Phoenix: A Low-Precision Floating-Point Quantization Oriented Architecture for Convolutional Neural Networks

Chen Wu, Mingyu Wang, Xiayu Li, Jicheng Lu, Kun Wang, Lei He

***Abstract*—Convolutional neural networks (CNNs) achieve state-of-the-art performance at the cost of becoming deeper and larger. Although quantization (both fixed-point and floating-point) has proven effective for reducing storage and memory access, two challenges – 1) accuracy loss caused by quantization without calibration, fine-tuning or re-training for deep CNNs and 2) hardware inefficiency caused by floating-point quantization – prevent processors from completely leveraging the benefits. In this paper, we propose a low-precision floating-point quantization oriented processor, named Phoenix, to address the above challenges. We primarily have three key observations: 1) 8-bit floating-point quantization incurs less error than 8-bit fixed-point quantization; 2) without using any calibration, fine-tuning or re-training techniques, normalization before quantization further reduces accuracy degradation; 3) 8-bit floating-point multiplier achieves higher hardware efficiency than 8-bit fixed-point multiplier if the full-precision product is applied. Based on these key observations, we propose a normalization-oriented 8-bit floating-point quantization method to reduce storage and memory access with negligible accuracy loss (within 0.5%/0.3% for top-1/top-5 accuracy, respectively). We further design a hardware processor to address the hardware inefficiency caused by floating-point multiplier. Compared with a state-of-the-art accelerator, Phoenix is 3.32× and 7.45× better in performance with the same core area for AlexNet and VGG16, respectively.**

***Index Terms*—low precision floating-point, deep learning, CNN, processor, quantization.**

I. Introduction

CONVOLUTIONAL neural networks and deep neural networks have demonstrated a breakthrough in performance for a broad range of applications, including object recognition [1], object detection [2] and speech recognition [3]. The advantages mainly come from the huge computational complexity and huge amount of data. This motivates researchers in both academia and industry to focus on accelerating CNNs by using CPU/GPU clusters [4], FPGAs [5] and ASICs [6]. Among them, customized accelerators/processors on FPGAs and ASICs have shown more promising throughput and energy efficiency than traditional CPU/GPU clusters [7]–[10].

Meanwhile, algorithm researchers keep on designing larger and deeper CNNs to improve performance in a broader range of scenarios. Such networks use larger amount of parameters, e.g. AlexNet [11] in 2012 has 249.51MB parameters, while VGG-16 [12] in 2014 has 553.43MB parameters. The great quantities of parameters lead to a big challenge for communication between off-chip and on-chip memory because of bandwidth constraints. On the other hand, algorithm developers also focus on reducing parameters while maintaining performance by introducing residual [13] or inception blocks [14], [15]. They successfully decrease the parameter size dramatically, e.g., from 553.43MB in VGG16 to 68.63MB in DenseNet201 [16]. However, the number of *convolutional* layers or *fully-connected* layers increases significantly from 13 to 201, making a deeper CNN. Such deep CNNs render approximate computing (*e.g.,* quantization) for CNNs more difficult, as we need to balance the approximation error imposed by more layers [17].

Progress has been made to alleviate bandwidth constraints by reducing the amount of parameters. Quantization, which approximates full precision parameters with low-precision numbers, has emerged as an efficient solution among various techniques. 8-bit fixed-point quantization is one of the most commonly used techniques [6], [23], [24], which results in a 4× data reduction. More aggressive studies tried to further reduce the data size by introducing binary neural network [25] or ternary neural network [26]. Meanwhile, the authors in [18], [21], [27], [28] focused on maintaining comparable accuracy for deeper CNNs during quantization, by using calibration, fine-tuning or re-training techniques with training data after quantization.

However, quantization with calibration, fine-tuning or re-training requires extra computing and training data which could be difficult for hardware developers. Xilinx in [21] proposed a low-precision floating-point quantization method with a single inference batch for calibration. They only proved their effectiveness for *slim* CNNs, which is not enough because deeper CNNs, *e.g.,* ResNet152 [13] and DenseNet201 [16], are more popular to gain higher performance in different applications. Moreover, Xilinx did not present any hardware for their quantization method. This is because floating-point multiply-accumulators (MACs) have lower hardware efficiency than fixed-point ones. This is another challenge for accelerator design.

In this paper, we propose a cooperative software/hardware approach to overcome the above challenges efficiently. Initially, we observe that non-uniform quantization for both weights and activations (we use activations to represent the outputs of a layer) incurs less quantization error compared with uniform quantization. At the same time, applying nor-

C. Wu, M. Wang, K. Wang and L. He are with University of California, Los Angeles, 90095, CA, USA (e-mail: chenwu1989@ucla.edu, mingyuw@ucla.edu, wangk@ucla.edu, lhe@ee.ucla.edu).

X. Li and J. Lu are with Shanghai Fudan Microelectronics Group Company Limited, Shanghai, China (e-mail: lixiayu@fmsh.com.cn, lujicheng@fmsh.com.cn).

TABLE I
COMPARISON OF EXISTING QUANTIZATION METHODS AND CORRESPONDING ACCELERATORS.

	Slim	Medium	Deep	Hardware	Note
Nvidia [18]	√	√	√	GPU	Fixed-point, extra cost for calibration
OLAccel [19]	√	√	√	Customized Accelerator	Fixed-point, extra cost for re-training
ARM [20]	√	√	x	CPU	Fixed-point and floating-point mixed, no hardware support
Xilinx [21]	√	√	x	CPU	No hardware support, extra cost for calibration
BFP [22]	√	√	x	Customized Processor	Block floating-point
Phoenix	√	√	√	Customized Processor	Floating-point, no calibration, fine-tuning nor re-training

malization on activations before quantization can reduce accuracy loss without any calibration, fine-tuning or re-training processes. Moreover, 8-bit floating-point multiplier achieves higher hardware efficiency than 8-bit fixed-point one when maintaining full precision results after multiplication. Therefore, we propose a normalization-oriented 8-bit floating-point quantizer to reduce the data size with negligible accuracy loss. Our proposed quantizer works for deep CNNs (more than 100 *convolutional/fully-connected* layers). On average, the top-1 accuracy loss is within 0.5%, while the state-of-the-art work [28] which can reach to such *deep* CNNs has a top-1 accuracy loss about 1% with fine-tuning. After reducing the memory and bandwidth requirements by using 8-bit floating-point quantization method, we design a floating-point based hardware processor, named Phoenix, to further address the problem of hardware efficiency caused by floating-point MACs. We maintain full precision for intermediate results after 8-bit floating-point multiplier, which saves 8.14× area compared with the 8-bit fixed-point multiplier. Compared with a state-of-the-art accelerator [29], Phoenix is 3.32× and 7.45× better for AlexNet and VGG16 in terms of performance when considering the same core area at TSMC 65nm technology, respectively. Phoenix also achieves 151× better in terms of energy compared with Nvidia TITAN Xp GPU with single image inference.

Our main contributions can be summarized as follows: 1) We make the key observations that normalization and non-uniform quantization can help reduce the quantization error even in *deep* CNNs. 2) Based on the observations, we propose a normalization-oriented 8-bit floating-point quantization method that dramatically reduces the parameter size with negligible accuracy loss for *deep* CNNs. 3) We further design a floating-point oriented hardware processor, named Phoenix, which is placed and routed in TSMC 28nm technology, to solve the hardware inefficiency caused by floating-point MACs.

II. BACKGROUND AND MOTIVATION

A. Background

1) CNNs: CNNs are used to classify or recognize objects by passing the inputs through multiple types of layers. In each layer, multiple neurons are constructed to process different inputs and pass the outputs to the next layer through connections, which store the weights for the network. Based on different processing procedures, the layers are typically divided into *convolutional, pooling, activation, normalization, fully-connected, residual* and *inception* layers. Among them, *convolutional/fully-connected* layers consume most portions of computation while *fully-connected* layers require largest memory to store weights. According to this, we divide the size of CNNs into three categories with respect to the number of *convolutional/fully-connected* layers: *slim* for less than 50 layers, *medium* for 50 to 100 layers, and *deep* for more than 100 layers (as shown in Table II where we report the detailed network information).

2) Over Parameterization in Neural Networks: Modern CNNs show dominant performance advantages in various application domains by enlarging the network architecture. However, such technique results in a heavy burden for memory capacity, communication bandwidth, computation and communication energy. Existing work tried to overcome such challenges, including algorithm-level techniques (*e.g.,* dropout [30], low-precision training [31], [32]) and architecture-level techniques (*e.g.,* approximate computing [33], low-precision operation [34], [35]). Among them, quantization, one of the approximate computing techniques, turns out to be effective. 8-bit fixed-point quantization is widely used by FPGA or ASIC for *slim* and *medium* CNNs [36]–[39]. Recently, it has advanced to *deep* CNNs, *e.g.,* ResNet101 on GPU for negligible accuracy loss with calibration [18]. 8-bit floating-point quantization still stays in the algorithm-level optimization and can apply only to *medium* CNNs. More aggressively, binary neural and ternary neural networks are proposed to further reduce memory, bandwidth and energy but at the cost of a larger accuracy loss [40]–[42].

3) 8-bit Floating-Point: Similar to the definition of 32-bit floating-point from the IEEE-754 standard [43], the binary representation of 8-bit floating-point number comprises *sign, mantissa* and *exponent* in order. The decimal value of 8-bit floating-point number is then calculated by:

$$V_{dec} = (-1)^S \times 1.M \times 2^{E-bias}, \tag{1}$$

where V_{dec} is the value in decimal, S, M and E are all unsigned values and denote the *sign, mantissa* and *exponent*, respectively. For $bias$ in Eq. (1), it is introduced for both positive and negative exponents as

$$bias = 2^{E_w - 1} - 1, \tag{2}$$

where E_w is the data width of E. The data widths for M and E are not fixed for 8-bit floating-point and we will emulate all combinations. In the later sections, we use the term $MaEb$ to indicate different combinations, where a and b indicate the bit width of M and E, respectively, e.g., $M3E4$ means the mantissa is 3 bits while the exponent is 4 bits.

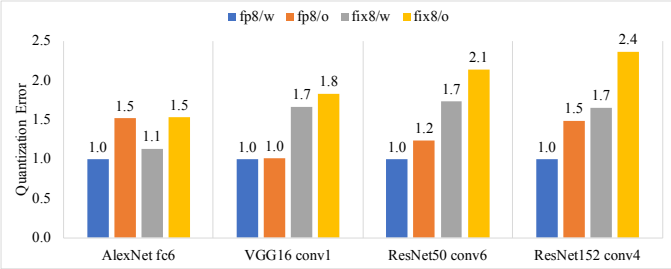

Fig. 1. Quantization error compared among floating-point and fixed-point quantizers with/without normalization (w: with normalization, o: without normalization).

There are three special definitions in IEEE-754 standard. The first is subnormal numbers when $E = 0$, and Eq. (1) is modified to:

$$V_{dec} = (-1)^S \times 0.M \times 2^{1-bias}. \quad (3)$$

Then, Infinity (Inf) and Not a Number (NaN) are the other two special cases, but not used in our work. This is because our saturation scheme saturates large numbers to the maximal number as illustrated in detail in Section III-A.

B. Motivation

1) Non-uniform quantizer: Non-uniform quantizer has been proven more accurate than uniform quantizer when the inputs follow the non-uniform distributions, such as Gaussian, Laplacian, and Gamma distributions [44]–[46]. Although the weights and activations in a CNN are more likely to follow a non-uniform distribution, few efforts have been done to explore non-uniform quantizer for CNNs. Uniform quantizers, *e.g.,* fixed-point quantizers, are exploited instead on both CPU/GPU [18], [47] and customized accelerators [39], [48], [49] because of their hardware efficiency. A few researchers recently support non-uniform quantizer [19]–[22]. However, they either require notably extra cost to compensate for quantization error or fail to have efficient hardware, as shown in Table I. Particularly, the work in [18] first reached *deep* CNNs, *e.g.,* ResNet101, with 8-bit fixed-point quantization by using calibration to compensate for quantization error. OLAccel [19] divided weights and activations into the low-precision region (97%) and high-precision region (3%). The data in low-precision region were then quantized to 4 bits while the data in high-precision region remained full precision. Re-training is required in order to maintain accuracy, otherwise the proportion of high-precision region must be increased, which leads to higher cost on hardware implementation. ARM [20] proposed a mixed quantizer, which used floating-point to represent weights and fixed-point to approximate activations. However, their approach remains in *medium* CNNs and lacks hardware support. Recently, Xilinx [21] developed the first low-precision floating-point quantization method. However, this approach needs to calibrate with an extra inference batch to maintain accuracy, and only *slim* and *medium* CNNs, *e.g.,* VGG16 and ResNet50, are validated. Moreover, the performance of running inference on a network is even worse than that of a full-precision network due to lack of hardware support. Block floating-point [22] divided data into different data blocks, in which the data had different mantissas and a shared exponent. However, this approach was only validated for *medium* CNNs. To conclude, existing quantization algorithms and corresponding architectures cannot completely benefit from non-uniform quantizers.

2) Observation: Previous quantization methods ignore the distribution properties of weights and activations. We fully analyze the distributions of weights and activations in different CNNs, and have two key observations: (1) A non-uniform quantizer fits weights and activations better than uniform quantizer as the weights and activations are more likely to follow Gaussian distributions; (2) Normalizing activations before quantization can further reduce quantization error. We select four representative layers – *fc6* in AlexNet, *conv1* in VGG16, *conv6* in ResNet50 and *conv4* in ResNet152 as driving examples. The quantization errors caused by 8-bit floating-point and 8-bit fixed-point quantizers with/without normalization are depicted in Figure 1. We normalize the quantization error of all the cases with respect to the error caused by 8-bit floating-point quantizer with normalization. As can be seen from Figure 1, 8-bit floating-point quantizer with normalization incurs the lowest quantization error among all the four cases. Fixed-point quantizer with/without normalization causes 1.50×/1.54× larger quantization error than 8-bit floating-point quantizer on average, respectively. In addition, with normalization, 8-bit floating-point and fixed-point quantizers both incur 1.3× less error. More detailed results about classification accuracy for CNNs will be illustrated in Subsection III-C.

As for the hardware implementation, we evaluate the hardware efficiency of 8-bit floating-point and 8-bit fixed-point multipliers by implementing them with Verilog and synthesizing with Synopsys Design Compiler (DC). We get another key observation that if we keep full precision for the intermediate results after multiplication, 8-bit floating-point multiplier consumes less area than 8-bit fixed-point multiplier. For instance, 8-bit floating-point multiplier with $M4E3$ consumes only 12.3% area of the 8-bit fixed-point multiplier. This is because $M4E3$ only needs a 5-bit unsigned fixed-point multiplier and a 3-bit unsigned fixed-point adder. The area savings are more significant for 8-bit floating-point multipliers with less mantissa bits (as shown in Figure 11). In short, one can design a hardware efficient processor with 8-bit floating-point multiplier, and maintain accuracy at the same time by using 8-bit floating-point quantizer with normalization.

III. Quantized Neural Network

In this section, we present the details of our proposed normalization-based 8-bit floating-point quantization method, including quantization process, quantization results and normalization analysis.

A. Quantization Process

Instead of quantizing the network directly, our quantization method is divided into three steps: normalization, merge normalization parameters, and quantization, as shown in Figure 2.

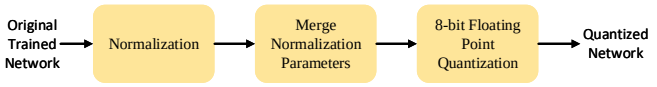

Fig. 2. Flow of 8-bit floating-point quantization.

1) Normalization: As illustrated in Subsection II-B, normalizing activations can reduce quantization error. Therefore, we first normalize activations based on the inference results on the original network. The normalization method is defined as (we exploit other normalization methods in Subsection III-C)

$$NORM\_O_i^m = \frac{O_i^m}{\sqrt{E((O^m)^2)}}, \tag{4}$$

where O_i^m and $NORM\_O_i^m$ indicate the ith activation before and after normalization for layer m, respectively; $E((O^m)^2)$ is the second moment of the activations, expressed as

$$E((O^m)^2) = \frac{1}{N}\sum_{j=0}^{N-1}(O_j^m)^2, \tag{5}$$

where N^m denotes the total number of activations of layer m with $N^m = OH^m \times OW^m \times OC^m$, OH^m, OW^m and OC^m being the height, width and channel number of the output layer, respectively.

2) Merge Normalization Parameters: As normalization is utilized to reduce quantization error in our proposed approach, it does not incur any accuracy loss. Therefore, we merge all the normalization parameters into the parameters of each layer, as shown in Figure 3(a). As can be seen in Figure 3(a), denormalization is first applied to the normalized outputs of layer m to make sure that normalization for layer m never incurs accuracy loss. After the operation (*e.g., convolution, pooling, activation*) of layer $m+1$, the outputs need to be normalized in order to reduce quantization error. As both the denormalization and normalization procedures are linear, they can be merged to the parameters of layer $m+1$, as marked with dashed line in Figure 3(a). To clearly explain the merging procedure, we use *convolutional* and *fully-connected* layers as driving examples. In the m-th *convolutional* layer or *fully-connected* layer, output neurons are calculated as

$$O^m = W^m \cdot I^m + b^m, \tag{6}$$

where W^m, I^m and b^m denote the weight matrix, input matrix and bias vector for layer m, respectively. Since the outputs of layer m are normalized with Eq. (4) and fed as the inputs of layer $m+1$, the denormalization process can be applied to the weights of layer $m+1$ as follows:

$$DE\_W^{m+1} = W^{m+1} \times \sqrt{E((O^m)^2)}, \tag{7}$$

where DE_W^{m+1} means the denormalized weights of layer $m+1$. In order to simplify the calculation of normalization for layer $m+1$, we also merge the normalization into the weights of layer $m+1$, formulated as

$$M\_W^{m+1} = DE\_W^{m+1} / \sqrt{E((O^{m+1})^2)} \tag{8a}$$

$$M\_b^{m+1} = b / \sqrt{E((O^{m+1})^2)}, \tag{8b}$$

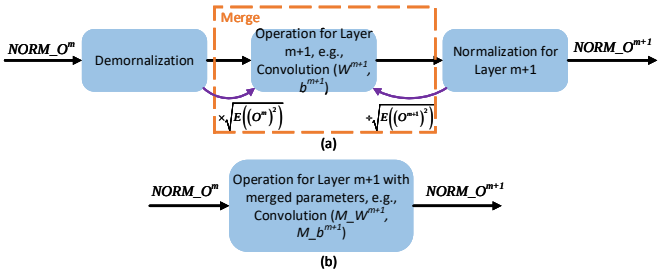

Fig. 3. (a) Flow of merging normalization parameters. (b) Simplified calculation process after merging all the normalization parameters.

where M_W^{m+1} and M_b^{m+1} are the weights and biases of layer $m+1$ with both denormalization and normalization. After merging denormalization and normalization into the weights and biases, we can get the normalized outputs with a simplified calculation process, as shown in Figure 3(b).

3) Quantization: 8-bit float point quantization is then applied to the activations and weights of the normalized network to save storage, and it is illustrated as follows:

$$V_{fp8} = round(V_{fp32} \times 2^{h\_s}, -MAX_{fp8}, MAX_{fp8}), \tag{9}$$

where V_{fp8} and V_{fp32} represent the 8-bit floating-point value and 32-bit floating-point value, respectively; MAX_{fp8} indicates the maximal number which can be represented with 8-bit floating-point, and h_s is the scaling factor to fit the data into the dynamic range of 8-bit floating-point data representation. The $round$ function in Eq. (9) rounds the data to the nearest value with saturation considered, formulated as

$$round(x, MIN, MAX) = \begin{cases} MIN & x <= MIN \\ MAX & x >= MAX \\ round(x) & \text{otherwise} \end{cases}, \tag{10}$$

where MIN and MAX are the minimal and maximal values, respectively.

During the quantization process, our aim is to find the optimal scaling factor to minimize the mean square error (MSE) compared with the full precision results, as illustrated by the pseudo-code in Algorithm 1. In our proposed quantization method, both the weights and activations are quantized. Since all the activations are normalized, we can set the scaling factor h_s the same for all the layers, which also simplifies the architecture design, especially for *residual* and *inception* layers. In the *residual* layer, element-wise addition are performed to the outputs of two previous layers, while in the *inception* layer, the outputs of several previous layers are concatenated into a single layer. If there are more than two layers, the hidden scale should be the same for all layers. This simplifies the architecture design.

4) Data flow in processor: The data flow to run inference of a quantized network in Phoenix is shown in Figure 4. In order to explicitly illustrate the data flow, we list the bit width in each step with $M4E3$ data format as an example. All the input image, weights and biases are represented by 32-bit floating-point. In our processor, the original input image and weights are quantized with $M4E3$ data format and

Algorithm 1 Quantization

while $m <= \#$ of layer **do**
  $i \leftarrow -10$
  while $i < 10$ **do**
    $W_{fp8} \leftarrow round(W_{fp32} \times 2^i, -MAX_{fp8}, MAX_{fp8})$
    $MSE \leftarrow \frac{1}{N}\sum_{k=0}^{N}(W_{fp8} - W_{fp32})^2$
    if $MSE < MSE_{min}$ **then**
      $h\_s(m) \leftarrow i$
      $MSE_{min} \leftarrow MSE$
    $i \leftarrow i + 1$
  $m \leftarrow m + 1$
return h_s

stored in external memory, while biases are quantized to 16-bit fixed-point to reduce quantization error. Multiplications are performed with the quantized image and weights, and the 15-bit floating-point ($M10E4$) products are aligned and truncated to t-bit fixed-point in the truncating module (the selection of t will be explained in Subsubsection VI-A2). In this way, all the accumulation can be done in fixed-point accumulators, which consumes fewer resources than floating-point accumulators. The final outputs in each output channel are converted to $M4E3$ floating-point again (and stored in the external memory) before used by another CNN layer. In the data flow, the truncating module and final data conversion step introduce bit truncation and precision loss. However, the precision loss has little impact on the final accuracy and is validated in Subsection III-B with comprehensive experimental results.

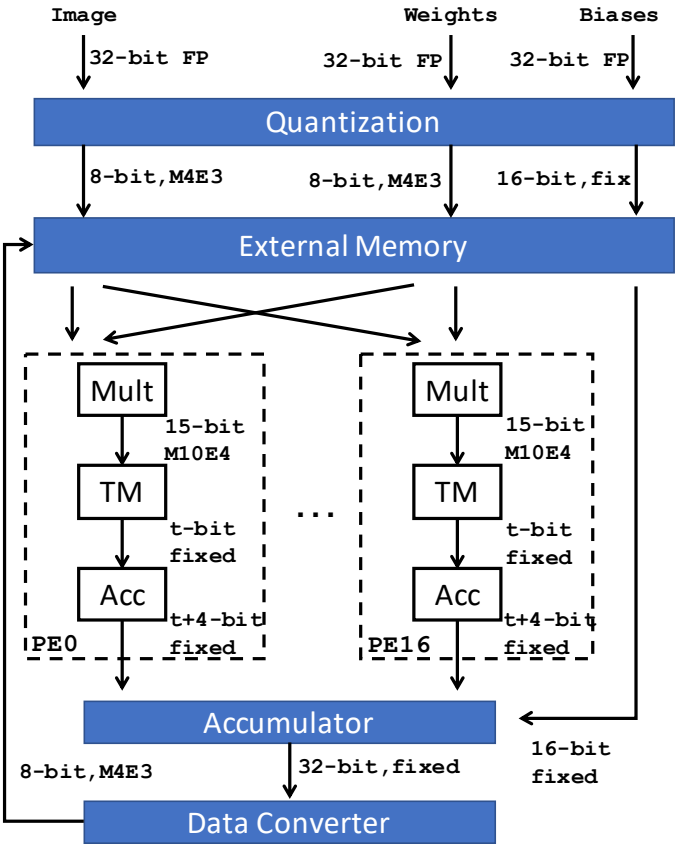

Fig. 4. The data flow in Phoenix with $M4E3$ data format as an example (FP: floating-point, Mult: 8-bit floating-point multiplier, TM: truncating module, Acc: accumulator).

B. Quantization Results

We implement our quantization method with C language based on the Darknet framework [50], by which the validation accuracy with single center-crop is evaluated with the ImageNet validation set (50,000 labelled images) [51]. Our quantization process is run on an Intel (R) Core (TM) i9-7960X CPU working under 2.86GHz, while the evaluation process is run on a Nvidia TITAN Xp GPU.

Six representative CNNs including the *slim, medium* and *deep* CNNs are evaluated, as listed in Table II. The detailed validation accuracy on the quantized network with all the benchmarks are shown in Figures 5 and 6. We emulate all 8 different (mantissa, exponent) combinations to validate accuracy of the quantized CNNs. The 32-bit floating-point results are included as the baseline.

In Figures 5 and 6, the dashed lines illustrate the 32-bit floating-point baseline, while the values above the dashed lines are the accuracy loss compared with the baseline. One can see that our proposed approach can maintain top-1 and top-5 accuracy comparable to the baseline. On average, the top-1 and top-5 accuracy loss is within 0.5% and 0.3% compared with the full precision results. Particularly, $M5E2$ always achieves the highest accuracy compared with the other cases. For the cases with more than or equal to 3-bit mantissa, they maintain a low accuracy loss for all the six CNNs, while the case with less than 3-bit mantissa can hardly find accurate results. We also compare our proposed approach with the fixed-point situation, marked as $M7E0$ in the figures ($M7E0$ means 7-bit mantissa and no exponent, exactly fixed-point). As shown in Figures 5 and 6, $M4E3$ and $M5E2$ outperforms the fixed-point for all six benchmarks. This is consistent with our observation that non-uniform quantization fits better than fixed-point quantization.

$M4E3$ and $M5E2$, which achieve the two best accuracies among all the test cases, are also compared with five typical approaches. We report both the top-1 and top-5 accuracy for all six benchmarks in Table III, where "-" indicates no reported results in the literatures. We use the normalized top-1 accuracy in Table III for the approaches proposed by ARM [20] and

TABLE II
CHARACTERISTICS OF CNN BENCHMARKS. GOP IS GIGA-OPERATIONS NEEDED BY ONE 224×224 RGB IMAGE.

CNN	Type	Operations	Model Weights
AlexNet	*slim*	2.27 GOP	249.51 MB
VGG16	*slim*	30.94 GOP	553.43 MB
ResNet50	*medium*	9.74 GOP	46.05 MB
ResNet101	*medium*	19.70 GOP	166.37 MB
ResNet152	*deep*	29.39 GOP	229.39 MB
DenseNet201	*deep*	10.85 GOP	68.63 MB

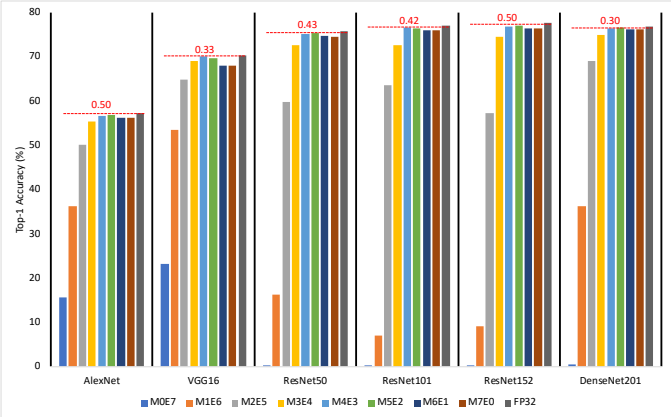

Fig. 5. Top-1 accuracy for different (mantissa, exponent) combinations with respect to different CNNs.

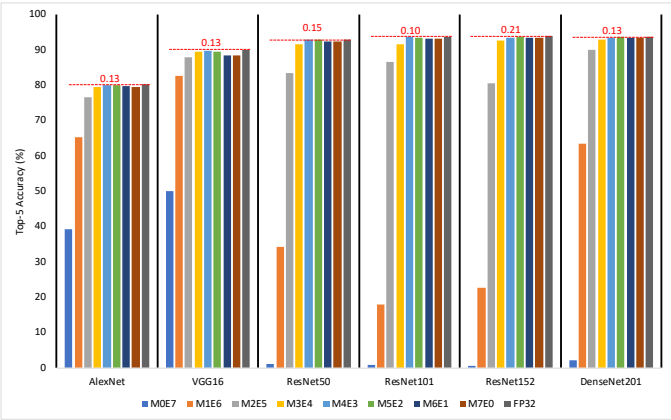

Fig. 6. Top-5 accuracy for different (mantissa, exponent) combinations with respect to different CNNs.

Xilinx [21] as reported in their paper. The top-1 and top-5 accuracy in Table III show that our 8-bit floating-point quantization method without any re-training can outperform the literatures in most cases. Moreover, besides the approach proposed by Nvidia [18], our method is the only one that can reach *deep* networks.

On the other hand, $M4E3$ also has a comparable top-1 and top-5 accuracy compared with the 32-bit floating-point baseline. This is one reason why we design our processor with the $M4E3$ case. The other reason is that 8-bit floating-point multiplier implemented with $M4E3$ costs less area and gains higher working frequency than $M5E2$ does, which will be discussed in detail in Subsection VI-A.

C. Normalization Analysis

In this subsection, different normalization methods are explored considering their influence on accuracy. During normalization, we first use a mini-batch of 100 test images to fetch the normalization parameter. This is because the normalization parameter is defined as the second moment of the output feature map, and more data can better describe its statistical characteristics. We then define the normalization parameter as the mean and standard deviation of the output feature map,

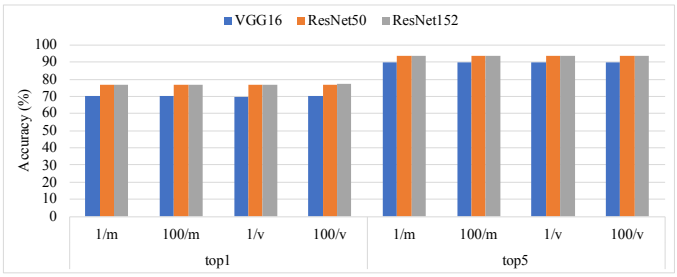

Fig. 7. Top-1 and Top-5 accuracy with different normalization methods and different number of test images (1/m: normalize with 2nd moment and 1 test image; 100/m: normalize with 2nd moment and 100 test images; 1/v: normalize with mean and standard deviation and 1 test image; 100/v: normalize with mean and standard deviation and 100 test images).

and the normalization process is changed from Eq. (4) to Eq. (11).

$$NORM\_O_i^m = \frac{O_i^m - \mu(O^m)}{\sigma(O^m)}, \tag{11}$$

where $\mu(O^m)$ and $\sigma(O^m)$ denote the mean and standard deviation of the outputs of layer m. During the evaluation process, both one test image and a mini-batch of 100 test images are applied to this normalization method. We take three representative networks – VGG16, ResNet50 and ResNet152 – for *slim, medium* and *deep* networks as driving examples. The evaluation results are shown in Figure 7. One can see that normalization with a mini-batch of 100 test images outperforms that with one image for all cases. On average, the improvement of applying a mini-batch is 0.05% for both top-1 and top-5 accuracy. However, the average quantization time of applying a mini-batch increases by 50×, *e.g.,* from 17s to 983s for ResNet152. The normalization method with mean and standard deviation turns out to have almost the same accuracy results as the normalization method with second moment. In this way, we select the normalization method with second moment, since this incurs fewer efforts in merging all the normalization parameters.

IV. Processor Architecture

In this section, we discuss in detail the architecture of Phoenix, which efficiently overcomes the challenge of hardware inefficiency caused by floating-point based operations.

A. Overview

The architecture of Phoenix is depicted in Figure 8. We develop a floating-point function unit (FPFU), which is composed of multiples of processing elements (PEs), to compute the outputs of a layer in parallel. The PE, which is the key component of Phoenix, is designed to efficiently perform multiplications and additions of 8-bit floating-point data. The on-chip memory system (MS) consists of three buffers: input feature map buffer (IFMB), weight buffer (WB) and output feature map buffer (OFMB). All these three buffers are ping-pong architecture to hide the communication time between on-chip and off-chip memory through direct memory access (DMA) module. We design a central control module (CCM) to arbitrate between different modules. The CCM decodes

TABLE III
ACCURACY COMPARISON BETWEEN $M4E3$, $M5E2$, REFERENCES AND FP32. "-" MEANS NO REPORTED RESULTS. RESULTS FOR ARM AND XILINX ARE CONVERTED ACTUAL ACCURACY FROM THE NORMALIZED ACCURACY REPORTED IN THEIR PAPERS WITHOUT CIRCUIT IMPLEMENTATIONS, WHILE OTHERS ARE BASED ON CIRCUIT IMPLEMENTATIONS.

	Top-1 Accuracy (%) Top-5 Accuracy (%) for each network											
	AlexNet		VGG16		ResNet50		ResNet101		ResNet152		DenseNet201	
Nvidia [18]	57.05	80.06	70.84	-	73.10	91.06	74.40	91.73	74.70	91.78	-	-
ARM [20]	56.71	-	70.38	-	-	-	-	-	-	-	-	-
Xilinx [21]	-	-	-	-	75.80	-	-	-	-	-	-	-
BFP [22]	-	-	68.32	-	72.76	-	-	-	-	-	-	-
V-Quant [28]	56.24	78.95	71.77	90.66	-	-	-	-	78.35	93.95	77.32	93.51
FP32 (Baseline)	57.28	80.18	70.38	89.81	75.80	92.90	77.10	93.70	77.60	93.83	76.85	93.62
Phoenix ($M4E3$)	**56.69**	**79.99**	**70.05**	**89.68**	**75.25**	**92.75**	**76.68**	**93.60**	**76.79**	**93.44**	**76.40**	**93.43**
Phoenix ($M5E2$)	**56.77**	**80.05**	**69.74**	**89.49**	**75.37**	**92.71**	**76.43**	**93.33**	**77.05**	**93.62**	**76.55**	**93.49**

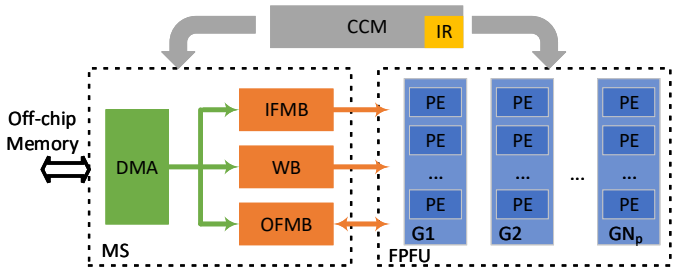

Fig. 8. The Architecture of Phoenix.

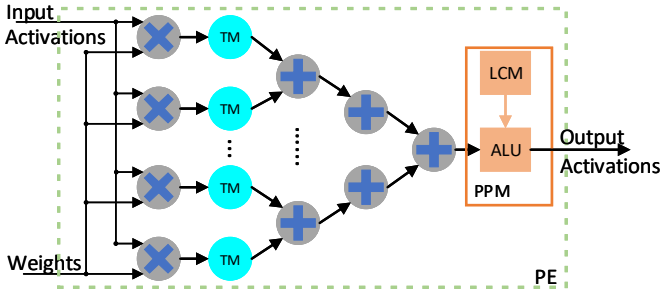

Fig. 9. The architecture of a PE.

various instructions stored in the instruction RAM (IR) into detailed signals for other modules.

B. Architecture of PE

Phoenix is designed to perform floating-point multiplications and additions efficiently for performance gain and energy reduction. In Phoenix, floating-point numbers are operated inside the PEs, which construct the FPFU. The FPFU receives activations from IFMB and weights from WB, then distributes to different PEs. Each PE operates the dot product of two vectors and stores the results into OFMB. Inside each PE, we design a fully pipelined data-flow-based architecture, as shown in Figure 9. Once a PE receives two vectors, it distributes the data to N_m multipliers, whose full-precision results are transferred into a truncating module (TM). The full-precision data are aligned to have same scale and truncated into low-precision to simplify the design of the adder tree, which is followed to sum up all the products. A post process module (PPM) then accumulates, activates and stores the data into the OFMB.

1) 8-bit floating-point multiplier: 8-bit floating-point numbers are represented with scientific notations in the sign-and-magnitude format, as illustrated in Eqs. (1) and (3). The multiplication of two numbers is then divided into three fixed-point components: (1) XOR of the signs; (2) multiplication of mantissas; (3) addition of exponents. Take the $MaEb$ format as an example. An $(a+1)$-bit unsigned multiplier and a b-bit unsigned adder are designed inside each 8-bit floating-point multiplier of the PE. We design the multiplier to be $(a+1)$-bit because the first bit of mantissa is hidden for saving storage – "1" for normal numbers and "0" for subnormal numbers. Meanwhile, $bias$ is not included during addition, because this is the same for all the numbers and we can address this at the last step to simplify the adders. The $2a+b+4$ bits result is arranged in the *sign-product-sum* order and delivered into the truncating module.

2) Truncating module: The full-precision results from the 8-bit floating-point multipliers incur high burden for adders because the results are in different scales, and only data in the same scale can be summed up. Therefore, we develop a truncating module to first align the data to the same scale and then truncate them into low-precision with t bits to simplify the design of adders (the selection of t will be explained in Subsection VI-A with experimental results). In the alignment and truncation process, the dot position is kept unchanged, and the data width is truncated to t bits according to the exponent. As shown in Figure 10, we set the scale to 2^b and the *sum* will be compared with 2^b to decide whether the product is shifted left or right. This is the alignment process to keep the full precision. The aligned value is then cut into t bits, where we consider saturation with overflow and round when discarding least significant bits. In this way, the adder tree can be implemented with fixed-point adders, which is more hardware efficient than floating-point adders.

3) Post-processing module: The post-processing module consists of a local controller module (LCM) and an arithmetic logic unit (ALU). Controlled by the CCM of our processor, LCM indicates the control signals for computing a layer, as well as the activation parameters. The ALU is designed to perform simple calculations, *e.g.,* add biases, activate with ReLU, accumulate intermediate results and convert the output to 8-bit floating-point format.

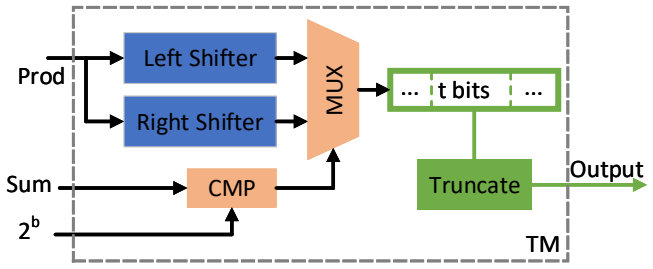

Fig. 10. Truncating process.

C. Memory System

The memory system in our processor is divided into three parts: an input feature map buffer (IFMB), an output feature map buffer (OFMB) and a weights buffer (WB), as shown in Figure 8. All the buffers are designed in ping-pong manner to hide the time of DMA memory accesses under the time of computation.

The width of IFMB is set to be $N_g \times N_m \times 8$ bits as to provide N_m 8-bit floating-point input activations to N_g PEs each time. The N_g PEs are grouped into one PE group to focus on N_g output activations on the same output feature map, which share the same weights. In our processor, we design N_p PE groups, where the input activations are shared. In this way, the width of WB is chosen to be $N_p \times N_m \times 8$ bits to provide weights for N_p different output channels. The OFMB is then set to have the width of $N_p \times N_g \times 16$ bits to save $N_p \times N_g$ output activations. Although each pixel in the output feature map is represented with 8-bit floating-point, we keep the intermediate results with 16-bit precision to reduce accuracy loss.

The parameters N_g, N_m and N_p are decided to trade off between area, overall performance and energy, which are discussed in Subsection VI-A. The sizes of the three buffers are also decisive to the area, overall performance and energy. Previous proposed work applies large enough buffers to store all the activations or weights for one layer [37] to avoid costly off-chip memory access. However, such designs incur large area and unscalability for larger and deeper CNNs. In our processor, we trade off among the area, scalability, performance and energy, and employ the smallest sizes which can hide the DMA communication time. After exploring different CNNs and buffer sizes, we deploy 64KB, 64KB and 32KB for IFMB, OFMB and WB, respectively. During inference on our processor, only when all the input feature map have been processed and reused, or all the weights have been processed and reused, or OFMB is full, will the off-chip memory be accessed for loading new input feature maps, loading new weights or storing output feature maps, respectively.

D. Central Control

The CCM is designed to arbitrate among different modules and control the whole execution process. Firstly, it decodes the instructions from IR efficiently and sets the corresponding control registers. Then, different modules are activated according to the control registers and the status of each module are monitored by the control registers as well. Finally, the CCM decides when to fetch the next instruction from the feedback of the control registers. We also design a compiler to generate the block-level instructions.

V. Experimental Methodology

In this section, we introduce the experimental methodology. We develop Phoenix in Verilog and then synthesize, place and route it with IC Compiler using TSMC 28nm library. The energy cost is evaluated with the PrimeTime PX tool based on the waveform files obtained from post-implementation simulation. The off-chip memory access energy is estimated by using the tools provided by MICRON [52]. We also design a cycle accurate simulator to estimate the throughput for different CNNs.

Baselines. We select the CPU, the GPU and customized accelerators as baselines.

CPU and GPU. We use the Darknet framework to evaluate the benchmarks on an Intel (R) Core (TM) i9-7960X CPU working under 2.86GHz. We also use darknet to evaluate the benchmarks on a Nvidia TITAN Xp GPU, which has a 12GB DDR5. Furthermore, we use the cuBLAS [53] to implement the benchmarks on the GPU.

Customized Accelerator. We compare Phoenix against three customized accelerators: 8-bit fixed-point processor, Eyeriss [54] and OLAccel [19]. The 8-bit fixed-point processor is implemented in the same architecture and with the same parameters as that of Phoenix. We select Eyeriss as another baseline because it provides an open-source estimation tool for comparison [29]. We perform an ISO-area comparison between Eyeriss and Phoenix. Thus, we scale Phoenix from TSMC 28nm to TSMC 65nm, which is the technology node used by Eyeriss. We allocate the same core area and the same amount of on-chip memory. We first get the core area of 168 8-bit PEs in Eyeriss. Then the number of PEs for our processor is decided to meet the area target, and the configuration is shown in Table IV. OLAccel does not provide any open-source estimation tool, but reports their comparison results with Eyeriss. Therefore, we compare our results with the reported results in OLAccel.

Benchmarks. The six CNNs listed in Table II are used as benchmarks for comparison with the CPU and the GPU. Only *convolutional* layers in AlexNet and VGG-16 are utilized for comparison with Eyeriss, since Eyeriss only supports *convolutional* layers.

VI. Experimental Results

A. Hardware Characteristics

TABLE IV
Configurations of Eyeriss and Phoenix.

	Eyeriss	Phoenix
# of PEs	168	768
core area (mm^2)	0.96	1.03
On-chip Memory	181.5KB	WB:51.5KB IFMB, OFMB: 64KB

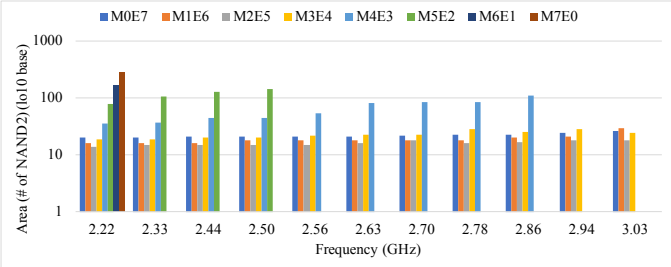

Fig. 11. Area (equivalent to the number of NAND2) for 8-bit floating-point multiplier with respect to frequency.

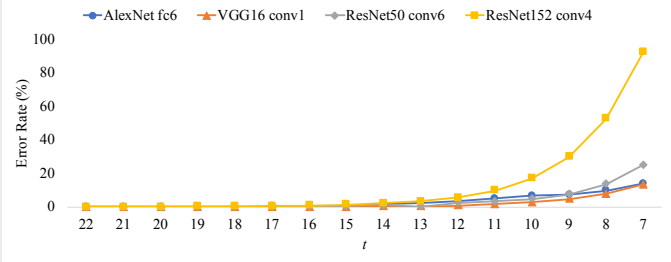

Fig. 12. Error rate compared with the quantized network for different layers with respect to different truncating parameter t.

1) Multiplier: We explore all the eight (mantissa, exponent) combinations for 8-bit floating-point multipliers before the implementation of our processor. We implement multipliers with Verilog and synthesize them with Synopsys design compiler to evaluate the maximal working frequency and corresponding area at TSMC 28nm technology node, as shown in Figure 11. $M4E3$ and $M5E2$ are able to work under 2.86GHz and 2.5GHz with the area equivalent to 112 and 146 2-input NAND gates, respectively. The maximal frequency for $M6E1$ and $M7E0$ (also known as fixed-point version) is 2.22GHz, while the area is equivalent to 169 and 285 2-input NAND gates, respectively. As for the cases with less than 4-bit mantissa, they all reach the frequency of 3.03GHz, while the corresponding area is 26, 29, 18 and 24 2-input NAND gates, respectively. As discussed in Subsection III-B, $M4E3$ and $M5E2$ outperform all the other cases for accuracy. In terms of working frequency and area, $M4E3$ is better than $M5E2$. Particularly, the maximal frequency of $M4E3$ is 14.4% higher than that of $M5E2$. As to the area, $M4E3$ is 3.24× more efficient than $M5E2$ when working under $M5E2$'s maximal frequency. Meanwhile, compared with the fixed-point (marked as $M7E0$ in the figure) version, $M4E3$ reduces the area by 8.14× at the same working frequency. Although the case $M3E4$ outperforms $M4E3$ in both working frequency and area, the top-1 and top-5 accuracy loss of $M3E4$ are 3.73× and 3.48× higher than that of $M4E3$, respectively. This is not acceptable since maintaining accuracy is one of the key motivations in our work.

2) Truncating Bit-width: In the truncating module (TM), we cut the full precision number into t bits to simplify the adder design. To convert the product of two $M4E3$ numbers to the same scale in full precision, we need at least 22 bits. This is because the mantissa of the product is 10 bits, and the exponent ranges from 2 to 14. Note that we exploit different ts to evaluate the errors incurred by reducing precision. We take four representative layers – *fc6* in AlexNet, *conv1* in VGG16, *conv6* in ResNet50 and *conv4* in ResNet152 – as driving examples. We depict the error rate for output activations of each layer caused by reducing t from 22 to 7, as shown in Figure 12. Compared with quantized networks, reducing t from 22 to 14 incurs little error to activations of the layers (less than 1%). However, when t is less than 14, the error increases dramatically, e.g., the error increases from 4.38% to 92.68% for *conv4* in ResNet152, which is not acceptable. This advantage comes from our quantization method. During the quantization process, we normalize all activations and use a Gaussian distribution to approximate the activations. When we truncate it from 22 bits to 14 bits, we discard large values located outside the 3σ region, which has less impact on the accuracy of the layer than that of the values within the 3σ region. Moreover, when t is less than 14, more values inside the 3σ region are discarded, which results in a higher error rate.

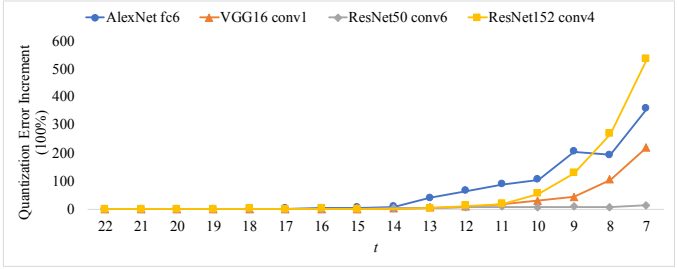

Fig. 13. Error rate increment with respect to different truncating parameter t.

We further explore the influence on quantization error when introducing the truncating technique, as shown in Figure 13. When t changes from 22 to 14, this error remains in the same level. However, when t decreases from 14 to 7, the quantization error increases significantly, e.g., the quantization error increases by 3.57% to 538.30% for *conv4* in ResNet152. This is not acceptable because it will incur large accuracy loss. Two experimental results demonstrate that if we truncate the bit width of activations from 22 bits to 14 bits, we will not suffer from large quantization error. Therefore, we select $t = 14$ in our current design. We speculate that in general t can be selected based on 3σ of Gaussian distribution in this study.

3) Memory System Parameters: In our current design, we select $N_g = 4$ which is also efficient with small feature map size. In CNNs based on the ImageNet data set, the smallest size of a feature map is always 14×14 or 8×8, which indicates that calculating 4 pixels in parallel achieves the highest efficiency. The parameter N_m is chosen to be 32 with the consideration of resource utilization. Most CNNs have their channel number to be multiples of 32, except the first layer. In the first layer, the channel number is 3 as they have RGB images as inputs. When $N_m > 32$, e.g, $N_m = 64$, 50% of the multipliers will be wasted for the layers like *conv1*, *conv2* in AlexNet, *conv1* in VGG-16, as the channel number of these layers is 32 or 96.

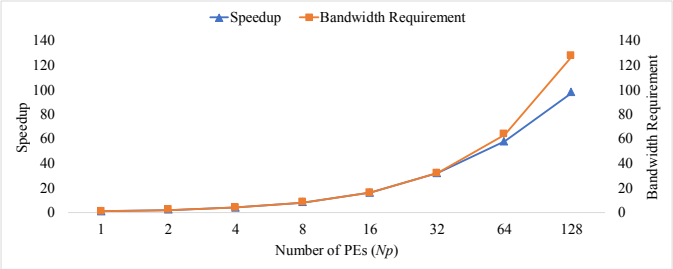

Fig. 14. Speedup and bandwidth requirement versus number of PEs.

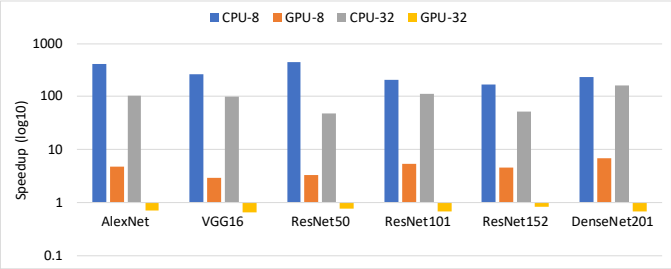

Fig. 15. The speedup of Phoenix over Intel (R) Core (TM) i9-7960X CPU and Nvidia TITAN Xp GPU.

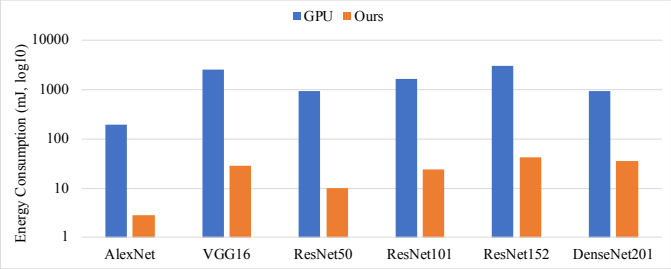

Fig. 16. Energy comparison between Nvidia TITAN Xp GPU and Phoenix.

At the same time, N_p is decided by considering the resource utilization and the memory bandwidth. As shown in Figure 14, we explore the speedup and bandwidth requirement for VGG-16 when enlarging N_p from 1 to 128 by multiplying 2 each time. When the N_p is less than 64, the speedup doubles as N_p doubles. This is because all PEs are working efficiently. However, the speedup turns out to have a saturation when $N_p >= 64$. This is because large amount of PEs are wasted when the output channel number is less than 64. Furthermore, the bandwidth requirement doubles when N_p doubles as the on-chip memory need to provide enough data for all PEs. However, the increment in bandwidth also enlarges the buffer size to hide the time of off-chip memory access, which will incur larger area and energy. Therefore, we select $N_p = 16$ in our current design to trade-off between performance, resource utilization, area and energy.

4) Implementation: Phoenix is implemented in TSMC 28nm technology node, and the results after placement and routing are listed in Table V. At 0.9V, the peak throughput is 2.048 TMAC/s (TMACS) with a 1GHz core clock rate. The core area is $1.44mm^2$ with the total power of $1091.2mW$. Among them, the memory system (including IFMB, WB, OFMB) consumes 50.7% of the total power.

B. Performance

We compare the execution time of Phoenix against CPU and GPU on the six CNNs listed in Table II. On CPU and GPU, we evaluate the CNNs with both 8-bit floating-point and 32-bit floating-point precision (i.e., CPU-8, GPU-8, CPU-32, GPU-32).

We normalize the execution time against that of Phoenix to gain the speedup, as shown in Figure 15. Compared with CPU and GPU in 8-bit precision, Phoenix achieves 290.7× and 4.7× speedup, respectively. This is because CPU and GPU do not have any optimization on 8-bit floating-point operations. During the evaluation, we make the conversions between 8-bit floating-point and 32-bit floating-point representations for each layer on both CPU and GPU, which incurs a large time burden. We also evaluate the 32-bit precision on CPU and GPU, and Phoenix is still 95.6× faster than CPU and is 70% as fast as Titan Xp. Such speedups against CPU mainly come from the FPFU modules in Phoenix, where we optimize multiplications and additions of the 8-bit floating-point number. Although Phoenix achieves 70% of the speed of Titan Xp, it consumes less energy than Titan Xp does, as explained in Subsection VI-C.

TABLE V
THE RESULTS OF Phoenix AFTER PLACEMENT AND ROUTING.

Category	Parameters
Technology	TSMC 28nm HPC+ 1P10M
Core Size	$1.2 \times 1.2mm^2$
Core Power	$1091.2mW$
# of MACs ($N_m \times N_g \times N_p$)	2048
Supply Voltage	Core 0.9V
Clock Rate	1GHz
Peak Throughput	2.048TMAC/s
Arithmetic Precision	8-bit floating-point

C. Energy

We report the energy comparison between Phoenix and Titan Xp across all the six benchmarks, as shown in Figure 16. We include the off-chip memory access energy in this comparison. Compared with GPU, Phoenix achieves 70.5× better energy efficiency on average. This advantage comes from the 8-bit floating-point quantization, which reduces the memory amount in 4×, thus reducing the number of memory accesses. Moreover, Phoenix is designed to reuse both the weights and activations, which also leads to the reduction in off-chip memory accesses. Regarding the processor energy cost without off-chip memory access, we can achieve 151× compared with GPU. This results demonstrate the high energy efficiency of Phoenix.

Meanwhile, the energy breakdown of Phoenix with off-chip memory access for all evaluated benchmarks is shown in Figure 17. We can observe that the energy consumed by off-chip memory access is more than 50%. This is because we still need to transfer large amount of data from the off-chip memory to the on-chip memory system. The result also

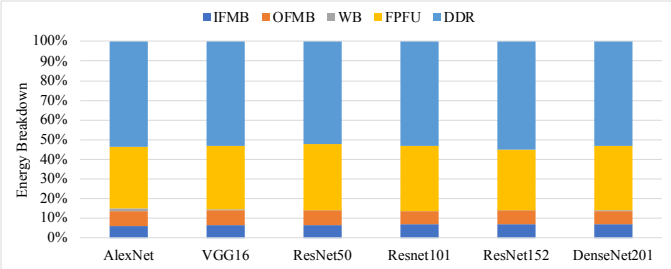

Fig. 17. Energy breakdown of Phoenix with off-chip memory accesses.

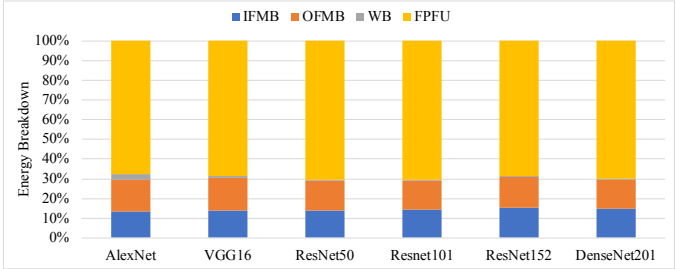

Fig. 18. Energy breakdown of Phoenix without off-chip memory accesses.

informs us that quantization and data reuse are the key points for reducing the energy caused by off-chip memory access.

We further present the energy breakdown without off-chip memory access in Figure 18. The on-chip memory system (including IFMB, WB and OFMB) consumes about 30% of the total energy. The WB consumes the least energy, because we reuse the weights for all the input activations stored in IFMB, which reduces the accesses for WB. Moreover, the WB consumes more energy in AlexNet and VGG-16 than that of other benchmarks. This is because these two networks have relatively larger *fully-connected* layers, which benefits less in Phoenix. As for the FPFU, it consumes about 70% of the total energy. The FPFU, which is a fully-pipelined data-flow-based architecture, works all the time when inputs exist. We do not skip zero activations nor weights in our design, which makes the FPFU consume more energy.

D. Discussion

1) Comparison with Fixed Point Processor: We also compare Phoenix with the 8-bit fixed-point processor with the same architecture and the same parameters. Hence, the two processors have the same performance. Based on this, Phoenix uses 1.2× less area than the 8-bit fixed-point processor according to the DC synthesize results, and both processors have similar energy. Although the fixed-point processor does not need the truncating module and also benefits from the data reuse technique, the 8-bit fixed-point multipliers consumes more area than 8-bit floating-point multipliers as explained in Subsubsection VI-A1.

2) Comparison with Other Accelerators: Phoenix is also compared with a state-of-the-art fixed-point accelerator, Eyeriss, as it provides the open-sourced estimation tool [29]. For fair comparison, we scale Phoenix to have the same area and the same on-chip memory as Eyeriss in 65nm technology node, as shown in Table IV. We only compare the computation time for *convolutional* layers, because Eyeriss only supports *convolutional* layers. All *convolutional* layers in AlexNet and VGG-16 are compared in Tables VI and VII.

One can seen from Tables VI and VII, Phoenix has a speedup of 3.32× and 7.45× for AlexNet and VGG-16, respectively. This is mainly because we have more multipliers for the same area as Eyeriss (768 in Phoenix versus 168 in Eyeriss). In Phoenix, we flatten input RGB images according to the kernel size to save computation time. Therefore, we have larger speedup for the first *convolutional* layer for both benchmarks. As for layers with small input feature map size, e.g. CONV5 in AlexNet (13×13) and CONV5-1, CONV5-2, CONV5-3 in VGG-16 (14×14), the advantages in performance reduce. In these layers, the whole input feature map can be stored in the IFMB, hence Phoenix benefits less from data reuse than the layers with large input feature map size.

We further compare with the OLAccel [19], whose hardware is based on the V-Quant quantization method. It does not provide any open-sourced estimation tools so that we compare with their reported performance comparison results with Eyeriss. Regarding AlexNet and VGG-16, OLAccel outperforms Eyeriss with 3.55× and 5.56× in performance, respectively. Compared with OLAccel, our processor does not have better performance in AlexNet, because they skip zero activations during calculation, which saves execution cycles. However, this does not work on larger CNN. Our processor outperforms OLAccel in VGG-16 with a 1.34× reduction in execution cycles. There are two reasons for that: 1) Phoenix has more multiplier units than OLAccel (768 in Phoenix versus 576 in OLAccel); 2) The quantization method used in OLAccel requires keeping around 3% of the weights and activations with full-precision, which brings the burden to the hardware and the execution process.

VII. Related Work

CNNs are typically over-parameterized, and extensive studies in recent years focus on CNN approximation algorithms, including weight reduction and quantization [55]. A deep compression method was proposed in [56], [57]. Weight pruning along with iterative re-training was first applied to CNNs to reduce weights [56], after which weights were quantized using k-means clustering. The quantized network was then re-trained to compensate for quantization error. Finally, they used Huffman coding to represent the quantized weights to save memory [57].

TABLE VI
Performance comparison against Eyeriss for AlexNet (ms).

Layer	Eyeriss	Phoenix	Speedup
CONV1	4.1	0.8	5.13×
CONV2	9.8	3.2	3.06×
CONV3	5.5	1.1	5.00×
CONV4	4.0	1.6	2.50×
CONV5	2.5	1.1	2.27×
Total	**25.9**	**7.8**	**3.32×**

TABLE VII
PERFORMANCE COMPARISON AGAINST EYERISS FOR VGG-16 (ms).

Layer	Eyeriss	Phoenix	Speedup
CONV1-1	12.7	0.8	15.88×
CONV1-2	270.2	23.7	11.40×
CONV2-1	135.1	13.2	10.23×
CONV2-2	270.3	26.5	10.20×
CONV3-1	68.0	11.3	6.02×
CONV3-2	136.0	22.7	5.99×
CONV3-3	136.0	22.7	5.99×
CONV4-1	35.0	7.3	4.79×
CONV4-2	70.0	14.5	4.83×
CONV4-3	70.0	14.5	4.83×
CONV5-1	16.1	3.6	4.47×
CONV5-2	16.2	3.6	4.50×
CONV5-3	16.2	3.6	4.50×
Total	**1251.8**	**168**	**7.45×**

A lot of researchers concentrate on quantization to save storage. Binarization quantizes parameters into just two values, typically {-1, 1} with a scaling factor [25], [34], [40], [58]. Although binarization achieves remarkable energy and storage saving, it suffers from significant accuracy loss when binarizing both weights and activations [25], [34], [58]. Among them, XNOR_Net [40] can maintain comparable accuracy for AlexNet by only applying weights binarization. Ternary representations, which adds zero to the binary set, were introduced to help improve the accuracy [26], [35]. Logarithmic based quantization was proposed in [59]. Their results showed that 5-bit weights with log-base $\sqrt{2}$ and 5-bit activations with log-base 2 can achieve 2.72% top-1 accuracy loss and 1.71% top-5 accuracy loss for VGG-16. Quantization with 8-bit fixed-point is one general way to maintain low accuracy loss [17], [47]. The authors explored the relationship between the bit width and accuracy by quantizing the weights with floating-point and activations with fixed-point [20]. Intel [60] also tried to find out the relationship between the bit width and accuracy. Their experiments also included one floating-point quantization case along with other fixed-point cases. In [21], the authors explored dynamic floating-point based quantization with a calibration process to compensate for the accuracy loss in their work. However, all the aforementioned work failed to show promising results for *deep* CNNs such as ResNet152.

The approach in [18] showed that 8-bit fixed-point quantization was possible for *deep* CNNs. In [27], the authors quantized the weights with 5-bit and activations with 6-bit using their weighted-entropy-based quantization method for ResNet101, which achieved a small accuracy loss. A more aggressive method [28] provided promising results for *deep* CNNs. They quantized the small values of the weights into 4 bits while remained the rest 16 bits as full precision, by dividing the weights into the low-precision and high-precision regions according to the values of the weights. However, these work all need extra components to address the full precision weights. Different from all the above methods, our quantization methods fully exploit the distribution of weights and activations, thus obtaining a comparable or better accuracy for *deep* CNNs without any extra calibration, fine-tuning or re-training that need labelled data and extra computing. Access to labelled data could be difficult in practice as hardware and CNN algorithms are often developed by different parties.

CNN accelerators benefit a lot from the approximation algorithms with respect to improving energy efficiency and throughput. EIE [37], Cambricon-X [61] and Cambricon-S [62] are the ones that use weights reduction techniques. They build sparse matrix oriented architectures to accelerate CNNs after deep compression. However, they need re-training to compensate for accuracy loss. In addition, they need extra hardware components to address the irregularity caused by sparsity. FINN [41] and FP-BNN [63] are two binarization based accelerators. Although they can achieve higher energy efficiency and throughput than the full-precision counterparts, they both have a high accuracy loss. The accelerator in [59] was developed to speedup the quantized CNN with log-based weights and activations. Stripes [48] and Bit Fusion [39] performed layer-wise mixed-precision inference using bit-serial MACs. However, they failed to exploit *deep* CNNs. OLAccel was developed based on the V-Quant method [19], which used a large amount of 4-bit MACs plus a small portion of mixed-precision MAC units to cope with the high-precision region. This leads to significant energy saving compared with 8-bit fixed-point accelerators. However, their quantization method also need re-training to compensate for the quantization error. In addition, they keep a small portion of full-precision weights and activations during their quantization process, which leads to a hardware overhead to cope with the full-precision values. Overall, Phoenix is better in terms of performance and energy efficiency.

VIII. CONCLUSION

In this paper, we propose a normalization-oriented 8-bit floating-point quantization method which saves memory storage and memory access as well as maintaining negligible accuracy degradation (less than 0.5% for top-1 and 0.3% for top-5 accuracy). Due to the use of normalization, our quantization method gets rid of extra cost for calibration, fine-tuning or re-training to compensate for accuracy loss. We further design a hardware processor named Phoenix to fully leverage the benefits of our proposed quantization method and address the hardware inefficiency caused by floating-point operations. The key feature of Phoenix is the fully pipelined data-flow-based PE, which shares input activations and weights with other PEs, thus reducing inner bandwidth requirements. The circuit placement and routing results show that Phoenix can achieve peak performance of 2.048TMAC/s with $1.44mm^2$ and $1091.2mW$ at TSMC 28nm technology, respectively. Compared with a state-of-the-art accelerator, Phoenix achieves 3.32× and 7.45× better performance with the same core area for AlexNet and VGG-16, respectively. Compared with Nvidia TITAN Xp GPU, Phoenix consumes 151× less energy with single image inference.

REFERENCES

[1] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.

[2] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 779–788.

[3] D. Amodei, S. Ananthanarayanan, R. Anubhai, J. Bai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, Q. Cheng, G. Chen *et al.*, "Deep speech 2: End-to-end speech recognition in english and mandarin," in *International conference on machine learning*, 2016, pp. 173–182.

[4] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le *et al.*, "Large scale distributed deep networks," in *Advances in neural information processing systems*, 2012, pp. 1223–1231.

[5] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi *et al.*, "A configurable cloud-scale dnn processor for real-time ai," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*. IEEE Press, 2018, pp. 1–14.

[6] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *ACM Sigplan Notices*, vol. 49, no. 4, 2014, pp. 269–284.

[7] C. Zhang, G. Sun, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks," in *2016 IEEE/ACM International Conference on Computer-Aided Design*, 2016, pp. 1–8.

[8] X. Wei, C. H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang, and J. Cong, "Automated systolic array architecture synthesis for high throughput cnn inference on fpgas," in *Proceedings of the 54th Annual Design Automation Conference 2017*, 2017, p. 29.

[9] 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 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014, pp. 609–622.

[10] S. Liu, Z. Du, J. Tao, D. Han, T. Luo, Y. Xie, Y. Chen, and T. Chen, "Cambricon: An instruction set architecture for neural networks," in *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, 2016, pp. 393–405.

[11] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

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

[13] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[14] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2818–2826.

[15] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning," in *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[16] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 4700–4708.

[17] D. Lin, S. Talathi, and S. Annapureddy, "Fixed point quantization of deep convolutional networks," in *International Conference on Machine Learning*, 2016, pp. 2849–2858.

[18] S. Migacz, "8-bit inference with tensorrt," in *GPU Technology Conference*, 2017.

[19] E. Park, D. Kim, and S. Yoo, "Energy-efficient neural network accelerator based on outlier-aware low-precision computation," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture*, 2018, pp. 688–698.

[20] L. Lai, N. Suda, and V. Chandra, "Deep convolutional neural network inference with floating-point weights and fixed-point activations," *arXiv preprint arXiv:1703.03073*, 2017.

[21] S. O. Settle, M. Bollavaram, P. D'Alberto, E. Delaye, O. Fernandez, N. Fraser, A. Ng, A. Sirasao, and M. Wu, "Quantizing convolutional neural networks for low-power high-throughput inference engines," *arXiv preprint arXiv:1805.07941*, 2018.

[22] X. Lian, Z. Liu, Z. Song, J. Dai, W. Zhou, and X. Ji, "High-performance fpga-based cnn accelerator with block-floating-point arithmetic," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2019.

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

[24] 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," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture*, 2017, pp. 1–12.

[25] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," in *Advances in neural information processing systems*, 2015, pp. 3123–3131.

[26] Z. Lin, M. Courbariaux, R. Memisevic, and Y. Bengio, "Neural networks with few multiplications," *arXiv preprint arXiv:1510.03009*, 2015.

[27] E. Park, J. Ahn, and S. Yoo, "Weighted-entropy-based quantization for deep neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017, pp. 5456–5464.

[28] E. Park, S. Yoo, and P. Vajda, "Value-aware quantization for training and inference of neural networks," in *Proceedings of the European Conference on Computer Vision*, 2018, pp. 580–595.

[29] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2017.

[30] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[31] M. Courbariaux, Y. Bengio, and J.-P. David, "Training deep neural networks with low precision multiplications," *arXiv preprint arXiv:1412.7024*, 2014.

[32] N. Wang, J. Choi, D. Brand, C.-Y. Chen, and K. Gopalakrishnan, "Training deep neural networks with 8-bit floating point numbers," in *Advances in neural information processing systems*, 2018, pp. 7686–7695.

[33] S. Venkataramani, A. Ranjan, K. Roy, and A. Raghunathan, "Axnn: energy-efficient neuromorphic systems using approximate computing," in *Proceedings of the 2014 international symposium on Low power electronics and design*, 2014, pp. 27–32.

[34] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1," *arXiv preprint arXiv:1602.02830*, 2016.

[35] C. Zhu, S. Han, H. Mao, and W. J. Dally, "Trained ternary quantization," *arXiv preprint arXiv:1612.01064*, 2016.

[36] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: Shifting vision processing closer to the sensor," in *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3, 2015, pp. 92–104.

[37] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: efficient inference engine on compressed deep neural network," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture*, 2016, pp. 243–254.

[38] A. Boutros, S. Yazdanshenas, and V. Betz, "Embracing diversity: Enhanced dsp blocks for low-precision deep learning on fpgas," in *2018 28th International Conference on Field Programmable Logic and Applications*, 2018, pp. 35–357.

[39] H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, V. Chandra, and H. Esmaeilzadeh, "Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural networks," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, 2018, pp. 764–775.

[40] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *European Conference on Computer Vision*, 2016, pp. 525–542.

[41] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "Finn: A framework for fast, scalable binarized neural network inference," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 65–74.

[42] A. Prost-Boucle, A. Bourge, F. Pétrot, H. Alemdar, N. Caldwell, and V. Leroy, "Scalable high-performance architecture for convolutional ternary neural networks on fpga," in *2017 27th International Conference on Field Programmable Logic and Applications*, 2017, pp. 1–7.

[43] D. Zuras, M. Cowlishaw, A. Aiken, M. Applegate, D. Bailey, S. Bass, D. Bhandarkar, M. Bhat, D. Bindel, S. Boldo *et al.*, "Ieee standard for floating-point arithmetic," *IEEE Std 754-2008*, pp. 1–70, 2008.

[44] H. Sun and Y. Q. Shi, "Image and video compression for multimedia engineering: Fundamentals, algorithms, and standards," 2008.

[45] J. Max, "Quantizing for minimum distortion," *IRE Transactions on Information Theory*, vol. 6, no. 1, pp. 7–12, 1960.

[46] M. Paez and T. Glisson, "Minimum mean-squared-error quantization in speech pcm and dpcm systems," *IEEE Transactions on Communications*, vol. 20, no. 2, pp. 225–230, 1972.

[47] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 2704–2713.

[48] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos, "Stripes: Bit-serial deep neural network computing," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture*, 2016, pp. 1–12.

[49] J. Albericio, A. Delmás, P. Judd, S. Sharify, G. O'Leary, R. Genov, and A. Moshovos, "Bit-pragmatic deep neural network computing," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017, pp. 382–394.

[50] J. Redmon, "Darknet: Open source neural networks in c," http://pjreddie.com/darknet/, 2013–2016.

[51] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "Imagenet large scale visual recognition challenge," *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, 2015.

[52] D. Micron, "System power calculators," 2014. [Online]. Available: https://www.micron.com/support/tools-and-utilities/power-calc

[53] C. Nvidia, "Cublas library," *NVIDIA Corporation, Santa Clara, California*, vol. 15, no. 27, p. 31, 2008.

[54] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture*, 2016, pp. 367–379.

[55] E. Wang, J. J. Davis, R. Zhao, H.-C. Ng, X. Niu, W. Luk, P. Y. Cheung, and G. A. Constantinides, "Deep neural network approximation for custom hardware: Where we've been, where we're going," *arXiv preprint arXiv:1901.06955*, 2019.

[56] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Advances in neural information processing systems*, 2015, pp. 1135–1143.

[57] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[58] Z. Cai, X. He, J. Sun, and N. Vasconcelos, "Deep learning with low precision by half-wave gaussian quantization," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017, pp. 5918–5926.

[59] S. Vogel, M. Liang, A. Guntoro, W. Stechele, and G. Ascheid, "Efficient hardware acceleration of cnns using logarithmic data representation with arbitrary log-base," in *2018 IEEE/ACM International Conference on Computer-Aided Design*, 2018, pp. 1–8.

[60] G. R. Chiu, A. C. Ling, D. Capalija, A. Bitar, and M. S. Abdelfattah, "Flexibility: Fpgas and cad in deep learning acceleration," in *Proceedings of the 2018 International Symposium on Physical Design*, 2018, pp. 34–41.

[61] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-x: An accelerator for sparse neural networks," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture*, 2016, pp. 1–12.

[62] X. Zhou, Z. Du, Q. Guo, S. Liu, C. Liu, C. Wang, X. Zhou, L. Li, T. Chen, and Y. Chen, "Cambricon-s: Addressing irregularity in sparse neural networks through a cooperative software/hardware approach," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture*, 2018, pp. 15–28.

[63] S. Liang, S. Yin, L. Liu, W. Luk, and S. Wei, "Fp-bnn: Binarized neural network on fpga," *Neurocomputing*, vol. 275, pp. 1072–1086, 2018.