

The overall system is shown in Fig. 4. We have altogether N_{PE} channels to process in parallel the data from the input cache. CONV/FC (*C/F*) layer includes processing elements (PEs) that are shared by the CONV and FC since they both mainly consist of MAC computations. Shift-Based Normalization (SBN) layer adopts shift operations to replace multiplications as mentioned in Section 2.3. Activation layer merges the HTanh and BNeu layers together to produce an output vector containing either 0 or 1. Parameters for each layer are fetched from on-chip BRAMs or registers to meet bandwidth requirements, and control signals select them for each iteration. The output for each iteration will be transferred to the intermediate result cache. For each next layer, the interconnection will be reconfigured by the controller according to the type (CONV or FC) of the layer.

Next, we take a look at the details of different types of PE design.

4.2. C/F PE

4.2.1. XNOR-based Binary MAC

Normally, it is necessary to utilize DSPs or customized LUT-based logic to complete a MAC operation for both floating-point or fixed-point input values. However, if input values become binary, it will be much different.

Consider two input vectors $\mathbf{A} = \{a_i\}$ and $\mathbf{B} = \{b_i\}$ $(i = 1\ to\ N_{in})$ which consist of binarized values either +1 or −1, then the product of the corresponding elements in two vectors will also be either +1 or −1. The sign of the product depends on the two input elements' signs - if they are identical, then the product will be positive, otherwise it will be negative. Then, we need to accumulate these binary values to get a final result. This process is depicted in Fig. 5(a).

Hardware implementations usually take 2 bits to represent +1 and −1. If we use only one bit, we should take 0 and 1 as the basic values. This can be achieved through *affine transformation*.

Since we have

$$\mathbf{A}_{\langle 0,1\rangle} = \frac{\mathbf{A}_{\langle -1,1\rangle} + \mathbf{A}_{\langle 1\rangle}}{2} \tag{15}$$

in which $\mathbf{A}_{\langle 1\rangle}$ represents the all-1 vector of the same length of $\mathbf{A}_{\langle -1,1\rangle}$. To keep the truth table for the result as shown in Table 5, we can infer that the operation should be transformed from multiplication to *XNOR*. In addition, if we assume r to be the dot product

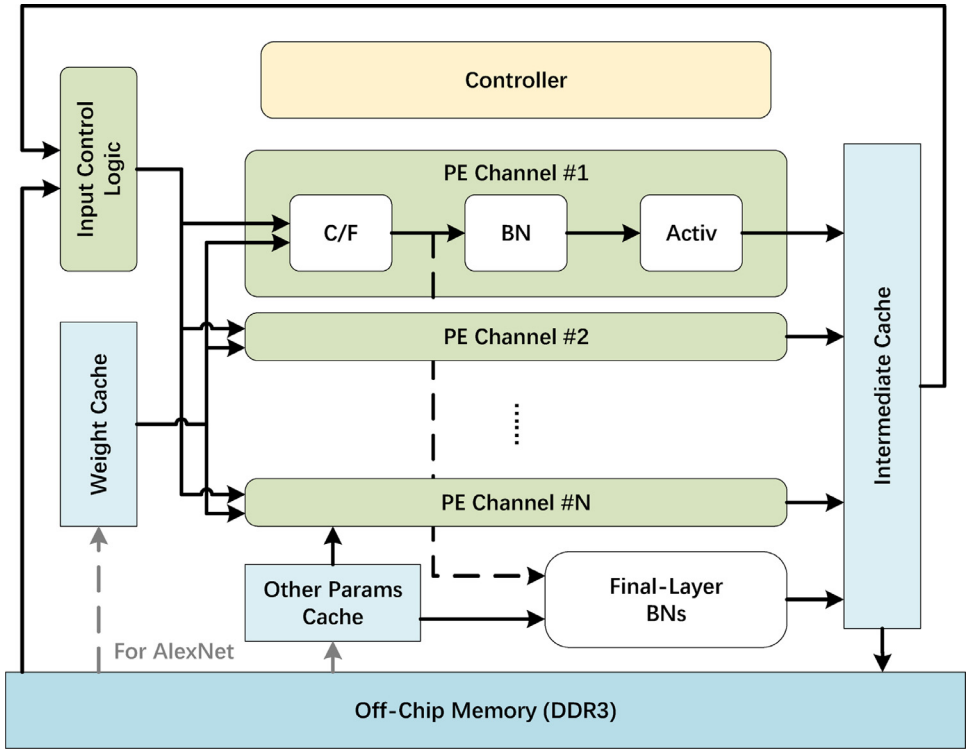


Fig. 4. The overall system architecture design.

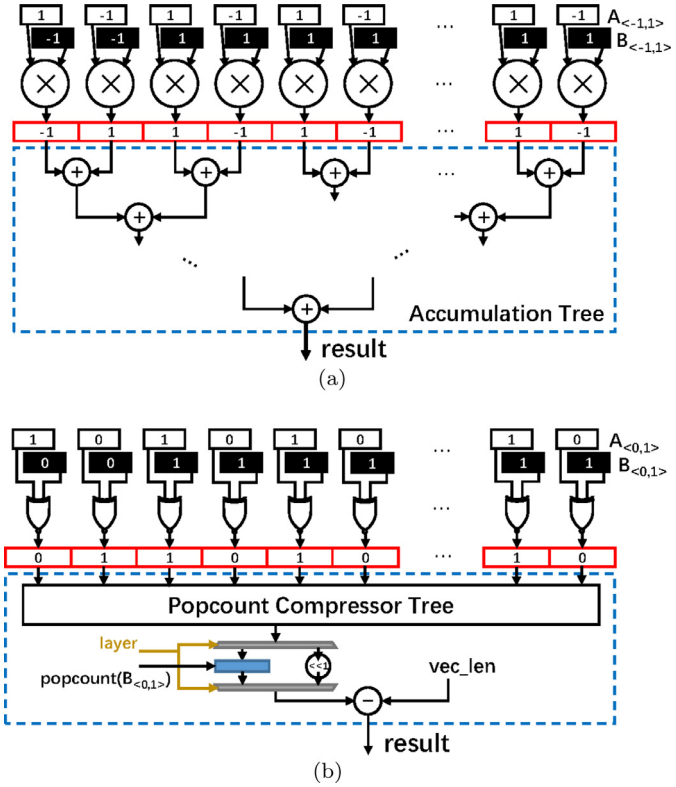


Fig. 5. Conversion from (a) ⟨−1, 1⟩-based MAC to (b) ⟨0, 1⟩-based XNOR and popcount operations.

Table 5
Truth table of affine transformed inputs and result.

Original multiplication			Affine transformed		
$a_{\langle -1,1\rangle}$	$b_{\langle -1,1\rangle}$	$a \cdot b_{\langle -1,1\rangle}$	$a_{\langle 0,1\rangle}$	$b_{\langle 0,1\rangle}$	$a \cdot b_{\langle 0,1\rangle}$
1	1	1	1	1	1
1	−1	−1	1	0	0
−1	1	−1	0	1	0
−1	−1	1	0	0	1

of vector $\mathbf{A}_{\langle -1,1\rangle}$ and $\mathbf{B}_{\langle -1,1\rangle}$ of length *vec_len*, then we will have

$$\begin{aligned} result &= \mathbf{A}_{\langle -1,1\rangle} \cdot \mathbf{B}_{\langle -1,1\rangle} = \sum_{i=1}^{vec\_len} a_{i\langle -1,1\rangle} \cdot b_{i\langle -1,1\rangle} \\ &= \sum_{i=1}^{vec\_len} \left[(a_{i\langle -1,1\rangle} \cdot b_{i\langle -1,1\rangle}) - (-a_{i\langle -1,1\rangle} \cdot b_{i\langle -1,1\rangle}) \right] \\ &= \sum_{i=1}^{vec\_len} \left[XNOR(a_{i\langle 0,1\rangle}, b_{i\langle 0,1\rangle}) - XOR(a_{i\langle 0,1\rangle}, b_{i\langle 0,1\rangle}) \right] \\ &= 2\,popcount(\mathbf{R}_{\langle 0,1\rangle}) - vec\_len \end{aligned} \tag{16}$$

in which

$$\mathbf{R}_{\langle 0,1\rangle} = \{XNOR(a_{i\langle 0,1\rangle}, b_{i\langle 0,1\rangle}), i = 1\ to\ vec\_len\} \tag{17}$$

If one of the inputs is already ⟨0, 1⟩ based, for example, the first layer, then we get the result with:

$$\begin{aligned} result &= \mathbf{A}_{\langle 0,1\rangle} \cdot \mathbf{B}_{\langle -1,1\rangle} = \frac{\mathbf{A}_{\langle -1,1\rangle} + \mathbf{A}_{\langle 1\rangle}}{2} \cdot \mathbf{B}_{\langle -1,1\rangle} \\ &= \frac{2\,popcount(\mathbf{R}_{\langle 0,1\rangle}) - vec\_len}{2} + \frac{\sum b_{i\langle -1,1\rangle}}{2} \\ &= popcount(\mathbf{R}_{\langle 0,1\rangle}) - vec\_len + \sum b_{i\langle 0,1\rangle} \end{aligned} \tag{18}$$

This means we need to add the popcount of vector $\mathbf{B}_{\langle 0,1\rangle}$ instead of left-shifting 1-bit, as shown in Fig. 5(b). The layer control

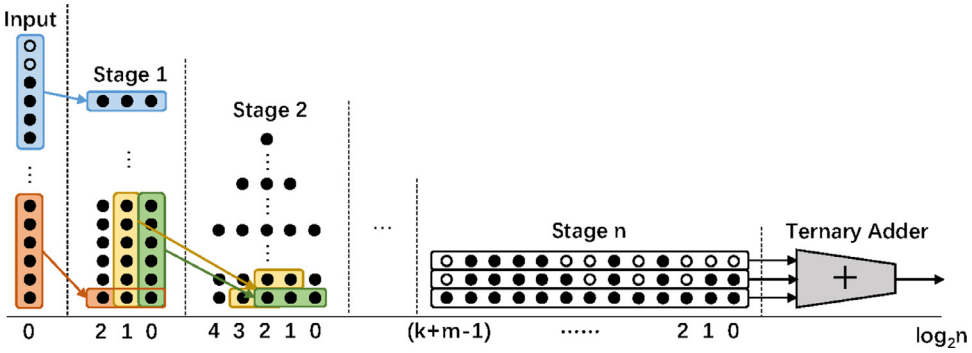


Fig. 6. The popcount compressor tree based on 6:3 compressors and one ternary adder.

signal selects the operation to the output of the popcount compressor tree.

4.2.2. Popcount Compressor (PC) tree

The popcount value, also known as Hamming Weight, can easily be calculated in parallel hardware. However, for long vectors, this process can be demanding both in time and in resource usage. The most common way is to use a binary full adder tree to sum up the bits in vectors, which will result in a delay of $log_2(vec\_len)$ and $n-1$ adders of different bitwidth. Here we present a compressor tree method inspired by [42].

The popcount process can be seen as compressing N input bits into $\lfloor \log_2 N \rfloor + 1$ result bits with weights. Since most modern FPGA architectures have 6-input LUTs, a 6:3 compressor (can be seen as $N = 6$) is therefore an efficient basic component, for it can calculate the popcount of a 6-bit input vector in a look-up table, which leads to a 3-bit popcount output with only three 6-input LUTs in parallel.

Given that tuple $\mathcal{T} = (p_k; q_{k+m-1}, \ldots, q_{k+1}, q_k)$ represents a p_k: m compressor, where the subscript $j = k, k+1, k+2\ldots$ stand for the bit weight 2^j and p_j, q_j stand for the input and output bit number of certain weight, respectively. In this way, a 6:3 compressor can be represented as (6; 1, 1, 1).

As shown in Fig. 6, the input vector is divided into 6-bit portions, each connected to the input ports of a 6–3 compressor. Empty input bits will be filled with dummy 0's (shown as hollow dots in Fig. 6). For the following stages, the bits with the same weight (we call it column vector) will repeat the same process, forming a compressor tree. The output bits will heap due to their weights. Our target is to reduce the height of the heap to 3, which can then be accepted as three input vectors of a ternary adder to get the final sum. Thus, the column vectors with height of less than 4 will stop being compressed for the next stage, and the compression process will terminate when all column vectors' heights are less than 4. The overall process of generating a compressor tree is described in Algorithm 2.

With the help of Algorithm 2, we obtain the compressor tree topology for the hardware implementations of popcount functions for different sizes of binary vectors. Table 6 gives the comparison between accumulation adder tree and compressor tree. As we can see, for long vectors, compressor tree saves around one third of LUT resources.

4.2.3. PE reuse

With the XNOR array connected to a popcount compressor tree, we can get the result for 0/1 input arrays. For all intermediate layers, the inputs (activations from the preceding layer) and weights must be binarized (either 0 or 1). However, this is not the case for the first layer – we usually take a fixed point input image from the input cache. Also for some large models like AlexNet some layers' weights are not binarized. To deal with this, if a vector $\mathbf{x}$ of n fixed-point inputs with m-bit precision:

$$\mathbf{x} = (\overline{x_{n-1}^{m-1} x_{n-1}^{m-2} \ldots x_{n-1}^{0}}, \overline{x_{n-2}^{m-1} x_{n-2}^{m-2} \ldots x_{n-2}^{0}}, \ldots, \overline{x_{0}^{m-1} x_{0}^{m-2} \ldots x_{0}^{0}}) \tag{19}$$

and a vector $\mathbf{w}$ of n p-bit weights:

$$\mathbf{w} = (\overline{w_{n-1}^{p-1} w_{n-1}^{p-2} \ldots w_{n-1}^{0}}, \overline{w_{n-2}^{p-1} w_{n-2}^{p-2} \ldots w_{n-2}^{0}}, \ldots, \overline{w_{0}^{p-1} w_{0}^{p-2} \ldots w_{0}^{0}}) \tag{20}$$

then the output vector $\mathbf{s}$ could be calculated by

$$\mathbf{s} = \mathbf{x} \cdot \mathbf{w} = \sum_{i=1}^{p} 2^{i-1} \sum_{j=1}^{m} 2^{j-1} \sum_{k=1}^{n} (x_{k-1}^{j-1} \cdot w_{k-1}^{i-1}) \tag{21}$$

Algorithm 2 Popcount compressor tree generation algorithm.

Require: Input vector: **i** of height N
Ensure: Updated: Column vector height $h(i, j)$, i stands for the weight of 2^i and j for the compression stage; Heap of stage j: $\mathcal{H}(j) = \{h(k, j)\}, k = 0, 1, \ldots, log_2(N)$.
$h(0, 0) = N, i = 0, j = 0$;
while $max(\mathcal{H}(j)) > 3$ **do**
  $\mathcal{H}(j+1) = zeros(1, log_2(N))$;
  for $k = 1$ to $log_2(N)$ **do**
    if $h(k, j) > 3$ **then**
      $n_{compressor}(k, j) = \lceil h(k, j)/6 \rceil$;
      $h(k, j+1) = h(k, j+1) + n_{compressor}(k, j)$;
      $h(k+1, j+1) = h(k+1, j+1) + n_{compressor}(k, j)$;
      $h(k+2, j+1) = h(k+2, j+1) + n_{compressor}(k, j)$;
    else
      $h(k, j+1) = h(k, j+1) + h(k, j)$;
    end if
  end for
  $j = j + 1$;
end while

Table 6
Comparison between accumulation adder tree and popcount compressor tree.

BW_{in} (bits)	BW_{out} (bits)	LUTs		
		Acc.	Pop.	Saved (%)
9 (3^2)	4	9	10	−11.1
16	5	21	19	9.52
64	7	98	79	19.39
256	9	398	291	26.88
1024	11	1596	1106	30.70
1152 (128×3^2)	11	1796	1228	31.63
1200 (48×5^2)	11	1864	1282	31.22
8192	14	12768	8362	34.51

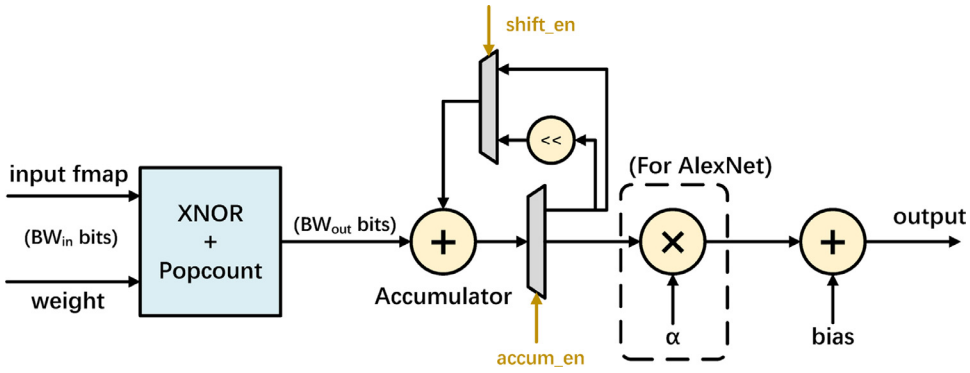


Fig. 7. The *C/F* layer PE module.

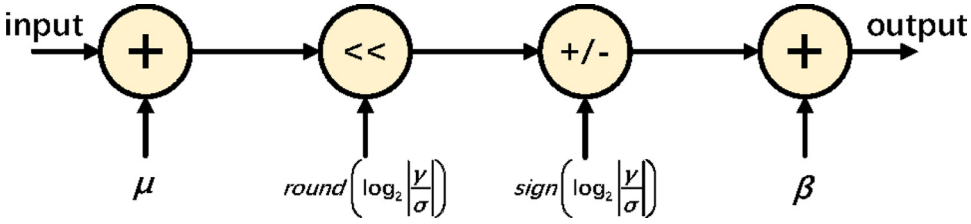


Fig. 8. The SBN layer PE module (MNIST and Cifar-10).

Implementing Eq. (21) requires reuse of PE. Hence, we introduce an accumulator to the PE with a selectable left-shifter. While the precedent lower bit vector is being processed, the next input vector can be loaded behind, and be added with the shifted result of the precedent vector. The start and the end of the accumulation will be set by the controller signal. A detailed scheduling will be introduced in Section 5.

The overall structure of C/F PE is given in Fig. 7. For AlexNet, the binarization method is different from sign function[20]. It introduces a binarized filter $\mathbf{w}_{bin}$ for $\mathbf{w}$ with a scaling factor α in order to approximate the MAC operation by $\mathbf{x}\cdot\mathbf{w}\approx\alpha(\mathbf{x}\cdot\mathbf{w}_{bin})$, where $\alpha=\frac{\mathbf{w}^T\mathbf{w}_{bin}}{n}=\frac{1}{n}\|\mathbf{w}\|_{\ell 1}$. So for AlexNet the accumulator is followed by a multiplier to time the scaling factor α.

4.3. BN PE

As described in Algorithm 1[23], the batch normalization process can be presented as:

$$y=\frac{x-\mu}{\sigma}\cdot\gamma+\beta, \tag{22}$$

where μ stands for the running mean value, σ stands for the standard deviation. γ and β are the learnt values to implement affine scale and shift for an identity transform. However, as mentioned in Section 2.3, floating-point multiplications are required for every normalization process, which will lead to a considerable resource cost. For this reason, [18] uses shifting to approximate the multiply operation. For Eq. (22), the shift-based approximation would be:

$$y=sal[(x-\mu),\phi]\cdot sign\left|\frac{\gamma}{\sigma}\right|+\beta, \tag{23}$$

where $\phi=round(\log_2|\frac{\gamma}{\sigma}|)$ is the left-shift value of both σ and γ.

As we have the pre-trained models in hand, we can calculate the required parameters for BN like $\frac{\gamma}{\sigma}$ in advance, and store them into the corresponding parameter cache. With the above, we get the SBN PE as presented in Fig. 8. We have also noticed that the shifting-based approximation would cause severe accuracy drop for AlexNet (from 42.9% to 31.9%), so we avoid using shift replacement for AlexNet and keep the original batch normalization, using a multiplier to replace the shift and sign operations.

For all models, we train them with the last BN layer kept as the original batch normalization in order to avoid accuracy loss, that is, with no shift-based operations. Floating-point multiplications are implemented independently with DSP blocks to accomplish the multiplication operations, and for ImageNet classification (AlexNet), the output process will be tiled.

4.4. Activation PE

For the last part of a normal macro layer, we need to binarize the result into either 0 or 1. In the training process, the HTanh function, as shown in Table 2, constricts the values between −1 and 1, while the final BNeu layer will push those values in between to the two boundaries, which means −1 for all the negatives and 1 for the others. Since we introduce in Section 4.1 that the (−1,1) based vectors can be affine transformed into (0,1) vectors, the case becomes much simpler: we just need to discern all the negative values from the SBN layer and set them to 1, with the others to 0. This can be done directly by accessing the signal bit of these values.

4.5. Pooling

For some macro-layers, pooling is applied to support subsampling in order to reduce the output fmap size. As shown in Table 4, pooling comes closely after CONV layer, and the pooling type for all of our models is max-pooling. If pooling comes after activation, most of the output values will be +1 which result in significant information loss for training [20]. So a C-P-B-A macro-layer structure is taken. However, for the inference process, a C-B-A-P structure can get an identical result and the pooling is applied to values of 0 and 1 only. This can be directly implemented with *OR* operations. We organize selective line buffers after the activation PEs, and when a pooling process is required, the activation values will stream into the line buffers. If the pooling size is *K*, we will enable *OR* operations to the horizontal targeted locations once *K* rows of activations arrive, and then a similar process is repeated along the vertical direction with other line buffers to complete a 2D max-pooling.

Please cite this article as: S. Liang et al., FP-BNN: Binarized neural network on FPGA, Neurocomputing (2017), https://doi.org/10.1016/j.neucom.2017.09.046

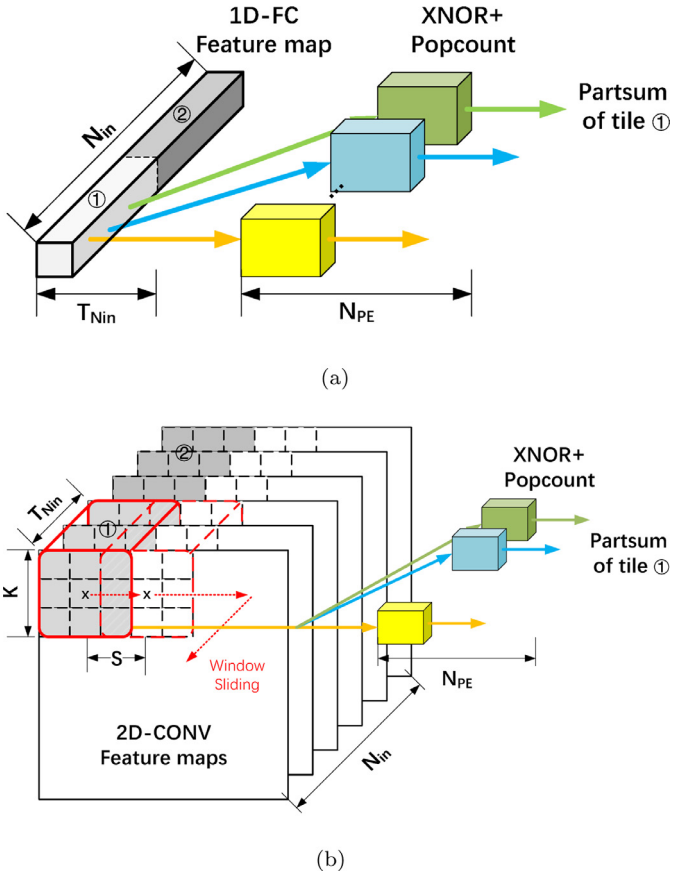


Fig. 9. Task scheduling for C/F layer: (a)FC; (b)CONV.

5. Task tiling and scheduling (T&S)

This section introduces T&S, a method for tiling and scheduling models on chip. The T&S method is depicted in Fig. 9. We assume the width of one PE to be PE_{size}, and the number of tiled input channels to be $T_{N_{in}}$. The number of tiled output channels equals to the number of PE channels N_{PE}.

For FC layers, one tiled input will be fed into all PE channels. For better resource utilization we set $T_{N_{\text{in}}}$ as close as possible to PE_{size}. It will take $\left\lceil \frac{N_{\text{in}}}{T_{N_{\text{in}}}} \right\rceil$ iterations to get the intermediate result accumulated for one output. This will be repeated by $\left\lceil \frac{N_{\text{out}}}{N_{PE}} \right\rceil$ times for all outputs get done.

For CONV layers, since most filter kernel size K is rather small, we consider joining several filters together as one input for *C/F* PE. For one PE, the input will be summed up to get one output, so the joined filters should be at the same location of input fmaps as different locations have no dependency to each other. The input vector size will be $L_{in} = T_{N_{\text{in}}} \times K^2$. Similar to the FC layers, $\left\lceil \frac{N_{\text{in}}}{T_{N_{\text{in}}}} \right\rceil$ will be taken to get one output pixel, and the next tiled input location will be given in a sliding window style.

Considering the datapath shared among different layers, N_{PE} should be a common divisor of N_{out} of different layers, $T_{N_{\text{in}}}$ for each layer should preferably be best a sub-multiple of N_{in}, and PE_{size} should be a big value and also close to L_{in} to best explore the resource utilization.

The first layer can become a bottleneck for the datapath since the number of N_{in} is small (usually 3). Let us study Eq. (21), for input values which are not binarized, $T_{N_{\text{in}}}$ can get multiplied if tiling could be achieved inside Eq. (21). Notice that the inner most MAC will be implemented through XNOR + Popcount, and the m-bit of input is actually calculated in different run and accumulated with shifting. As the PE_{size} is much larger than $L_{\text{in}} = N_{\text{in}} \times K^2$ in the first layer, we can repeat the innermost MAC by 2^j times inside one PE to complete the j iterations in one run. This process is illustrated in Fig. 10. If t-bit is tiled in one run, then we have

$$L_{\text{in}} = N_{\text{in}} \cdot K^2 \cdot \sum_{i=1}^{t} 2^{i-1} \tag{24}$$

and through this tiling, the utilization rate of PE for the first layer is increased.

With RAMA and the information given in Tables 4 and6, our tiling strategy is shown in Table 7.

Table 7
Tiling strategy for different models.

Model	N_{PE}	PE_{size}	L_{in} of layer								
			1	2	3	4	5	6	7	8	9
MNIST	64	1024	784		1024						
Cifar-10		1152	405			1152				1024	
AlexNet		1200	1089	1200		1152			1024		

6. Memory system design

In this section, we introduce the memory system design for FP-BNN. This mainly consists of two parts: the first is parameter quantization and storage, and the second is on-chip fmap caching.

6.1. Quantization over other parameters

Since in BNN models, weights have already been binarized, so to take a further step, we quantize non-weight parameters which we call as *other parameters*. It would be essential for small models since we would like to store everything on-chip. These parameters are in floating-point format, and their number mostly is equal to the number of output channels of a layer. BRAMs would be too wasteful for their storage, since these parameters need to be provided in parallel and the width equals to N_{PE}, which would make the depth of their storage too shallow. So we place them in fast distributed memories. Such memories are available in Altera FPGAs as memory logic array blocks (MLABs). Since we also need to use MLABs to construct intermediate cache, quantization of other parameters is adopted to make the best use of limited MLAB storage.

A fixed-point number can be represented as:

$$n = \sum_{i=0}^{BW-1} N_i \cdot 2^{i-f_l}, \tag{25}$$

where BW stands for the overall bitwidth (including the sign bit) of the number and f_l stands for the fractional part bitwidth. Here we use $\mathcal{Q} = (BW, -f_l)$ to capture the quantization strategy for a particular type of parameters in a layer. To transform a floating-point number n_{float} to a fixed-point number n_{fixed} using a given $\mathcal{Q}$ strategy, we make use of the round-to-nearest rounding mode[43] to shift and cut:

$$n_{\text{fixed}} = sal\{round[sal(n_{\text{float}}, f_l)], -f_l\} \tag{26}$$

We use this method to quantize parameters for various models to see how the accuracy fluctuates with the bitwidth variation (Fig. 11). It is obvious that when the bitwidth of parameters drops below a particular threshold, the model accuracy drops significantly.

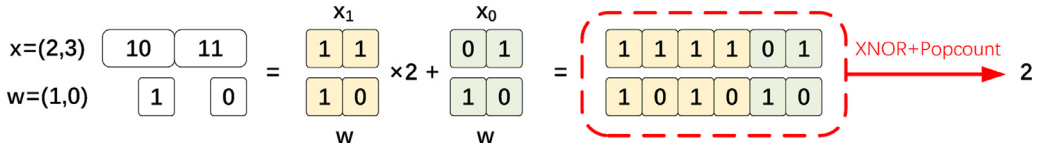


Fig. 10. Example of tiling for multiple bit case (2-bit input and 1-bit weight).

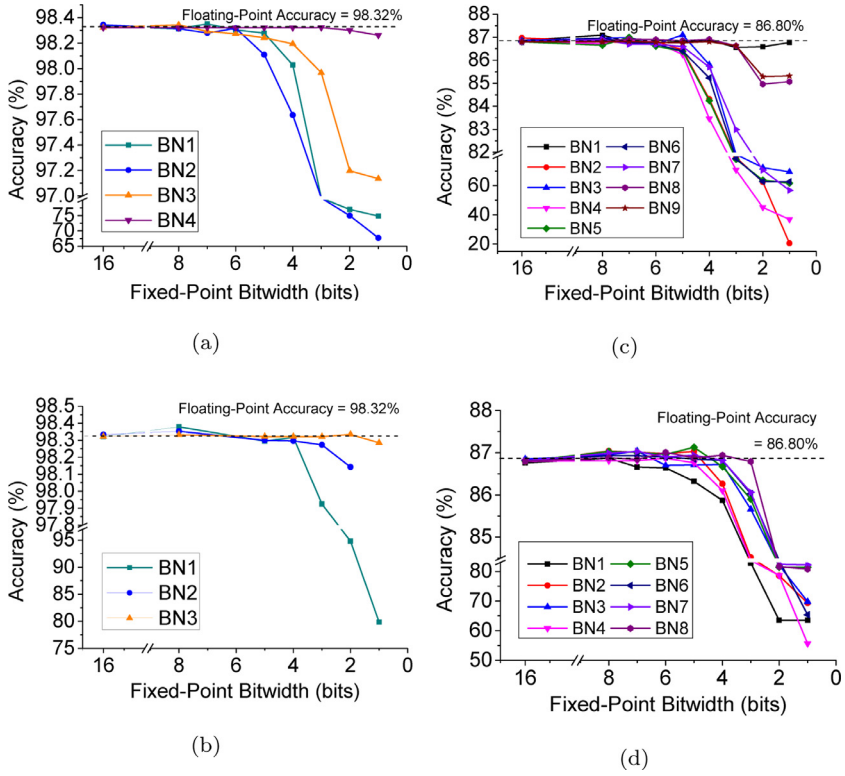


Fig. 11. The model accuracy variation as a function of bitwidth of BN's mean μ ((a)MNIST; (c)Cifar-10) and affine bias β ((b)MNIST; (d)Cifar-10), respectively.

For the biases of *C/F* layers, we discover that even if their bitwidth drops to 0, there is still no significant variance of the results. So, to save storage and computing resources, we ignore the *C/F* layer bias addition. For the running means μ and the affine biases β, the point varies between 2 to 8-bit for different layers. For the shift parameter of BN layers, we can express each ϕ as $\phi_{\min} + \Delta\phi$, and therefore we only need to store the variance using $\Delta\phi$ to reduce resource usage. So the bitwidth w of *BW* can be calculated as:

$$w = \begin{cases} \lfloor \log_2(\phi_{\max} - \phi_{\min}) \rfloor + 1, & \phi_{\max} > \phi_{\min} \\ 0 & \phi_{\max} = \phi_{\min} \end{cases} \quad (27)$$

We choose a dynamic fixed-point strategy [44] to optimize the bitwidth for each layer. Table 8 shows all the value ranges and *BW* strategies that we have chosen for each parameter, and Table 9 shows the comparisons between the quantized models and the original models.

6.2. Memories for parameters

The memory storage structure for weights is given in Fig. 12. In order to reduce the memory access time, we keep the parallelism of memories identical to the number of PE channels N_{PE}, and the width of each memory equals to PE_{size}. Weights for each computing tile, which is represented in form, will be arranged serially. An address generator will be controlled by the overall controller in order to provide the exact weight. For MNIST MLP and Cifar-10 ConvNet, the weights can fit into an array of BRAMs. For AlexNet, the weights will be tiled by each layer or inside a layer once they get too large. The oldest weights for finished tiles will be covered by weights for the next tile from off-chip memory in a ping-pong fashion. For other parameters, a similar storage structure is proposed based on MLABs.

6.3. Intermediate cache

For the intermediate outputs, that is, the output fmaps of each macro layer, we place a cache to hold them for the next macro layer to read. The design of intermediate cache is given in Fig. 13. For CONV layers, the cache structure facilitates the sliding window data fetching. For each input iteration, we separate the intermediate cache into $T_{N_{\text{in}}}$ groups, each containing K memory blocks, and every two adjacent blocks storing consecutive rows. The input logic needs to offer corresponding addresses for the required K rows, and select rows to choose the horizontal required K-bits. So in total we get $T_{N_{\text{in}}} \times K^2$ windows to form an L_{in} long tensor. The output fmaps of each layer will be stored into the spared space from the input fmaps in a ping-pong way. For FC layers, we just need to ensure that the cache bitwidth can satisfy $T_{N_{\text{in}}}$. For MNIST MLP we choose 32 32×32 MLABs for intermediate cache, and for Cifar-10 ConvNet and AlexNet, we take 384 32×32 MLABs.

Table 8
Quantization strategies for parameters other than *C/F* weights (MNIST & Cifar-10),

Model	Layer	FC/CONV			Batch normalization								
		Bias			Mean (μ)			Stdv & affine weight ($\phi = \log_2\lvert\frac{\gamma}{\sigma}\rvert$)			Affine bias (β)		
		Min	Max	$\mathcal{Q}$	Min	Max	$\mathcal{Q}$	Min	Max	$\mathcal{Q}$	Min	Max	$\mathcal{Q}$
MNIST	1	−24.3146	22.2333	(0,0)	−139.1505	148.0735	(6,3)	1	10	(4,0)	−3.0121	3.0051	(4,−1)
	2	−29.2907	34.2132	(0,0)	−156.4832	166.0365	(6,3)	0	10	(4,0)	−3.1457	3.1249	(4,−1)
	3	−26.6156	35.4796	(0,0)	−135.1257	139.6590	(7,2)	2	11	(4,0)	−2.6029	2.6408	(2,1)
	4	−0.1449	0.0467	(0,0)	−44.5449	39.5620	(2,5)						
Cifar-10	1	−0.9777	0.9976	(0,0)	−0.9788	0.9983	(4,−3)	1	2	(1,0)	−1	0.9998	(8,−6)
	2	−0.9990	0.9998	(0,0)	−42.0525	112.1753	(6,2)	6	7	(1,0)	−0.7739	0.5044	(5,−4)
	3	−0.9987	0.9998	(0,0)	−97.7165	76.5325	(5,3)	6	6	(0,0)	−0.9993	0.9991	(7,−6)
	4	−1	0.9923	(0,0)	−78.6930	190.5350	(6,3)	5	7	(2,0)	−0.9661	0.6624	(6,−5)
	5	−0.9968	0.9964	(0,0)	−155.4622	172.6664	(7,2)	6	7	(1,0)	−1	1	(5,−3)
	6	−0.9862	1	(0,0)	−29.3337	270.3043	(6,4)	5	8	(2,0)	−0.9983	0.9684	(5,−4)
	7	−1	1	(0,0)	−798.1793	761.8890	(7,4)	4	8	(3,0)	−1	1	(5,−3)
	8	−1	1	(0,0)	−131.6111	144.3177	(4,5)	−6	7	(4,0)	−1	1	(4,−2)
	9	−0.2266	0.1051	(0,0)	108.3246	203.2282	(4,5)						

Table 9
Model classification accuracy and size comparisons among original, binarized and quantized ones.

		Model Accuracy	Parameter Size	
			C/F Weights (*M* bit)	Others (*K* bit)
MNIST	Original	(98.7 ± 0.2)%	305.63	961.56
	BNN	98.32%	9.55	961.56
	Ours	98.24%	9.55	82.02
Cifar-10	Original	89.06%	427.92	601.56
	BNN	86.80%	13.37	601.56
	Ours	86.31%	13.37	49.66
AlexNet	Original	56.6%(top-1), 79.4%(top-5)	1903.31	1651.25
	XNOR-Net & Ours	42.90%(top-1), 66.80%(top-5)	87.05	1651.25

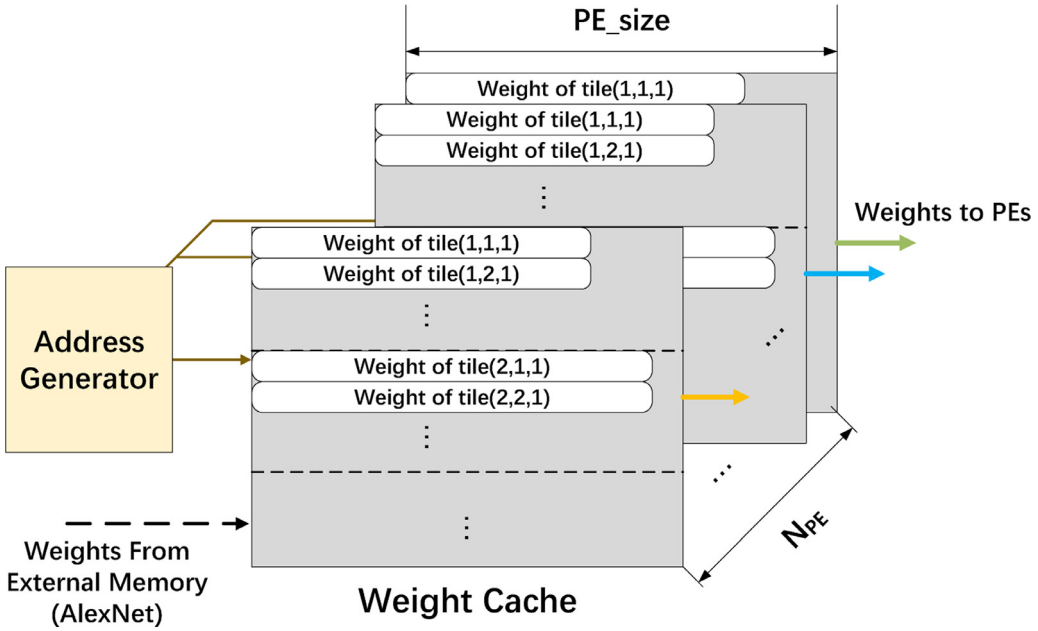


Fig. 12. Memory storage management pattern for weight cache.

7. System evaluation

We evaluate the performance of our accelerator system in this section. Environment setup and NN model preparation will be introduced first. We target CNN models to train for a binarized version, and apply quantization to the parameters to further compress the model. Then we map the optimized BNN models onto FPGA, and provide performance analysis comparing FP-BNN with general purpose processors and other FPGA designs.

7.1. System environment

We train the models on an IBM x3650 M4 server equipped with an NVIDIA Tesla K40 (28 nm feature size, 2880 CUDA cores with 12 GB GDDR5 external memory) and a K80 GPU (28 nm feature size, 4992 CUDA cores with 24 GB GDDR5 external memory) card, and use both cards to accelerate the training process. The evaluation system is built on Maxeler's MPC-X2000 platform [45]. The system has 8 dataflow engines (DFEs), each comprising a single Altera Stratix-V 5SGSD8 FPGA (28 nm feature size) connected

Please cite this article as: S. Liang et al., FP-BNN: Binarized neural network on FPGA, Neurocomputing (2017), https://doi.org/10.1016/j.neucom.2017.09.046

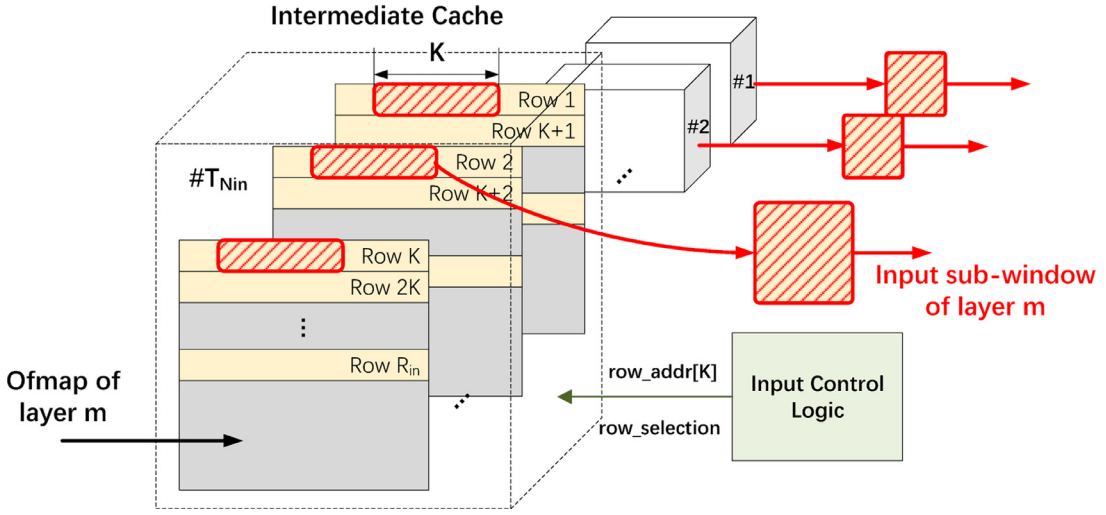


Fig. 13. The structure of intermediate cache.

to 48 GB of DDR3 RAM, and can communicate with other DFEs through MaxRing interconnections. Besides, two Intel Xeon E5-2640 6-core CPUs (32 nm feature size) are included in the server, and can communicate with the DFEs through InfiniBand. Here we take only one of the DFEs to implement the models. The Maxeler system offers a convenient solution to support data communication between software algorithms and FPGA hardware.

Table 10
FPGA Resource utilization of different models.

	ALM	DSP	BRAM
MNIST	182301(69.5%)	20(1.02%)	2210(86.09%)
Cifar-10	219010(83.5%)	20(1.02%)	2210(86.09%)
AlexNet	230918(88.0%)	384(19.6%)	2210(86.09%)
Available	262400	1963	2567

7.2. Model preparation

We use Torch 7 framework [46] to train the NN models for MNIST and Cifar-10 based on Hubara's BNN framework [18] and for AlexNet based on Rastegari's XNOR-Net framework [20]. The MNIST dataset is a permutation-invariant version consists of 60 K examples of 28×28 gray level digit images for training and 10 K examples for testing. The Cifar-10 dataset consists of 50 K examples of 32×32 RGB colour images in 10 classes for training and 10 K examples for testing, and global contrast normalization and ZCA whitening are used in the same way as Goodfellow et al. [47] and Lin et al.[48] did. The ImageNet dataset consists of 1.2 M images from 1 K categories and 50 K images for testing, and a center crop of 224×224 is extracted for forward propagation. Adam [49] learning rule is adopted for training, with a mini-batch size of 100, 200 and 800. The binarization method used here is deterministic [50] considering the convenience for implementing hardware for inference. Model accuracies are measured and presented in Table 9. As we can see, even the quantized versions for MNIST and Cifar-10 keep high accuracies close to the state-of-the-art results. XNOR-Net based AlexNet for our design suffers from a 13% accuracy drop, while supporting state-of-the-art performance among the existing BNN solutions for ImageNet.

7.3. Hardware implementation

We use MaxCompiler to generate the executable bit-stream for FPGA, which takes Altera Quartus II v13.1 to synthesize, place and route the designs. The resource utilization of the final implementation is shown in Table 10. The design can be driven with an achievable 150 MHz clock. We notice that the utilization of DSP blocks is not high, since only a small portion of arithmetic operations needs floating-point multipliers.

7.4. Performance Analysis

We implement binarized models for the Xeon E5-2640 CPU, the NVIDIA Tesla K40 GPU and the Altera Stratix-V FPGA. Performance is measured as shown in Table 11. To feed CPU and GPU with enough data, we take batch size to be identical to training for forward propagation. As we can see, with about an order of magnitude slower clock frequency and much lower power consumption, our accelerator still gets an average speed-up of 314.07 times over CPU and 19.08 times over GPU for MNIST, 51.83 times over CPU and 5.07 times over GPU for Cifar-10, and 11.67 times over CPU and 2.72 times over GPU for ImageNet. Peak speed-ups can reach 705.19 times over CPU and 70.75 times over GPU. Although the model has been compressed for about 32 times, the low-precision operations can exploit the potential of fine-grained parallelism in FPGA, which can offer higher performance than CPUs and GPUs. If we take energy efficiency as the criterion, with similar feature size, the FPGA implementation can offer an efficiency of two to three orders of magnitude of CPU's and GPU's.

We take another comparison with some previous FPGA accelerator designs for CNN and BNN models, as listed in Table 12. We can see that our FP-BNN reaches a TOP/s speed which is significantly faster than the previous CNN designs. For some designs (such as that in [15]), one major problem is that for memory-centric FC layers, the data and parameter loading time is much longer than the computing time as the number of input ports for data and weight RAMs is limited to 8, while in our design all the computing channels can be fed with data and weights in parallel. FP-BNN is also much faster than the most recent BNN design [40]. Although our design involves a large FPGA, the power efficiency is also 10 times better. Another BNN design FINN [41] reaches a performance similar to ours. For the MNIST case, FINN has taken a smaller MLP network in which the input dimension is larger than the number of neurons in each layer, which results in a higher resource utilization after task tiling. If we are prepared to reduce model accuracy

Please cite this article as: S. Liang et al., FP-BNN: Binarized neural network on FPGA, Neurocomputing (2017), https://doi.org/10.1016/j.neucom.2017.09.046

Table 11
Performance analysis among Xeon E5-2640 CPU, NVIDIA Tesla K40 GPU and Maxeler MAX4 (Stratix V) FPGA systems.

			CPU		NVIDIA Tesla K40 GPU		Maxeler MPC-X2000 with Stratix V FPGA			
Core clock (MHz)			2.5 K (Base) / 3 K (Boost)		745 (Base) / 810 & 875 (Boost)		150			
DDR memory			–		12 GB GDDR5 @3.0GHz		48 GB DDR3 @1.6 GHz			
Power			95 W		235 W(Board)		26.2 W(Board)			
Model	Macro layer	Ops	Time (ms)	Perf (GOP/s)	Time (ms)	Perf (GOP/s)	Time (ms)	Perf (GOP/s)	Speedup to CPU	Speedup to GPU
MNIST	1(FC)	3.21 *M*	17.54	18.32	1.08	298.63	1.97×10^{-3}	1633.89	89.19x	5.47x
	2(FC)	8.39 *M*	44.28	18.95	2.57	326.61	6.87×10^{-4}	12219.40	644.91x	37.41x
	3(FC)	8.39 *M*	43.98	19.08	2.57	326.61	6.87×10^{-4}	12219.40	640.51x	37.41x
	4(FC)	40.96 *K*	0.77	5.33	0.26	15.76	5.33×10^{-5}	768.19	144.17x	48.75x
	Total	20.04 *M*	106.57	18.80	6.47	309.48	3.39×10^{-3}	**5904.40**	**314.07x**	**19.08x**
Cifar-10	1(CONV)	7.08 *M*	19.70	71.85	4.38	323.20	4.10×10^{-2}	172.61	2.40x	0.53x
	2(CONV)	301.99 *M*	355.74	169.78	31.0	1947.69	2.74×10^{-2}	11040.33	65.03x	5.67x
	3(CONV)	151.00 *M*	140.85	214.40	14.7	2059.96	1.37×10^{-2}	11021.55	51.41x	5.35x
	4(CONV)	301.99 *M*	283.23	213.25	27.8	2170.09	2.05×10^{-2}	14712.09	68.99x	6.78x
	5(CONV)	151.00 *M*	146.28	206.44	16.2	1858.86	1.03×10^{-2}	14678.75	71.10x	7.90x
	6(CONV)	301.99 *M*	297.53	203.00	30.7	1965.06	1.71×10^{-2}	17646.50	86.93x	8.98x
	7(FC)	16.78 *M*	104.95	31.97	7.00	479.38	1.01×10^{-3}	16667.13	521.27x	34.77x
	8(FC)	2.10 *M*	12.35	33.98	0.92	455.14	2.60×10^{-4}	8069.91	237.48x	17.73x
	9(FC)	20.48 *K*	0.64	6.45	0.33	12.27	6.67×10^{-5}	307.35	47.62x	25.05x
	Total	1.23 *G*	1361.28	181.29	133	1853.87	1.3×10^{-1}	**9396.41**	**51.83x**	**5.07x**
AlexNet	1(CONV)[a]	211.83 *M*	790.69	213.31	345.68	487.91	9.98×10^{-1}	211.19	0.99x	0.43x
	2(CONV)	895.80 *M*	2125.08	337.23	186.57	3841.23	5.84×10^{-2}	15347.72	45.51x	4.00x
	3(CONV)	299.04 *M*	1715.52	139.45	127.28	1879.54	2.03×10^{-2}	14711.75	105.50x	7.83x
	4(CONV)	448.56 *M*	1460.71	245.67	165.95	2162.39	2.71×10^{-2}	16560.22	67.41x	7.66x
	5(CONV)	299.04 *M*	1346.86	177.62	108.86	2197.61	1.81×10^{-2}	16545.97	93.15x	7.53x
	6(FC)	75.50 *M*	1640.79	36.81	168.97	357.44	4.31×10^{-3}	17503.28	475.50x	48.97x
	7(FC)	33.55 *M*	1229.86	21.83	123.40	217.54	2.18×10^{-3}	15391.94	705.19x	70.75x
	8(FC)[a]	8.19 *M*	499.78	13.66	30.00	218.40	2.74×10^{-2}	298.47	21.85x	1.37x
	Total	2.27 *G*	10789.29	168.35	1256.72	722.68	1.16	**1963.96**	**11.67x**	**2.72x**

[a] The weights of these layers are quantized to 8-bit.

Table 12
Performance comparison with former FPGA-based CNN accelerator designs.

	FPGA'16 [51]	FPGA'16 [15]	FPL'16 [32]	FPGA'17 [40]	FPGA'17 [41]		This work		
Platform	Stratix-V 5SGSD8	Zynq XC7Z045	Virtex-7 VX690T	Zynq XC7Z020	Zynq XC7Z045		Stratix-V 5SGSD8		
Clock(MHz)	120	150	156	143	200		150		
Precision (bit)	8–16	16	16	Input: 8 weight: 1	Input: 8 weight: 1		Input: 8 weight: 1 (8 for the first and last layer of AlexNet) others: 2–8 (MNIST & Cifar-10), 32 (AlexNet)		
Model size (OPs)	30.9 G	30.76 G	1.45 G	1.24 G	MNIST	Cifar-10	MNIST	Cifar-10	AlexNet
					5.8 M	112.5 M	20.02 M	1.23 G	2.27G
Performance[a] (GOP/s)	117.8	136.97 (O) 187.80 (C) 1.20 (F)	565.94	207.8 (O) 318.9 (C)	9085.67	2465.5	5905.40 (O) 12219.40 (P)	9396.41 (O) 17646.50 (P)	1963.96 (O) 17503.28 (P)
Power (W)	25.8	9.63	30.2	4.7	22.6	11.7	26.2		
Efficiency (GOP/s/W)	4.57	14.22	22.15	44.2	402.02	210.72	225.36 (O) 466.39 (P)	358.64 (O) 673.53 (P)	74.96 (O) 668.06 (P)

[a] O = Overall, P = Peak, C = CONV, F = FC.

for a smaller network, the overall performance should get closer to the peak value (12 TOP/s). For the Cifar-10 case, our CONV-Net model can achieve a throughput of almost 4 times of FINN's. We also support large datasets for our FP-BNN design, which proves the compatibility of our design method with various CNN models.

7.5. Discussion

There is considerable scope for improvement in FP-BNN especially for the first layers, since datapath utilization is low due to the limited number of input channels. Moreover, the utilization of DSP blocks is low, and more DSP blocks can be involved if they can effectively support low-bandwidth operations to enhance the overall throughput. Furthermore, we can exploit the heterogeneity of logic elements in FPGAs, such as introducing different bit-width choices together with binarized data for better use of DSP multipliers.

This implementation shows that it is promising to implement BNN models especially for an embedded system, which can offer a competitive speed and accuracy with low power consumption. Recently, various designs [36,52] have shown that more complicated NN models can also be binarized with tolerable loss of accuracy. Considering the similarity of component layers and logic generation algorithms, it is feasible to implement these models layer-by-layer in a sequential way as long as there is sufficient amount of on-chip memory for parameters.

Please cite this article as: S. Liang et al., FP-BNN: Binarized neural network on FPGA, Neurocomputing (2017), https://doi.org/10.1016/j.neucom.2017.09.046

8. Conclusion

This paper presents FP-BNN – our design for binarized neural networks targeting FPGA technology. Based on the RAMA analysis method, we design a 64-channel accelerator architecture, which can accommodate both CONV and FC type layers. An XNOR-based method is introduced for binarized vector MAC operations, and the summing up process is achieved with a popcount compressor tree which can be automatically generated. For small models like MNIST MLP and Cifar-10 ConvNet, shift-based normalization is introduced which largely reduces the cost of multipliers. With proper dynamic quantization to the input and parameters, the model keeps good performance with the weights binarized and other parameters compressed by over 10 times. Optimized on-chip data storage is managed with parameter quantization. Our implementation on Maxeler MPC-X2000 platform (with Stratix-V 5SGSD8 FPGA) shows a promising TOP/s speed with only 26.2 W power at 150 MHz clock frequency. We expect expect enhanced accuracy in future binarized models, which should greatly extend their range of applications.

Acknowledgement

The support of Maxeler University Programme, Altera, Intel, UK EPSRC (EP/P010040/1, EP/L00058X/1, EP/L016796/1 and EP/N031768/1), the European Union Horizon 2020 Research and Innovation Programme under grant agreement number 671653, and the HiPEAC NoE is gratefully acknowledged.

References

[1] Y. LeCun, C. Cortes, C. J. Burges, The MNIST database of handwritten digits, 1998, http://yann.lecun.com/exdb/mnist/.
[2] A. Krizhevsky, V. Nair, G. Hinton, The CIFAR-10 dataset, 2014, https://www.cs.toronto.edu/~kriz/cifar.html.
[3] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, et al., Imagenet large scale visual recognition challenge, Int. J. Comput. Vis. 115 (3) (2015) 211–252.
[4] G. Hinton, L. Deng, D. Yu, G.E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T.N. Sainath, et al., Deep neural networks for acoustic modeling in speech recognition: the shared views of four research groups, IEEE Signal Process. Mag. 29 (6) (2012) 82–97.
[5] D. Amodei, R. Anubhai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, J. Chen, M. Chrzanowski, A. Coates, G. Diamos, et al., Deep speech 2: end-to-end speech recognition in English and Mandarin, International Conference on Machine Learning (2016) 173–182.
[6] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, M. Riedmiller, Playing atari with deep reinforcement learning, arXiv preprint arXiv:1312.5602 (2013).
[7] D. Silver, A. Huang, C.J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al., Mastering the game of go with deep neural networks and tree search, Nature 529 (7587) (2016) 484–489.
[8] A. Krizhevsky, I. Sutskever, G.E. Hinton, Imagenet classification with deep convolutional neural networks, in: Proceedings of the Advances in neural Information Processing Systems, 2012, pp. 1097–1105.
[9] K. He, X. Zhang, S. Ren, J. Sun, Delving deep into rectifiers: surpassing human-level performance on imagenet classification, in: Proceedings of the IEEE International Conference on Computer Vision, 2015a, pp. 1026–1034.
[10] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, in: Proceedings of the IEEE conference on computer vision and pattern recognition, 2016, pp. 770–778.
[11] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, N. Andrew, Deep learning with COTS HPC systems, in: Proceedings of the thirtieth International Conference on Machine Learning, 2013, pp. 1337–1345.
[12] NVIDIA, Tesla K40 GPU Active Accelerator, NVIDIA, 2013.
[13] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, Y. LeCun, Neuflow: a runtime reconfigurable dataflow processor for vision, in: Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops (CVPRW), IEEE, 2011, pp. 109–116.
[14] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, J. Cong, Optimizing FPGA-based accelerator design for deep convolutional neural networks, in: Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, ACM, 2015, pp. 161–170.
[15] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, et al., Going deeper with embedded FPGA platform for convolutional neural network, in: Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, ACM, 2016, pp. 26–35.
[16] S. Han, H. Mao, W.J. Dally, Deep compression: compressing deep neural networks with pruning, trained quantization and huffman coding, arXiv preprint arXiv:1510.00149 (2015).
[17] F.N. Iandola, M.W. Moskewicz, K. Ashraf, S. Han, W.J. Dally, K. Keutzer, Squeezenet: alexnet-level accuracy with 50x fewer parameters and less than 1MB model size, arXiv preprint arXiv:1602.07360 (2016).
[18] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, Y. Bengio, Binarized neural networks: training deep neural networks with weights andactivations constrained to +1 or −1, arXiv preprint arXiv:1602.02830 (2016).
[19] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, A.Y. Ng, Reading digits in natural images with unsupervised feature learning, in: Proceedings of the NIPS Workshop on Deep Learning and Unsupervised Feature Learning, 2011.
[20] M. Rastegari, V. Ordonez, J. Redmon, A. Farhadi, Xnor-net: imagenet classification using binary convolutional neural networks, European Conference on Computer Vision, Springer International Publishing (2016) 525–542.
[21] V. Sze, Y.-H. Chen, T.-J. Yang, J. Emer, Efficient processing of deep neural networks: a tutorial and survey, arXiv preprint arXiv:1703.09039 (2017).
[22] W. Liu, Z. Wang, X. Liu, N. Zeng, Y. Liu, F.E. Alsaadi, A survey of deep neural network architectures and their applications, Neurocomputing 234 (2017) 11–26.
[23] S. Ioffe, C. Szegedy, Batch normalization: accelerating deep network training by reducing internal covariate shift, International Conference on Machine Learning (2015) 448–456.
[24] A. Ng, J. Ngiam, C. Foo, Y. Mai, C. Suen, Backpropagation algorithm of ufldl tutorial, http://ufldl.stanford.edu/wiki/index.php/Backpropagation_Algorithm.
[25] G. Hinton, Neural Network for Machine Learning, Coursera, 2012.
[26] C. Farabet, C. Poulet, J.Y. Han, Y. LeCun, CNP: an FPGA-based processor for convolutional networks, in: Proceedings of the nineteenth International Conference on Field Programmable Logic and Applications (FPL), IEEE, 2009, pp. 32–37.
[27] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, O. Temam, Diannao: a small-footprint high-throughput accelerator for ubiquitous machine-learning, in: Proceedings of the ACM Sigplan Notices, vol. 49, ACM, 2014a, pp. 269–284.
[28] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, et al., Dadiannao: a machine-learning supercomputer, in: Proceedings of the fourty seventh Annual IEEE/ACM International Symposium on Microarchitecture, IEEE Computer Society, 2014b, pp. 609–622.
[29] N.P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, et al., In-datacenter performance analysis of a tensor processing unit, arXiv preprint arXiv:1704.04760 (2017).
[30] W. Wen, C. Wu, Y. Wang, Y. Chen, H. Li, Learning structured sparsity in deep neural networks, in: Advances in Neural Information Processing Systems, 2016, pp. 2074–2082.
[31] T.-J. Yang, Y.-H. Chen, V. Sze, Designing energy-efficient convolutional neural networks using energy-aware pruning, arXiv preprint arXiv:1611.05128 (2016).
[32] H. Li, X. Fan, L. Jiao, W. Cao, X. Zhou, L. Wang, A high performance FPGA-based accelerator for large-scale convolutional neural networks, in: Proceedings of the Twenty Sixth International Conference on Field Programmable Logic and Applications (FPL), IEEE, 2016, pp. 1–9.
[33] P. Gysel, Ristretto: Hardware-oriented approximation of convolutional neural networks, arXiv preprint arXiv:1605.06402 (2016).
[34] F. Li, B. Zhang, B. Liu, Ternary weight networks, arXiv preprint arXiv:1605.04711 (2016).
[35] C. Zhu, S. Han, H. Mao, W.J. Dally, Trained ternary quantization, arXiv preprint arXiv:1612.01064 (2016).
[36] S. Zhou, Z. Ni, X. Zhou, H. Wen, Y. Wu, Y. Zou, Dorefa-net: training low bitwidth convolutional neural networks with low bitwidth gradients, arXiv preprint arXiv:1606.06160 (2016).
[37] H. Alemdar, N. Caldwell, V. Leroy, A. Prost-Boucle, F. Pétrot, Ternary neural networks for resource-efficient ai applications, arXiv:1609.00222 (2016).
[38] W. Meng, Z. Gu, M. Zhang, Z. Wu, Two-bit networks for deep learning on resource-constrained embedded devices, arXiv preprint arXiv:1701.00485 (2017).
[39] R. Andri, L. Cavigelli, D. Rossi, L. Benini, YodaNN: an architecture for ultra-low power binary-weight cnn acceleration, IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. PP (2017) 1–14.
[40] R. Zhao, W. Song, W. Zhang, T. Xing, J.-H. Lin, M. Srivastava, R. Gupta, Z. Zhang, Accelerating binarized convolutional neural networks with software-programmable fpgas, in: Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, ACM, 2017, pp. 15–24.
[41] Y. Umuroglu, N.J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, K. Vissers, Finn: a framework for fast, scalable binarized neural network inference, in: Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, ACM, 2017, pp. 65–74.
[42] M. Kumm, P. Zipf, Pipelined compressor tree optimization using integer linear programming, in: Proceedings of the twenty fourth International Conference on Field Programmable Logic and Applications (FPL), IEEE, 2014, pp. 1–8.
[43] S. Gupta, A. Agrawal, K. Gopalakrishnan, P. Narayanan, Deep learning with limited numerical precision, CoRR 392 (2015). abs/1502.02551
[44] D. Williamson, Dynamically scaled fixed point arithmetic, in: Proceedings of the IEEE Pacific Rim Conference on Communications, Computers and Signal Processing, 1991., IEEE, 1991, pp. 315–318.
[45] Maxeler, MPC-X series, https://www.maxeler.com/products/mpc-xseries/.
[46] R. Collobert, K. Kavukcuoglu, C. Farabet, Torch7: A matlab-like environment for machine learning, in: Proceedings of the NIPS Workshop on BigLearn, in: EPFL-CONF-192376, 2011.

Please cite this article as: S. Liang et al., FP-BNN: Binarized neural network on FPGA, Neurocomputing (2017), https://doi.org/10.1016/j.neucom.2017.09.046

[47] I.J. Goodfellow, D. Warde-Farley, M. Mirza, A.C. Courville, Y. Bengio, Maxout networks., ICML (3) 28 (2013) 1319–1327.
[48] M. Lin, Q. Chen, S. Yan, Network in network, arXiv preprint arXiv:1312.4400 (2013).
[49] D. Kingma, J. Ba, Adam: A method for stochastic optimization, arXiv preprint arXiv:1412.6980 (2014).
[50] M. Courbariaux, Y. Bengio, J.-P. David, Binaryconnect: training deep neural networks with binary weights during propagations, in: Proceedings of the Advances in Neural Information Processing Systems, 2015, pp. 3123–3131.
[51] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-s. Seo, Y. Cao, Throughput-optimized opencl-based FPGA accelerator for large-scale convolutional neural networks, in: Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, ACM, 2016, pp. 16–25.
[52] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, Y. Bengio, Quantized neural networks: Training neural networks with low precision weights and activations, arXiv preprint arXiv:1609.07061 (2016).

Shuang Liang received the B.S. degree from the Institute of Microelectronics, Tsinghua University, Beijing, China, in 2011. He is working toward the Ph.D. degree at the Institute of Microelectronics, Tsinghua University, Beijing, China. He was a visiting scholar at the Department of Computing, Imperial College London, UK in 2016. His research interests include reconfigurable computing, hardware acceleration of machine learning algorithms and distributed systems.

Shouyi Yin received the B.S., M.S. and Ph.D. degrees in Electronic Engineering from Tsinghua University, China, in 2000, 2002 and 2005, respectively. He has worked in Imperial College London as a research associate. Currently he is with Institute of Microelectronics at Tsinghua University as an Associate Professor. His research interests include SoC design, reconfigurable computing and mobile computing. Prof. Yin has published more than 40 refereed papers, and served as TPC member or reviewers for the international key conferences and leading journals.

Leibo Liu received the B.S. degree in electronic engineering from Tsinghua University, Beijing, China, in 1999 and the Ph.D. degree in Institute of Microelectronics, Tsinghua University in 2004. He now serves as an Associate Professor in Institute of Microelectronics, Tsinghua University. His research interests include Reconfigurable Computing, Mobile Computing and VLSI DSP.

Wayne Luk received the M.A., M.Sc., and D.Phil. Degrees in Engineering and ComputingScience from the University of Oxford, Oxford, U.K. He is a Professor of Computer Engineering with Imperial College London, London, U.K. He was a Visiting Professor with Stanford University, Stanford, CA, USA. His current research interests include theory and practice of customizing hardware and software for specific application domains, such as multimedia, networking, and finance.

Shaojun Wei was born in Beijing, China in 1958. He received Ph.D. degree from Faulte Polytechnique de Mons, Belgium, in 1991. He became a professor in Institute of Microelectronics of Tsinghua University in 1995. He is a senior member of Chinese Institute of Electronics. His main research interests include VLSI SoC design, EDA methodology, and ASIC design.

Please cite this article as: S. Liang et al., FP-BNN: Binarized neural network on FPGA, Neurocomputing (2017), https://doi.org/10.1016/j.neucom.2017.09.046