Low-precision Floating-point Arithmetic for High-performance FPGA-based CNN Acceleration

CHEN WU, MINGYU WANG, XINYUAN CHU, KUN WANG, and LEI HE,
Electrical and Computer Engineering, University of California, Los Angeles, USA

Low-precision data representation is important to reduce storage size and memory access for convolutional neural networks (CNNs). Yet, existing methods have two major limitations: (1) requiring re-training to maintain accuracy for *deep* CNNs and (2) needing 16-bit floating-point or 8-bit fixed-point for a good accuracy. In this article, we propose a low-precision (8-bit) floating-point (LPFP) quantization method for FPGA-based acceleration to overcome the above limitations. Without any re-training, LPFP finds an optimal 8-bit data representation with negligible top-1/top-5 accuracy loss (within 0.5%/0.3% in our experiments, respectively, and significantly better than existing methods for *deep* CNNs). Furthermore, we implement one 8-bit LPFP multiplication by one 4-bit multiply-adder and one 3-bit adder, and therefore implement *four* 8-bit LPFP multiplications using one DSP48E1 of Xilinx Kintex-7 family or DSP48E2 of Xilinx Ultrascale/Ultrascale+ family, whereas one DSP can implement only *two* 8-bit fixed-point multiplications. Experiments on six typical CNNs for inference show that on average, we improve throughput by 1.5× over existing FPGA accelerators. Particularly for VGG16 and YOLO, compared to six recent FPGA accelerators, we improve average throughput by 3.5× and 27.5× and average throughput per DSP by 4.1× and 5×, respectively.

CCS Concepts: • **Hardware** → **Hardware accelerators**; • **Computing methodologies** → *Neural networks;*

Additional Key Words and Phrases: Low-precision floating-point, CNN, deep learning, FPGA processor, FPGA acceleration

ACM Reference format:
Chen Wu, Mingyu Wang, Xinyuan Chu, Kun Wang, and Lei He. 2021. Low-precision Floating-point Arithmetic for High-performance FPGA-based CNN Acceleration. *ACM Trans. Reconfigurable Technol. Syst.* 15, 1, Article 6 (November 2021), 21 pages.
https://doi.org/10.1145/3474597

1 INTRODUCTION

Convolutional neural networks (CNNs) have demonstrated a breakthrough in performance for a broad range of applications including object recognition [46], object detection [39], and speech recognition [2]. However, CNNs often have huge computation complexity. This motivates accelerating CNNs by Central Processing Unit (CPU)/Graphics Processing Unit (GPU) clusters [7],

Authors' address: C. Wu, M. Wang, X. Chu, K. Wang, and L. He (corresponding author), Electrical and Computer Engineering, University of California, Los Angeles, Westwood, Los Angeles, CA, 90095; emails: {chenwu1989, mingyuw}@ucla.edu, xinyuansjtu@gmail.com, wangk@ucla.edu, lhe@ee.ucla.edu.

1936-7406/2021/11-ART6 $15.00
https://doi.org/10.1145/3474597

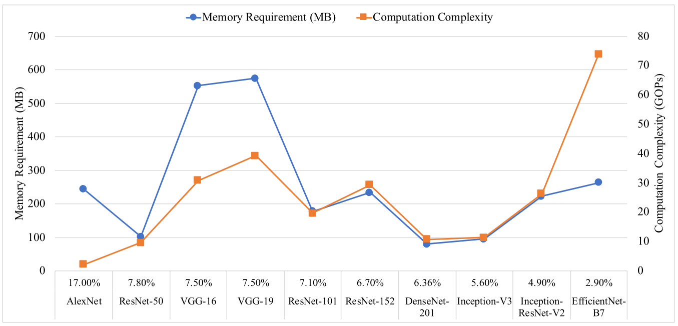


Fig. 1. Computation complexity and memory requirement with respect to different CNNs. The x-axis indicates the name and the top-5 classification error under ImageNet dataset of the CNN.

Filed Programmable Gate Arrays (FPGAs) [20, 33], and **Application Specific Integrated Circuits (ASICs)** [3]. Customized accelerators/processors on FPGAs have shown more promising throughput and power efficiency than traditional CPU/GPU clusters [52, 60].

Larger and deeper CNNs have been developed to improve performance for a broader range of scenarios. For example, the top-5 error for ImageNet [41] classification decreases from 17% to 2.9%. However, computation complexity and number of parameters increase dramatically as depicted in Figure 1. To be specific, the computation complexity of a feed-forward process of a 224 × 224 RGB image increases from 2.27 **Giga Operations (GOPs)** of AlexNet in 2012 to 74 GOPs of EfficientNet-B7 in 2019. At the same time, the number of parameters stays large at 264 MB. Such great computation complexity makes it harder for general-purpose processor to meet the requirements of real-time applications. However, the great quantities of parameters lead to a big challenge for communication between off-chip and on-chip memories because of bandwidth constraints.

There are two types of research to reduce computation and parameter complexities for CNN inference. The first one is deep compression including weight pruning, weight quantization, and compression storage [11, 12]. However, irregularity caused by deep compression degrades parallelism and hardware performance. Cambricon-S [64] alleviates irregularity in sparse neural networks through a software/hardware co-design approach to improve hardware performance. However, all the above accelerators need a time-consuming retraining process to maintain accuracy.

The second type of research is more efficient data representation, also known as quantization for circuit implementation. Reference [30] used 16-bit floating-point in contrast to the 32-bit commonly used for computing. However, one 16-bit floating-point multiplier on FPGA needs 1 DSP, 85 LUTs and 167 FFs when using Xilinx floating-point IP [25] as shown in Table 1, leading to a low hardware efficiency. Since one 16-bit or smaller fixed-point multiplier can be fit into one DSP, both 16-bit [29, 56] and 8-bit [57–59] fixed-point were employed to gain more hardware efficiency than 16-bit floating-point does. The 8-bit floating-point is also tried in both FPGA and ASIC [53, 54]. Another 8-bit arithmetic, called **block floating-point (BFP)**, is also applied [23, 45], where a parameter has its own mantissa but shares a same exponent for one data block. Reference [22] proposed a mixed data representation with floating-point for weights and fixed-point for activations (e.g., outputs of a layer). Reference [42] developed an 8-bit floating-point quantization scheme, which needs an extra inference batch to compensate for the quantization error. However, Reference [22] and Reference [42] did not present a circuit design for their approaches. While all aforementioned work has a good accuracy with retraining, more aggressive data representations such as binary [6], ternary [24], and mixed precision (2-bit activations and ternary weights) [5] may suffer from great accuracy loss even with time-consuming retraining.

Table 1. Resource Utilization of Multipliers on FPGA for Different Data Representations

Data Representation	DSP	LUT	FF
one 16-bit floating multiplication	1	85	167
one 16-bit fixed multiplication	1	0	0
two 8-bit fixed multiplications	1	2	0
four 8-bit floating ($M4E3$) multiplications	1	20	27

DSP: digital signal processing, LUT: look-up table, FF: flip-flop. $M4E3$: 1-bit sign, 4-bit mantissa and 3-bit exponent.

In this article, we first propose a **low-precision floating-point (LPFP)** to quantize both weights and activations. During the quantization process, an optimal LPFP data format and the corresponding scale factor are decided for a workload of CNNs. 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 V-Quant [35] that works for such *deep* CNNs has a top-1 accuracy loss about 1% with fine-tuning. Then, we design an LPFP-based FPGA processor to further improve the performance for CNN inference. We are able to implement four 8-bit floating-point multiplications within one DSP (see Table 1). We experimented for inference of AlexNet, VGG16 [43], ResNet50/101/152 [13], and DenseNet201 [14] via Xilinx KC705 and Xilinx ZCU106. We can achieve an average throughput of 1100.4 **Giga-Operations Per Second (GOPS)** and 1.43 GOPS per DSP on KC705. On ZCU106, the average throughput and per DSP throughput are 1650.6 GOPS and 2.15 GOPS, respectively. Moreover, the average throughput for these networks is 82.3% and 1.5× over Intel I7-8700T CPU and existing accelerators, respectively. Compared with six existing accelerators for VGG16 and YOLO, on average, our processor improves throughput by 3.5× and 27.5×, while improving per DSP throughput by 4.1× and 5×, respectively. To the best of our knowledge, this is the first work that can fit four 8-bit multiplications for inference in one DSP while maintaining comparable accuracy without any retraining.

The main contributions of this article are listed as follows:

- The *non-uniform quantization method with low-precision float-point data format* are used to quantize the input activations and weights for CNNs. The optimal data format, which achieves negligible accuracy loss, can be selected automatically without any fine-tuning, calibration or retraining.
- The previous work with 8-bit fixed-point quantization converted the original 32-bit floating point multiplication into 8-bit fixed-point multiplication, and one DSP slice can only be implemented to perform *two* 8-bit fixed-point multiplication. However, LPFP quantization converts the original 32-bit floating point multiplication into 8-bit floating-point multiplication, *four* of which can be implemented inside one DSP slice. Thus, 2× number of multipliers can be implemented under the same resource constraints for the same FPGA. Note that the computational throughout mainly comes from the number of multipliers, and our approach can achieve 2× computational throughput compared with previous work with 8-bit fixed-point quantization methods.

2 BACKGROUND AND MOTIVATION

2.1 Background

2.1.1 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, and the connections are used to 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: (1) *slim* for less than 50 layers, (2) *medium* for 50 to 100 layers, and (3) *deep* for more than 100 layers.

2.1.2 Low-precision Floating-point. Similarly to the definition of 32-bit floating-point from the IEEE-754 standard [65], the binary representation of LPFP number comprises *sign, mantissa*, and *exponent* in order. The decimal value of LPFP number is then calculated by

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

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

$$E_b = 2^{DW_E-1} - 1, \tag{2}$$

where DW_E is the data width of E. Different from the IEEE Standard, data widths for M and E in this article are not fixed. In later sections, we use the term $MaEb$ to indicate different combinations, where a and b indicate the bit width of M and E, respectively. For example, $M3E4$ means the mantissa is 3 bits while the exponent is 4 bits.

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

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

Note that Infinity and Not a Number are the other two special cases but are not used in our work. This is because our saturation scheme saturates large numbers to the maximal number, as illustrated in detail in Section 3.1.

2.2 Motivation

CNN accelerators with lower data width have significant improvements in terms of memory size, memory bandwidth, and power efficiency. Due to the lack of floating-point arithmetic units in FPGAs, researchers have used low-precision fixed-point instead of floating-point. A 16-bit fixed-point quantization to find the best scale factor for each layer was proposed in Reference [56]. However, this required time-consuming retraining to amend the weights to maintain accuracy. Furthermore, a model was developed to quantitatively analyze the convolution loops and optimize design objectives such as memory access and latency [29]. However, it had an accuracy loss as large as 2%. A shared drawback for the above two approaches is the low per DSP throughput (0.279 GOPS/DSP in Reference [56] and 0.472 GOPS/DSP in Reference [29]) because of using 16-bit multiplication.

An 8-bit fixed-point accelerator was designed in Reference [9] for embedded FPGAs, with a low per DSP throughput of 0.444 GOPS/DSP. DNNBuilder [62] aimed to automatically build high-performance DNN hardware accelerators for both cloud and edge-FPGAs with 8-bit fixed-point quantization. It increased the per DSP throughput to 0.771 GOPS by better architecture exploration; however, its quantization method incurred 4.6% top-1 accuracy degradation without fine-tuning. FPGA accelerators with the aforementioned BFP arithmetic [23] had a per DSP throughput of 0.741 GOPS. However, only *slim* and *medium* CNNs were validated in their approach. Approaches in the industry focus on improving the accuracy with 8-bit fixed-point data representation [15, 19, 31].

However, even if we use the quantization policies from these studies, we will still suffer from the low per DSP throughput as one DSP can only be decomposed to two 8-bit fixed-point multiplications. In short, existing approaches cannot improve the per DSP throughput while maintaining comparable accuracy for all *slim, medium*, and *deep* CNNs.

3 LOW-PRECISION FLOATING-POINT QUANTIZATION

In this section, we present the details of our proposed LPFP quantization method, including the quantization process, dataflow in the processor and quantization results.

3.1 Quantization Process

The quantization process is divided into two steps: (1) fusing operations and (2) finding scaling factor and quantizing data. We will discuss these in detail in this section.

3.1.1 Fusing Layers. In each CNN, we define the convolutional/fully-connected layers as key layers while others as subordinate layers. During the quantization process, we try to merge as many subordinate layers into the previous key layers as possible to simplify the design. This is because most of the subordinate layers will not change the data precision during calculation. For example, batch normalization layer is linear and can be merged in advance (see details in Appendix); max pooling layer only does comparisons and will not influence the precision. Moreover, for the subordinate layers that cannot be merged into the key layers (i.e., average pooling), we treat them as a separate layer and will do quantization in the next step.

3.1.2 Finding Scaling Factor and Quantizing Data. The proposed LFPF quantization method is applied to the output activations and weights of each fused layers. The quantization function is defined as follows:

$$V_{lfp} = quan(V_{fp32} \times 2^{sf}, MIN_{lfp}, MAX_{lfp}), \tag{4}$$

where V_{lfp} and V_{fp32} denote the decimal values represented by LPFP and traditional single floating-point format, respectively; MIN_{lfp} and MAX_{lfp} indicate the minimal and maximal numbers represented by LPFP, and sf is the scaling factor that is used to better fit the data into the dynamic range of LPFP. After finding the optimal scaling factor and quantizing the activations and weights of each layer, the scaling factor needs to be re-normalized for accuracy. To simplify the calculation in the processor, we do the re-normalization of the scaling factor when we re-quantize the output activations to LPFP format in the data conversion step (detailed illustrated in the section dataflow in processor). The *quan* function in Equation (4) rounds the data to the nearest value with saturation considered, formulated as

$$quan(x, MIN, MAX) = \begin{cases} MIN & x <= MIN \\ MAX & x >= MAX \\ round(x) & \text{otherwise} \end{cases}, \tag{5}$$

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

The **mean square error (MSE)** of the values before and after quantization is used as the metric to evaluate the quantization error, illustrated as

$$MSE = \frac{1}{N}\sum_{i=0}^{N}(V_{lfp}/2^{sf} - V_{fp32})^2, \tag{6}$$

where N denotes the amount of data.

As illustrated from Equations (4) to (6), MSE is influenced by the data format of LPFP and the scaling factor (sf). Since the quantization process is performed offline only once for each CNN, we

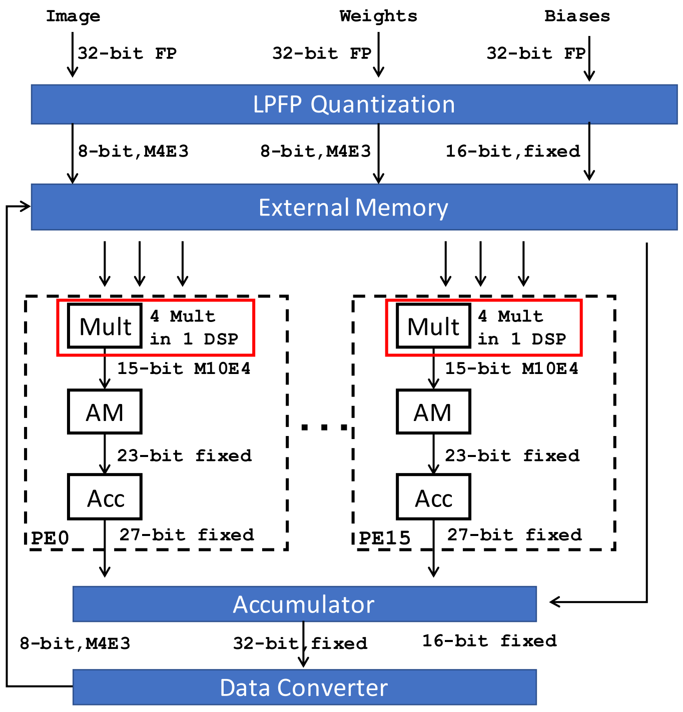


Fig. 2. The dataflow in our processor with *M4E3* data format as an example (FP: floating-point; Mult: LPFP multiplier; AM: alignment module; Acc: accumulator; DC: data converter).

use exhaustive search to find the optimal combination of LPFP data format and scaling factors for both weights and activations. During the inference process for each CNN, the quantized weights and scaling factors of activations are fixed for different test images. Therefore, no extra computation is needed during the inference process. In this article, we assume *the same data format for a CNN and a same scaling factor for each layer*. This assumption can be removed as needed. Furthermore, we choose to use the same optimized data format for all test cases in our experiments, while the problem formulation is to decide a data format for each CNN.

3.2 Dataflow in Processor

The dataflow of our proposed approach is shown in Figure 2. To explicitly illustrate the dataflow, we list the bit width in each step with *M4E3* data format as an example. The weights and biases of the pre-trained model are represented by 32-bit floating-point. The weights are first quantized into *M4E3* while the biases are quantized into 16-bit fixed-point to reduce quantization error. All the quantized weights and biases are then stored into the external memory of the FPGA board. The quantization of weights and biases are performed on a server only once for each CNN. In our processor, the raw input image that indicates the input of the first layer is also quantized from 32-bit floating point into *M4E3* and stored into the external memory.

During the inference on our FPGA processor, the quantized image, weights and biases are fetched from the external memory, and the multiplications of image and weights are performed with *M4E3* data format. The *M4E3* multiplication is decomposed into three parts: (1) xor of the sign bit, (2) multiplication of the mantissa, and (3) addition of the exponent. To maintain full precision during computation, the results of the LPFP multiplier are kept 15-bit, with 1 sign bit, 10

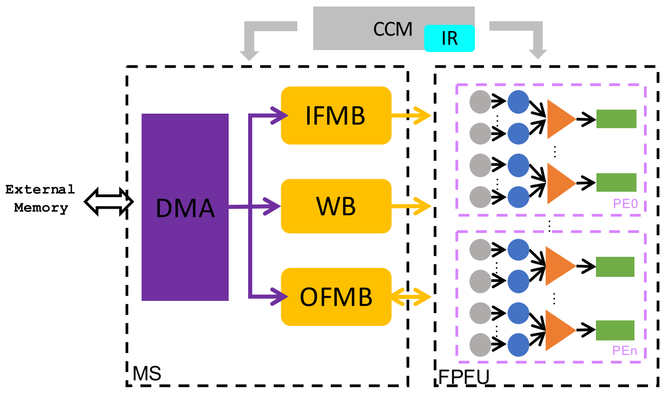


Fig. 3. The overall architecture of proposed processor (DMA: direct memory access; IFMB: input feature map buffer; WB: weight buffer; OFMB: output feature map buffer; MS: memory system; FPFU: floating-point function unit; CCM: central control module, IR: instruction ram).

mantissa bits, and 4 exponent bits. The followed **align module (AM)** converts the 15-bit products to 23-bit fixed-point without any precision loss. Considering the exponent of the product can be varied from −4 to 8, we can use 12 more bits to cover all the 13 cases (12 bits can have 13 dot positions). Moreover, we set the 23-bit fixed-point to have 12 decimal places by considering the worst case (4 bits from the worst case of exponent to be −4 and 8 bits from the 10-bit mantissa). In this way, all the accumulation can be done in 32-bit fixed-point accumulators with saturation, which consumes fewer resources in FPGA than floating-point accumulators. Since the fixed-point accumulator does not change the dot position, the final outputs of the accumulators still have 12 decimal places. Finally, we quantize the 32-bit fixed-point data back to $M4E3$ floating-point and store them in the external memory before being used by another CNN layer.

As the scaling factors of the input activations and kernels are propagated to the output activations during convolution, we need to re-normalize the output activations for accuracy by multiplying $2^{sf_{oa}-sf_{ia}-sf_k}$, where sf_{oa}, sf_{ia}, and sf_k indicate the scaling factors of output activations, input activations, and kernels. This is done by shifting the 32-bit fixed-point data in the data conversion step. Moreover, to simplify the data conversion, we will first quantize the shifted data to 16-bit fixed-point with 1 sign bit, 7 integer bits, and 8 decimal bits. After that, the 16-bit fixed-point data is quantized to 8-bit LPFP data format. In the whole dataflow, only the final data conversion step introduces bit truncation and precision loss. However, the precision loss introduced by the final step has little impact on the final accuracy and is validated in Section 5.1.2 with comprehensive experimental results.

4 PROCESSOR ARCHITECTURE

In this section, we discuss in detail the architecture of the processor, which efficiently supports the inference process of quantized networks for various CNNs.

4.1 Overview

The overall architecture of the proposed processor is depicted in Figure 3. A **floating-point function unit (FPFU)**, which is composed of multiple **processing elements (PEs)**, is developed to compute the outputs of a layer in parallel. The PE, which is the key component of FPFU, is designed to efficiently perform dot product with LPFP data format. The on-chip **memory system (MS)** consists of three buffers, e.g., **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 memories through **direct memory access (DMA)** module. The **central control module (CCM)** is designed to arbitrate between different modules. Moreover, the CCM decodes various instructions stored in the **instruction RAM (IR)** into detailed signals for other modules.

4.2 Floating-point Function Unit

FPFU, which is constructed by multiple PEs, is designed to perform convolution in LPFP data format efficiently for performance gain and power reduction. Different parallel computation patterns, including parallel in input feature maps, parallel in output feature maps, and parallel in both input and output feature maps, are developed in FPFU and are discussed in the following paragraphs. FPFU receives activations and weights from IFMB and WB, respectively, and distributes the activations and weights to different PEs to perform convolution according to the control signals decoded by CCM.

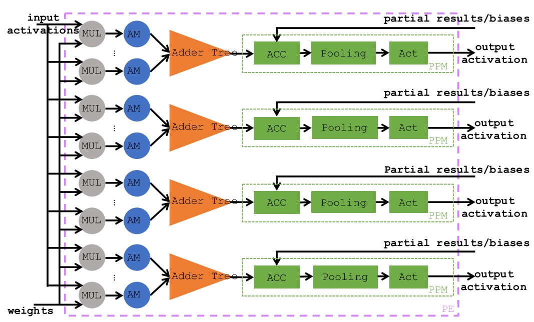
4.2.1 Architecture of PE. The PE is designed as a fully pipelined dataflow-based architecture, as shown in Figure 4(a). Once a PE receives the activation and weight vector, which are represented with $M4E3$ data format, it distributes the data to N_m multipliers inside the PE. The products of the multipliers keep full precision and are transferred into the AM. The full precision products are aligned and converted to fixed-point numbers without any bit truncation. The aligned products are then fed into four fixed-point adder trees to finalize four dot product processes in parallel, which indicates the feed-forward process of four pixels in two output channels (see details in Section 4.2.2). The accumulation of partial results (including bias), pooling and activation processes are performed in series inside the post process module.

The multipliers in each PE are developed for LPFP, which are represented with scientific notation in the sign-and-magnitude format, as illustrated in Equations (1) and (3). The multiplication of two LPFP numbers is then divided into three fixed-point components: (1) XOR of the signs, (2) multiplication of mantissas, and (3) addition of exponents. Take the $MaEb$ format as an example. An a-bit unsigned MAC and a b-bit unsigned adder are needed. Considering the first hidden bit of mantissas—“1” for normal numbers and “0” for subnormal numbers—we design an a-bit multiplier and an $a + 2$-bit adder to perform the $a + 1$-bit multiplication. The a-bit multipliers and $a + 2$-bit adders are implemented within one DSP slice to improve per DSP throughput (see details in Section 5.2.2). Meanwhile, the exponent bias E_b is not included during addition, because the E_b is the same for all the numbers in one CNN as we assume, and we can address this at the last step to simplify the adders.

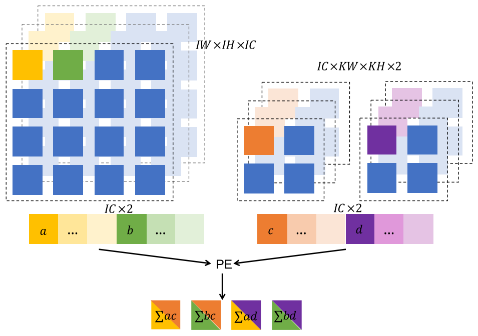
4.2.2 Parallel Computation Pattern. The convolutional/fully-connected layers are calculated in the PE. For the fully-connected layers, we treat the input feature maps as “weights” and weights as “input feature maps” to increase the data reuse and make the calculation pattern the same as convolutional layers. Therefore, each pixel in one output channel for convolutional/fully-connected layers is calculated as

$$y_i = \sum_{k=0}^{KW\times KH} \sum_{ic=0}^{IC} x_{k,ic} w_{k,ic} + b_i, \tag{7}$$

where IC indicates the number of input channel, KW and KH denotes the width and height of the kernel, and x, y, w, and b are input activation, output activation, weight, and bias, respectively. In our implementation on FPGA, we implement 4 LPFP multipliers with one DSP slice, which follows the pattern: $(a + b) \times (c + d) = ac + bc + ad + bd$ (see details in Section 5.2.2). Therefore, each PE is designed to process convolution in two output channels in parallel, and in each output



(a) The architecture of a PE. MUL: LPFP multiplier, AM: alignment module, ACC: accumulator, Act: activation.



(b) Parallel computation pattern in one PE.

Fig. 4. The architecture of a PE and the parallel computation pattern in a PE.

channel, it will calculate the convolutional results of two pixels at the same time, as shown in Figure 4(b). To be specific, in the first cycle, the first pixel in IC input channels and the first value in the corresponding kernels are fed into the PE, marked with a and c in Figure 4(b), respectively. To follow the computation pattern in these four multipliers, the second pixel in IC input channels (marked with b), and the corresponding kernels to calculate the pixel in another output channel (marked with d) are also fed into the PE. In this way, a and b are reused to produce the pixels in different output channels, while c and d are reused to produce the pixels in different positions of the same output channel. After $KW \times KH$ cycles, four convolution results are produced by one PE.

As illustrated in Section 4.2.1, N_m multipliers are used in each PE, and IC is designed to be $N_m/4$. In this way, $N_m/4$ input channels are calculated in parallel in each PE. With the corresponding weights and biases, 2 pixels in 2 output channels are calculated in parallel. When the number of input channels is larger than $N_m/4$ and/or when the number of pixels in each output channel is larger than 2 and/or when the number of output channels is larger than 2, multiple rounds of computation are needed in series to finalize the convolution. To further increase the parallelism, we use N_p PEs in the FPFU. In different PEs, we can feed in different pixels in input feature maps and weights to perform different parallel computation pattern. For example, the N_p PEs can share the same input feature map and use different weights to parallelize the computation in output channels, or the N_p PEs can share the same weights and use different input feature maps to parallelize the computation in input channels. The N_m, N_p and the parallel computation pattern are decided by considering the CNNs, the throughput and the bandwidth requirement. This will be explained with experiments in Section 5.2.2.

4.3 Memory System

To keep the PE working without waiting for the data to be ready, the bandwidth of IFMB and WB for each PE are designed to be $N_m/2$ LPFP input activations and weights per cycle, respectively, while the OFMB is 4 output activations per cycle. Although each pixel in the output feature map is represented with LPFP data format, we keep the intermediate results with 16-bit precision to reduce accuracy loss. In this way, the bandwidth of OFMB for each PE is set to 64 bits per cycle. As the input activations and/or weights can be shared by different PEs according to different computation patterns, we define P_{ifm} and P_{ofm} ($P_{ifm} \times P_{ofm} = N_p$) to indicate the parallelisms in input feature map and output feature map, respectively. In this definition, P_{ifm} indicates that we have P_{ifm} PE groups where the same weights are shared during calculation, while in each PE group, P_{ofm} PEs share the same input activations. In conclusion, the bandwidth for IFMB, WB, and

OFMB are $N_m/2 \times P_{ifm} \times BW$, $N_m/2 \times P_{ofm} \times BW$, and $64N_p$ per cycle, respectively, where BW denotes the bit width of LPFP data format.

As the amount of on-chip buffers are limited, we stored all the feature maps, weights, and biases in the off-chip memory and preload the feature maps, weights and biases needed by a computation block in the on-chip memories before computation. In this way, the parameters N_m, P_{ifm}, and P_{ofm} are decided to tradeoff between the throughput, bandwidth requirement, and resource utilization. Previous proposed work applied large enough buffers to store all the activations or weights for one layer [10] to avoid costly off-chip memory access. However, such designs incurred large area and unscalability for larger and deeper CNNs. In our processor, we tradeoff among the throughput, bandwidth requirement, resource utilization and scalability, and employ the smallest sizes that can hide the DMA communication time. In our implementation on FPGA, we use block RAM to deploy IFMB and OFMB, while we use distributed RAM to deploy WB, as distributed RAM can provide higher bandwidth than block RAM. During inference on our processor, only when all the input feature maps 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.

4.4 Central Control

The CCM is designed to arbitrate among different modules and control the whole execution process. First, CCM decodes the instructions from IR efficiently and sets the corresponding control registers. Second, different modules are activated according to the control registers and the status of each module is 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.

5 EVALUATION

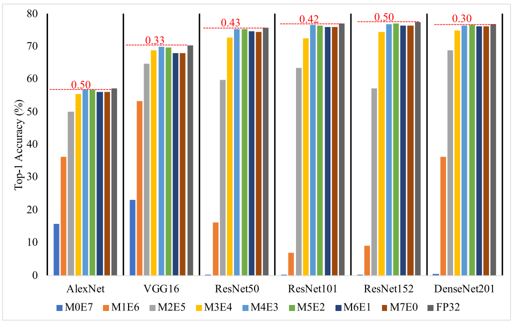
In this section, the evaluation of the proposed quantization method is first provided, and then the implementation details and comprehensive experimental results are provided.

5.1 Evaluation of Quantization Method

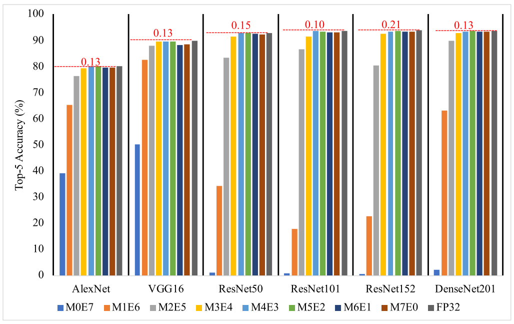
5.1.1 Experiment Setup. We implement our LPFP quantization method with C language based on the Darknet framework [38], and the inference process of the quantized network follows the same dataflow as that in our processor illustrated in Figure 2. The validation accuracy with single center-crop is then evaluated via the ImageNet validation set (50,000 labelled images) [41]. Our quantization process is run on an Intel Core I7-8700T CPU working under 2.40 GHz, while the evaluation process is run on a Nvidia TITAN Xp GPU. During the evaluation process on GPU, all the quantized data are converted to 32-bit floating without any precision loss. In addition, the computation on GPU are based on 32-bit floating point. Six representative CNNs (AlexNet, VGG16, ResNet50/101/152, and DenseNet201) including the *slim, medium*, and *deep* CNNs are evaluated.

5.1.2 8-bit Quantization. The detailed validation accuracies on the quantized networks with 8-bit floating-point data format are shown in Figure 5(a) and 5(b). We emulate all 8 different (mantissa, exponent) combinations to validate the top-1 and top-5 accuracy of the quantized CNNs, and the 32-bit floating-point results are included as the baseline.

In Figure 5(a) and 5(b), the dashed lines illustrate the 32-bit floating-point baseline where the values above the dashed lines are the accuracy loss compared with the baseline. We can see that our LPFP quantization approach can maintain comparable top-1 and top-5 accuracy 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



(a) Top-1 accuracy



(b) Top-5 accuracy

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

Table 2. Accuracy Comparison among *M4E3*, *M5E2*, References, and FP32

	Top-1 Accuracy (%)	Top-5 Accuracy (%) for each network										
	AlexNet		VGG16		ResNet50		ResNet101		ResNet152		DenseNet201	
Nvidia [31]	0.03	−0.01	0.03	—	0.13	0.12	−0.01	0.06	0.08	0.05	—	—
BFP [23]	—	—	0.03	0.02	0.11	0.12	—	—	—	—	—	—
Ours (M4E3)	*0.58*	*0.19*	*0.33*	*0.13*	*0.55*	*0.15*	*0.42*	*0.10*	*0.81*	*0.39*	*0.45*	*0.19*
Ours (M5E2)	*0.48*	*0.13*	*0.64*	*0.32*	*0.43*	*0.19*	*0.67*	*0.37*	*0.50*	*0.21*	*0.30*	*0.13*
Ours (M4E3) R	*0.03*	*0.01*	*0.03*	*0.00*	*0.08*	*0.06*	*0.05*	*0.02*	*0.04*	*0.01*	*0.07*	*0.03*
Ours (M5E2) R	*0.04*	*0.00*	*0.02*	*0.00*	*0.05*	*0.03*	*0.06*	*0.04*	*0.03*	*0.02*	*0.03*	*0.01*

"—" means no reported results; "R" means the method with retraining.

results, respectively. Particularly, *M5E2* always achieves the highest accuracy compared with the other cases. Data formats with more than or equal to 3-bit mantissa all have a low accuracy loss for all the six CNNs, while those 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 1-bit sign, 7-bit mantissa, and no exponent, exactly fixed-point). As shown in Figure 5(a) and 5(b), *M4E3* and *M5E2* outperform the fixed-point for all six benchmarks.

5.1.3 Comparison with the Prior Quantization Strategies.

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 losses for all six benchmarks in Table 2, where "—" indicates no reported results in the literatures. We also retrain our quantized network using *M4E3* and *M5E2* for 10 epochs with the original training data, and the Top-1 and Top-5 accuracy loss are also included in Table 2. Although the top-1 and top-5 accuracy losses achieved by our LPFP quantization method without retraining are not the best among all the literatures, our method without retraining is suitable for fast deployment and does not need any training data in the real-world. Moreover, after retraining, our LPFP quantization method can regain the validation accuracy compared with the 32-bit floating point networks. Moreover, our method can reach *deep* networks.

5.1.4 Lower Bit Width Quantization.

We further reduce the bit width from 8-bit to 4-bit and also evaluate the top-1 and top-5 accuracy of the quantized networks. We pick the best (mantissa, exponent) combination for each data format and the results are shown in Figure 6. We can see that both the top-1 and top-5 accuracy decrease when lower bit length is utilized to represent the weights and activations of CNNs. Particularly, the average top-5 accuracy degradations for 7-bit

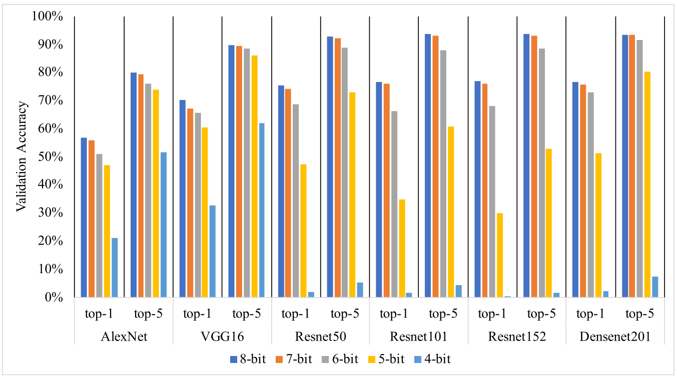


Fig. 6. Top-1 and top-5 accuracies for different bit width with respect to different CNNs.

and 6-bit are 0.8% and 4.2%, respectively. However, the accuracy drops dramatically when the bit width decreases to less than 6 bits, which means our LPFP quantization approach can hardly find accurate results without any retraining process.

5.2 Evaluation of Hardware Implementation

5.2.1 Environment Setup. Our processor is implemented on the KC705 evaluation board with a Xilinx Kintex-7 XC7K325T FPGA and ZCU106 evaluation board with a Xilinx Zynq UltraScale+ XCZU7EV FPGA. First, we explore the parallel computation patterns to find the optimal parameters to best fit the two FPGAs. Second, with these parameters, the processor is described in Verilog-HDL, and synthesized and implemented with the Xilinx Vivado 2018.2 Design Suite. Finally, we evaluate the throughput, inference latency and per DSP throughput of running different networks on our processor, and the results are compared with two prior accelerators [27, 29]. The Intel Core I7-8700T CPU under 2.40 GHz working frequency and the Nvidia Xavier NX GPU with a 8 GB LPDDR4 are also used for comparison. More comprehensive experimental results on VGG16 and YOLO are compared with latest FPGA accelerators [9, 23, 27, 29, 30, 48, 56].

5.2.2 Implementation Details. We use the $M4E3$ data format for FPGA implementation in this article for two reasons. First, $M4E3$ achieves the top two best validation accuracies among all the LPFP (mantissa, exponent) combinations we tested (see Section 5.1). Particularly, the average top-1 and top-5 accuracy loss of $M4E3$ compared with 32-bit floating-point are 0.53% and 0.19%, respectively. Second, $M4E3$ only needs a 4-bit fixed-point MAC and a 3-bit fixed-point adder, resulting in fewer resources on FPGA than $M5E2$. To be specific, four 4-bit fixed-point MACs can be implemented inside one DSP48E1 slice in XC7K325T FPGA.

To clearly explain the way to implement four MACs with one DSP48E1 slice, we take the multiplication of two normal numbers (X and Y) as an example. The mantissa of the product can be explained as

$$\begin{aligned} Prod &= 1.M_x \times 2^{E_x - E_b} \times 1.M_y \times 2^{E_y - E_b} \\ &= (0.M_x \times 0.M_y + (1.M_x + 0.M_y)) \times 2^{E_x + E_y - 2E_b}, \end{aligned} \tag{8}$$

where M_x, M_y, E_x, and E_y are the mantissas and exponents of X and Y, respectively. In Equation (8), the term $0.M_x \times 0.M_y + (1.M_x + 0.M_y)$ is performed with a 4-bit unsigned fixed-point MAC and the term $E_x + E_y$ is performed with an extra 3-bit unsigned fixed-point adder. As the DSP48E1 slice can be implemented as a MAC followed by $P = A \times B + C$ (where the maximal bit width

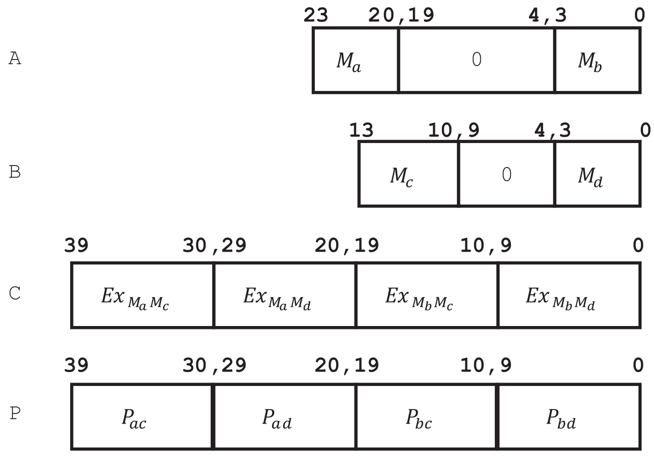
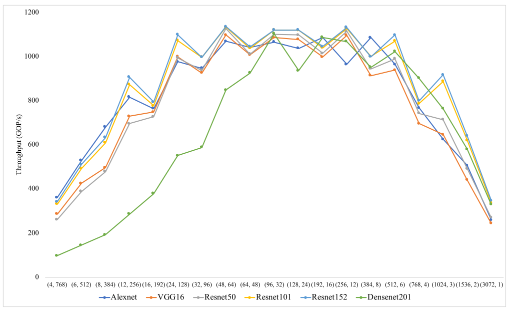


Fig. 7. Data format of the DSP to implement four 4-bit MACs. M_a, M_b, M_c, and M_d: the mantissas of LPFP data a, b, c, and d, respectively; $Ex_{M_aM_c}, Ex_{M_aM_d}, Ex_{M_bM_c}$, and $Ex_{M_bM_d}$: the extra term expressed as $Ex_{M_aM_c} = 1.M_a + 0.M_c$; P_{ac}, P_{ad}, P_{bc}, and P_{bd}: the mantissas of the product of two LPFP data expressed as $P_{ac} = 1.M_a \times 1.M_c$.

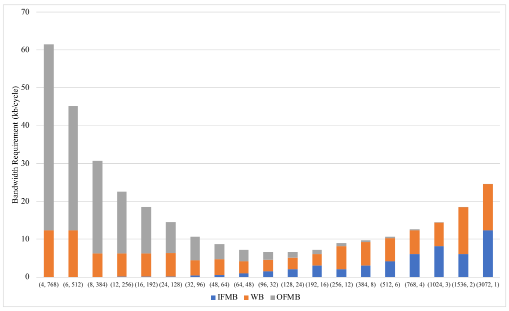
of A, B, and C are 25, 18, and 48, respectively), we add blank bits to the three inputs to fully utilize the functionality of DSP48E1, as shown in Figure 7. This proposed method also works for the next generation of Xilinx DSP slice (DSP48E2), which can also be configured as a MAC, with the maximal bit width of A, B, and C to be 27, 18, and 48. During the calculation process, the dot position is kept at the right most position. That is, the terms $0.M_x$ and $0.M_y$ are converted to 4-bit integers, while the extra term $1.M_x + 0.M_y$ is converted to 10-bit integers to make sure that no overlap occurs. In this way, with a few LUTs and FFs to perform additions of the exponents and the extra term $1.M_x + 0.M_y$, four multiplications with $M4E3$ data format can be carried out in on DSP slice (see Table 1), thus dramatically increasing the per DSP throughput.

Parallel Exploration. Since one DSP slice is divided into four 4-bit LPFP MACs in our implementation, the parameters should meet the requirement that $N_m \times N_p = 4 \times \#of DSP$. Considering the resources of XC7K325T FPGA, we set the targeted number of DSP as 768, which accounts for 91.43% of the available DSPs. We then evaluate the throughput for different CNNs and the bandwidth requirement with respect to different N_m and N_p combinations as shown in Figure 8(a) and 8(b), respectively. We also explore different combinations of the parameters P_{ifm} and P_{ofm}, and only depict the P_{ifm} and P_{ofm} for achieving the optimal throughput and minimal bandwidth requirement in Figure 8(a) and 8(b).

In general, when N_m keeps increasing, the throughput first increases and then decreases when it reaches the peak. The small N_m and large N_p indicate that more output channels are calculated in parallel while large N_m and small N_p mean more input channels are calculated in parallel. When N_m is larger than the total number of input channel (denoted as IC), only IC multipliers are used while the rest are wasted, resulting in a low throughput. This is the same for large N_p, and the peak throughput comes from balanced N_m and N_p. For different CNNs, the peak throughput comes from different N_m and N_p combinations due to different network configurations. For example, DenseNet201 has lots of inception layers, which concatenate layers with small output channels (e.g., 32) to form layers with large input channels (e.g., 1,568). In this case, larger N_m and smaller N_p incur fewer wasted computations and lead to higher throughput. From Figure 8(a), we can see that the combination of $N_m = 96$ and $N_p = 32$ results in an optimal throughput for all cases on average.



(a) Throughput for different CNNs with respect to different N_m and N_p combinations.



(b) Bandwidth requirement with respect to different N_m and N_p combinations.

Fig. 8. Parallel exploration with respect to throughput and bandwidth requirement.

Table 3. Resource Utilization in XC7K325T

Resource	LUT	LUTRAM	FF	BRAM	DSP
Used	154625	7860	180561	234.5	768
Available	203800	64000	407600	445	840
Utilization	75.87%	12.28%	44.30%	52.70%	91.43%

Table 4. Resource Utilization on Ultrascale+ 7EV

Resource	LUT	LUTRAM	FF	BRAM	URAM	DSP
Used	133517	15760	201465	84.5	48	768
Available	230400	101760	460800	312	96	1728
Utilization	57.95%	15.48%	43.72%	27.08%	50%	44.5%

The bandwidth requirement is extremely high when N_p is large. This is because larger N_p indicates more parallel computations in output channels. Moreover, OFMB is designed to store 16-bit intermediate results, which also lead to higher bandwidth requirement with larger N_p. The total bandwidth requirement decreases when N_p decreases, and then increases again, since larger N_m needs more bandwidth to load input activations and weights. The smallest bandwidth requirement comes when we have a balanced combination of N_m and N_p. As concluded from Figure 8(b), the optimal combinations are $N_m = 96, N_p = 32$ and $N_m = 128, N_p = 24$. Take the case for optimal throughput, we set $N_m = 96$ and $N_p = 32$ in this implementation.

5.2.3 Experimental Results. **Resource Utilization.** Given the parameters that $N_m = 96$ and $N_p = 32$, the detailed post-implementation resource utilization under 200 MHz working frequency on Kintex 325T is listed in Table 3 and resource utilization under 300 MHz working frequency on Ultrascale+ 7EV is listed in Table 4.

Throughput and per DSP throughput for different CNNs. Six representative CNNs, including *slim, medium*, and *deep* networks, are mapped on our processor. When calculating the CNN size, one MAC is counted as two operations. The throughput is measured in GOPS and is reported for different networks on our processor, Intel I7-8700T and Nvidia Xavier NX GPU in Table 5. We map our processor on two typical FPGAs (Xilinx Kintex-7 325T and Xilinx Ultrascale+ MPSoC 7EV) that target on edge applications. For fair comparison, we use the reported int8 results with OpenVINO on Intel I7-8700T [16] and run all the 8-bit fixed-point networks on Nvidia Xavier NX with

Table 5. Comparison between Intel I7-8700T CPU, Nvidia Xavier NX GPU, Existing Accelerators, and Our Processor with Respect to Different CNNs

	Throughput (GOPS)	Inference Latency (ms)	per DSP throughput (GOPS/DSP)						
	AlexNet			VGG16			ResNet50		
Intel i7-8700T [16]	—	—	—	—	—	—	755.6	10.19	—
Nvidia Xavier NX	1317.6	1.72	—	4253.4	7.19	—	2402.4	3.21	—
Ma et al. [29]	—	—	—	715.9	42.74	0.47	611.4	12.59	0.40
RNA[27]	687.8	3.3	—	878.1	34.85	—	804.3	9.57	—
ours on 325T	*1066.4*	*2.13*	*1.39*	*1086.8*	*28.16*	*1.42*	*1101.9*	*6.99*	*1.43*
ours on 7EV	*1599.6*	*1.42*	*2.08*	*1630.2*	*18.77*	*2.12*	*1652.9*	*4.66*	*2.15*
	ResNet101			ResNet152			DenseNet201		
Intel i7-8700T [16]	—	—	—	—	—	—	—	—	—
Nvidia Xavier NX	2584	5.88	—	2734.6	8.26	—	972	11.1	—
Ma et al. [29]	—	—	—	707.2	31.96	0.47	—	—	—
RNA[27]	—	—	—	—	—	—	—	—	—
ours on 325T	*1121.4*	*13.6*	*1.46*	*1121.3*	*20.2*	*1.46*	*1104.7*	*9.78*	*1.44*
ours on 7EV	*1682.1*	*9.04*	*2.19*	*1682.0*	*13.4*	*2.19*	*1657.1*	*6.52*	*2.16*

"—" means no reported results.

Table 6. Comparison with Prior Accelerators on VGG16

	Mei et al. [30]	Xiao et al. [56]	Ma et al. [29]	Angel-Eye [9]	RNA [27]	BFP [23]	*ours*
Year	2017	2017	2018	2018	2018	2019	*2019*
Platform	XC7VX690T	XC7Z045	Arria 10 GX1150	XC7Z020	XC7Z045	XC7VX690T	*XC7K325T*
Frequency (MHz)	200	100	200	214	—	200	*200*
Quantization	16-bit	16-bit	16-bit	8-bit	8/4-bit	8-bit	*8-bit*
Strategy	floating	fixed	fixed	fixed	fixed/log	block floating	*floating*
DSP Used	1728	824	1518	780	—	1027	*768*
Throughput (GOPS)	202.42	229.55	715.9	84.3 (CONV)	878.11	760.83	*1086.8*
per DSP Throughput (GOPS/DSP)	0.117	0.279	0.472	0.444	—	0.741	*1.42*
Power (W)	10.81	9.4	—	3.5	10.56	9.18	*9.42*
Power Efficiency (GOPS/W)	18.72	24.42	—	24.1	83.15	82.88	*115.4*

"—" means no reported results.

TensorRT from PyTorch models [18]. As the existing studies [27, 29] support multiple networks, we also include their results in Table 5.

Compared with the existing accelerators, our processor outperforms them in both throughput, inference latency and per DSP throughput. Particularly, the average improvement of throughput is 63.5% and 1.45× for 325T and 7EV compared with Reference [29], respectively. Meanwhile, compared with RNA, we can achieve 38.6% and 1.08× better throughput for 325T and 7EV, respectively. Moreover, the average improvement of per DSP throughput is 2.2× and 3.9× compared with Reference [29] for 325T and 7EV, respectively. In the approach proposed in Reference [27], they use LUT to implement multipliers, so we do not compare per DSP throughput with them. For the comparison with CPU on Resnet50 that is only reported by OpenVINO, our FPGA processor outperforms 82.3% in terms of throughput because of the high parallelism in our processor. Although Nvidia Xavier NX can achieve higher throughput (due to more hardware resources) in most cases than our processor does, their computation efficiency is low as their peak throughput is reported to be 21TOPS [17].

Comparison with Previous Accelerators on VGG16. We run the classification network VGG16 on our processor and compare the results with six typical studies, as shown in Table 6. We also list the detailed implementation information, such as platform, working frequency, and quantization strategy in Table 6. First, our processor, which uses the LPFP quantization scheme, has a negligible top-1 and top-5 accuracy degradation. Although the work in Reference [30] and

Table 7. Comparison with Prior Accelerators on YOLO

	Ma et al. [28]	Aristotle [8]	Angel-Eye [9]	Wai et al. [48]	*ours*	
Year	2017	2017	2018	2018	*2019*	
Platform	XC7V485T	XC7020	XC7Z020	Cyclone V	*XC7K325T*	
Frequency (MHz)	143	214	–	117	*200*	
Quantization Strategy	16-bit fixed	8-bit fixed	8-bit fixed	8-bit fixed	*8-bit floating*	
Network	tiny-yolo	tiny-yolo	tiny-yolo	tiny-yolo-v2	*tiny-yolo*	*tiny-yolo-v2*
mAP loss (%)	—	—	—	—	*0.3*	*0.1*
DSP Used	112	198	—	122	*768*	
Throughput (GOPS)	48	36.5	62.9	21.6	*987.2*	*1095.4*
per DSP Throughput (GOPS/DSP)	0.429	0.184	—	0.177	*1.29*	*1.43*

"—" means no reported results.

Reference [27] can also maintain negligible accuracy loss, the approach in Reference [30] uses 16-bit floating-point data format, which results in higher bandwidth and memory requirement and lower per DSP throughput, while the approach in Reference [27] needs 144 extra hours for the fine-tuning process. Second, our processor outperforms all the six accelerators in terms of throughput and per DSP throughput. Particularly, the improvements of throughput is from 24% to 11.89× and per DSP throughput is from 92% to 11.14×, respectively. These improvements mainly come from the parallel computation pattern in FPFU and the implementation of four 4-bit MACs within one DSP slice. To the best of our knowledge, this is the first work that can simplify the multiplication to 4-bit and implement four MACs inside one DSP slice while maintaining comparable top-1/top-5 accuracy without any retraining process. Finally, we also show the power efficiency in Table 6, and our processor improves the power efficiency by 39%—5.16× on average compared with the existing accelerators.

Comparison with Previous Accelerators on YOLO. We further compare the detection network YOLO [39, 40] with prior accelerators [8, 9, 28, 48] and we use the tiny version of the YOLO network. The comparison results are shown in Table 7, where we also list the **mean average precision (mAP)** loss of our quantized networks. Compared with the full-precision network, the mAP loss of quantized tiny-yolo and tiny-yolo-v2 is 0.3% and 0.1%, respectively. The hardware comparison with prior accelerators shows that our processor is 20.1× and 49.7× higher in terms of throughput for tiny-yolo and tiny-yolo-v2, respectively. Moreover, due to the implementation of four 4-bit MACs within one DSP slice, the per DSP throughput improves by 5× compared with prior accelerators on average.

6 RELATED WORK

Weight and Computation Reduction. CNNs are typically over-parameterized, and extensive accelerator developers in recent years focus on using CNN approximation algorithms, including weight reduction, computation complexity reduction, and quantization to accelerate CNN inference [49]. The accelerator proposed in References [26, 55] used Winograd algorithm to reduce the number of multiplications in convolution, thus reducing computation complexity. EIE [10], Cambricon-X [61], and Cambricon-S [64] were the mainstreaming accelerators that benefit from weight and computation complexity reduction techniques. Unnecessary computations (i.e., multiplication of zeros) in CNN inference are skipped for better inference time and energy [1, 44]. All these methods for computation reduction take the 8-bit fixed-point as the target data representation. However, the irregularity caused by these algorithms degrades the parallelism and hardware efficiency [51].

Quantization. Accelerators with quantization is another concentration. XNOR_Net [37] applied weights binarization by quantizing weights into {−1, 1} with a scaling factor for AlexNet. The lightweight YOLOv2 [32] was another binarization approach that focused on object detection CNN. Accelerator with ternary representation, which added zero to the binary set, was introduced to help improve the accuracy [36]. Although these accelerators achieve remarkable power and storage saving, they both suffer from significant accuracy loss. Moreover, they all need time-consuming retraining process to compensate for the quantization error. The 16-bit quantization oriented accelerators, including floating-point and fixed-point representations, solved the problem of accuracy loss [28–30, 56]. However, the storage requirement is still huge, and the per DSP throughput is extremely low (less than 0.5GOPS/DSP) because of the usage of 16-bit.

Eight-bit quantization makes a tradeoff between storage and accuracy. The accelerators [9, 27, 63] optimized the computation patterns with 8-bit fixed-point quantization to improve the performance for different CNNs. DNNBuilder [62] was proposed to automatically build DNN accelerators to satisfy the performance and power efficiency demands on embedded and cloud FPGAs, while Cloud-DNN [4] was the framework for mapping DNN models to cloud FPGAs. Block floating-point scheme with 8-bit mantissa was used in Reference [23] to accelerate the inference of CNN while maintaining accuracy. However, all these accelerators need 8-bit MAC to perform convolution, leading to a low per DSP throughput (less than 0.8 GOPS/DSP). A more aggressive method quantized the small values of the weights into 4 bits and keeps the remaining 16 bits as full precision, by dividing the weights into the low-precision and high-precision regions according to the values of the weights [34]. HAQ [50] proposed a mixed-precision quantization approach with a tradeoff between quantization policy and hardware performance. However, both studies need a time-consuming retraining process to compensate for quantization errors.

Different from all the above methods, the proposed LPFP quantization scheme fully exploits the properties of weights and activations, thus obtaining a comparable or better accuracy for *deep* CNNs. Moreover, the LPFP quantization method gets rid of the time-consuming retraining process that needs labelled data and extra computing, because access to labelled data can be difficult in practice as hardware and CNN algorithms are often developed by different parties. Furthermore, with the help of the LPFP quantization method, our processor only needs 4-bit MACs, thus dramatically improving the per DSP throughput. Moreover, our LPFP quantization method can also be applied to the aforementioned computation reduction methods. This is because the LPFP and fixed-point data representation share the same representation of zeros, and the aforementioned computation reduction methods all focus on eliminating the computations on zeros. Overall, the proposed processor achieves better performance on FPGA.

7 CONCLUSION

We have proposed a low-precision floating-point quantization method, called LPFP, to reduce memory size and memory access with negligible accuracy degradation (less than 0.5% for top-1 and 0.3% for top-5 accuracy) for CNN interference. Furthermore, we have reduced the bit width for multiplication to 4-bit with comparable accuracy and implemented four 4-bit MACs within one DSP48E1 slice in Xilinx Kintex 7 FPGA family or DSP48E2 in Xilinx Ultrascale/Ultrascale+ FPGA family. Experiments using Xilinx KC705 platforms and six typical CNN networks show that we achieve an average throughput and per DSP throughput of 1100.4 GOPS and 1.43 GOPS, respectively. Moreover, the average throughput for these networks is 82.3% and 1.5× over Intel I7-8700T CPU and existing accelerators, respectively. Particularly for VGG16 and YOLO, we outperform six existing accelerators in terms of average throughput by 3.5× and 27.5×, while improving per DSP throughput by 4.1× and 5×, respectively. To the best of our knowledge, this is the first in-depth work that can simplify the multiplication to 4-bit and accommodate four MACs in one DSP slice while maintaining comparable top-1/top-5 accuracy without any retraining.

APPENDIX

A MERGING BATCH NORMALIZATION LAYER INTO CONVOLUTIONAL LAYER

To simplify the design on FPGA, we merge the batch normalization layer into the convolutional layer in advance. Here we explained the merging process in detail. The convolutional layer is computed as follows:

$$y = \sum x \times w + b, \tag{9}$$

where x, w, b, and y denote the input activation, weight, bias and output activation, respectively. The batch normalization layer followed is computed as follows:

$$z = \gamma \frac{y - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta, \tag{10}$$

where γ, σ, ϵ, and β are the normalization parameters, respectively, which are kept the same for each layer during inference, and z are the outputs of batch normalization layer. Equation (9) is substituted into Equation (10), shown as

$$\begin{aligned} z &= \gamma \frac{\sum x \times w + b - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta \\ &= \sum x \times \frac{\gamma w}{\sqrt{\sigma^2 + \epsilon}} + \frac{\gamma (b - \mu)}{\sqrt{\sigma^2 + \epsilon}} + \beta. \end{aligned} \tag{11}$$

In this way, we can define the new weights and biases as Equation (12) to merge the batch normalization layer into the convolutional layer.

$$\begin{aligned} w' &= \frac{\gamma w}{\sqrt{\sigma^2 + \epsilon}} \\ b' &= \frac{\gamma (b - \mu)}{\sqrt{\sigma^2 + \epsilon}} + \beta. \end{aligned} \tag{12}$$

REFERENCES

[1] Vahideh Akhlaghi, Amir Yazdanbakhsh, Kambiz Samadi, Rajesh K. Gupta, and Hadi Esmaeilzadeh. 2018. Snapea: Predictive early activation for reducing computation in deep convolutional neural networks. In *Proceedings of the ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA'18)*. IEEE, 662–673.

[2] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Qiang Cheng, Guoliang Chen, et al. 2016. Deep speech 2: End-to-end speech recognition in english and mandarin. In *Proceedings of the International Conference on Machine Learning*. 173–182.

[3] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *ACM SIGARCH Computer Architecture News* 42, 1 (2014), 269–284.

[4] Yao Chen, Jiong He, Xiaofan Zhang, Cong Hao, and Deming Chen. 2019. Cloud-DNN: An open framework for mapping DNN models to cloud FPGAs. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 73–82.

[5] Philip Colangelo, Nasibeh Nasiri, Eriko Nurvitadhi, Asit Mishra, Martin Margala, and Kevin Nealis. 2018. Exploration of low numeric precision deep learning inference using intel® FPGAs. In *Proceedings of the IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'18)*. IEEE, 73–80.

[6] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2015. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in Neural Information Processing Systems*. 3123–3131.

[7] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V. Le, et al. 2012. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*. 1223–1231.

[8] Kaiyuan Guo, Song Han, Song Yao, Yu Wang, Yuan Xie, and Huazhong Yang. 2017. Software-hardware codesign for efficient neural network acceleration. *IEEE Micro* 37, 2 (2017), 18–25.

[9] Kaiyuan Guo, Lingzhi Sui, Jiantao Qiu, Jincheng Yu, Junbin Wang, Song Yao, Song Han, Yu Wang, and Huazhong Yang. 2017. Angel-eye: A complete design flow for mapping CNN onto embedded FPGA. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 37, 1 (2017), 35–47.

[10] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: Efficient inference engine on compressed deep neural network. In *Proceedings of the ACM/IEEE 43rd Annual International Symposium on Computer Architecture.* 243–254.

[11] Song Han, Huizi Mao, and William J. Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding. arXiv:1510.00149. Retrieved from https://arxiv.org/abs/1510.00149.

[12] Song Han, Jeff Pool, John Tran, and William Dally. 2015. Learning both weights and connections for efficient neural network. In *Advances in Neural Information Processing Systems.* 1135–1143.

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.* 770–778.

[14] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. 2017. Densely connected convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.* 4700–4708.

[15] Intel Inc. 2021. INT8 vs FP32 Comparison on Select Networks and Platforms. Retrieved from https://docs.openvinotoolkit.org/latest/openvino_docs_performance_int8_vs_fp32.html.

[16] Intel Inc. 2021. OpenVINO™ Model Server Benchmark Results. Retrieved from https://docs.openvinotoolkit.org/latest/openvino_docs_performance_benchmarks_ovms.html.

[17] Nvidia Inc. 2020. Jetson Xavier NX. Retrieved from https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-xavier-nx/.

[18] Nvidia Inc. 2021. torch2trt. Retrieved from https://github.com/NVIDIA-AI-IOT/torch2trt.

[19] Sambhav R Jain, Albert Gural, Michael Wu, and Chris H Dick. 2019. Trained quantization thresholds for accurate and efficient fixed-point inference of deep neural networks. arXiv:1903.08066. Retrieved from https://arxiv.org/abs/1903.08066.

[20] Hamza Khan, Asma Khan, Zainab Khan, Lun Bin Huang, Kun Wang, and Lei He. 2021. NPE: An FPGA-based overlay processor for natural language processing. arXiv:2104.06535. Retrieved from https://arxiv.org/abs/2104.06535.

[21] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems.* 1097–1105.

[22] Liangzhen Lai, Naveen Suda, and Vikas Chandra. 2017. Deep convolutional neural network inference with floating-point weights and fixed-point activations. arXiv:1703.03073. Retrieved from https://arxiv.org/abs/1703.03073.

[23] Xiaocong Lian, Zhenyu Liu, Zhourui Song, Jiwu Dai, Wei Zhou, and Xiangyang Ji. 2019. High-performance FPGA-based CNN accelerator with block-floating-point arithmetic. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27, 8, (2019), 1874–1885.

[24] Zhouhan Lin, Matthieu Courbariaux, Roland Memisevic, and Yoshua Bengio. 2015. Neural networks with few multiplications. arXiv:1510.03009. Retrieved from https://arxiv.org/abs/1510.03009.

[25] IP LogiCORE. 2012. Floating-point Operator v6. 0. Xilinx Inc.

[26] Liqiang Lu, Yun Liang, Qingcheng Xiao, and Shengen Yan. 2017. Evaluating fast algorithms for convolutional neural networks on FPGAs. In *Proceedings of the IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'17).* IEEE, 101–108.

[27] Cheng Luo, Yuhua Wang, Wei Cao, Philip H. W. Leong, and Lingli Wang. 2018. RNA: An accurate residual network accelerator for quantized and reconstructed deep neural networks. In *Proceedings of the 28th International Conference on Field Programmable Logic and Applications (FPL'18).* IEEE, 60–603.

[28] Jing Ma, Li Chen, and Zhiyong Gao. 2017. Hardware implementation and optimization of tiny-yolo network. In *Proceedings of the International Forum on Digital TV and Wireless Multimedia Communications.* Springer, 224–234.

[29] Yufei Ma, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. 2018. Optimizing the convolution operation to accelerate deep neural networks on FPGA. *IEEE Trans. VLSI Syst.* 26, 7 (2018), 1354–1367.

[30] Chunsheng Mei, Zhenyu Liu, Yue Niu, Xiangyang Ji, Wei Zhou, and Dongsheng Wang. 2017. A 200MHZ 202.4 GFLOPS@ 10.8 W VGG16 accelerator in Xilinx VX690T. In *Proceedings of the IEEE Global Conference on Signal and Information Processing (GlobalSIP'17).* IEEE, 784–788.

[31] Szymon Migacz. 2017. 8-bit inference with TensorRT. In *Proceedings of the GPU Technology Conference.*

[32] Hiroki Nakahara, Haruyoshi Yonekawa, Tomoya Fujii, and Shimpei Sato. 2018. A lightweight yolov2: A binarized cnn with a parallel support vector regression for an fpga. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays.* ACM, 31–40.

[33] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric S. Chung. 2015. Accelerating deep convolutional neural networks using specialized hardware. *Microsoft Resrarch Whitepaper* 2, 11 (2015), 1–4.

[34] Eunhyeok Park, Dongyoung Kim, and Sungjoo Yoo. 2018. Energy-efficient neural network accelerator based on outlier-aware low-precision computation. In *Proceedings of the ACM/IEEE 45th Annual International Symposium on Computer Architecture*. 688–698.

[35] Eunhyeok Park, Sungjoo Yoo, and Peter Vajda. 2018. Value-aware quantization for training and inference of neural networks. In *Proceedings of the European Conference on Computer Vision*. 580–595.

[36] Adrien Prost-Boucle, Alban Bourge, Frédéric Pétrot, Hande Alemdar, Nicholas Caldwell, and Vincent Leroy. 2017. Scalable high-performance architecture for convolutional ternary neural networks on FPGA. In *Proceedings of the 27th International Conference on Field Programmable Logic and Applications*. 1–7.

[37] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. Xnor-net: Imagenet classification using binary convolutional neural networks. In *Proceedings of the European Conference on Computer Vision*. 525–542.

[38] Joseph Redmon. 2013–2016. Darknet: Open Source Neural Networks in C. Retrieved from http://pjreddie.com/darknet/.

[39] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. 2016. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 779–788.

[40] Joseph Redmon and Ali Farhadi. 2017. YOLO9000: better, faster, stronger. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 7263–7271.

[41] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet large scale visual recognition challenge. *Int. J. Comput. Vis.* 115, 3 (2015), 211–252.

[42] Sean O Settle, Manasa Bollavaram, Paolo D'Alberto, Elliott Delaye, Oscar Fernandez, Nicholas Fraser, Aaron Ng, Ashish Sirasao, and Michael Wu. 2018. Quantizing convolutional neural networks for low-power high-throughput inference engines. arXiv:1805.07941. Retrieved from https://arxiv.org/abs/1805.07941.

[43] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. arXiv:1409.1556. Retrieved from https://arxiv.org/abs/1409.1556.

[44] Mingcong Song, Jiechen Zhao, Yang Hu, Jiaqi Zhang, and Tao Li. 2018. Prediction based execution on deep neural networks. In *Proceedings of the ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA'18)*. IEEE, 752–763.

[45] Zhourui Song, Zhenyu Liu, and Dongsheng Wang. 2018. Computation error analysis of block floating point arithmetic oriented convolution neural network accelerator design. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence*.

[46] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 1–9.

[47] Mingxing Tan and Quoc Le. 2019. EfficientNet: Rethinking model scaling for convolutional neural networks. In *Proceedings of the International Conference on Machine Learning*. 6105–6114.

[48] Yap June Wai, Zulkalnain bin Mohd Yussof, Sani Irwan bin Salim, and Lim Kim Chuan. 2018. Fixed point implementation of tiny-yolo-v2 using opencl on fpga. *Int. J. Adv. Comput. Sci. Appl.* 9, 10 (2018), 506–512.

[49] Erwei Wang, James J Davis, Ruizhe Zhao, Ho-Cheung Ng, Xinyu Niu, Wayne Luk, Peter YK Cheung, and George A Constantinides. 2019. Deep neural network approximation for custom hardware: Where we've been, where we're going. arXiv:1901.06955. Retrieved from https://arxiv.org/abs/1901.06955.

[50] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. 2019. HAQ: Hardware-aware automated quantization with mixed precision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 8612–8620.

[51] Shuo Wang, Zhe Li, Caiwen Ding, Bo Yuan, Qinru Qiu, Yanzhi Wang, and Yun Liang. 2018. C-lstm: Enabling efficient lstm using structured compression techniques on fpgas. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 11–20.

[52] Xuechao Wei, Cody Hao Yu, Peng Zhang, Youxiang Chen, Yuxin Wang, Han Hu, Yun Liang, and Jason Cong. 2017. Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs. In *Proceedings of the 54th Annual Design Automation Conference*. 29.

[53] Chen Wu, Mingyu Wang, Xinyuan Chu, Kun Wang, and Lei He. 2020. Low precision floating point arithmetic for high performance FPGA-based CNN acceleration. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 318–318.

[54] Chen Wu, Mingyu Wang, Xiayu Li, Jicheng Lu, Kun Wang, and Lei He. 2020. Phoenix: A low-precision floating-point quantization oriented architecture for convolutional neural networks. arXiv:2003.02628. Retrieved from https://arxiv.org/abs/2003.02628.

[55] Di Wu, Jin Chen, Wei Cao, and Lingli Wang. 2018. A novel low-communication energy-efficient reconfigurable CNN acceleration architecture. In *Proceedings of the 28th International Conference on Field Programmable Logic and Applications (FPL'18)*. IEEE, 64–643.

[56] Qingcheng Xiao, Yun Liang, Liqiang Lu, Shengen Yan, and Yu-Wing Tai. 2017. Exploring heterogeneous algorithms for accelerating deep convolutional neural networks on FPGAs. In *Proceedings of the 54th ACM/EDAC/IEEE Design Automation Conference (DAC'17)*. IEEE, 1–6.

[57] Yunxuan Yu, Chen Wu, Tiandong Zhao, Kun Wang, and Lei He. 2019. OPU: An FPGA-based overlay processor for convolutional neural networks. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 28, 1 (2019), 35–47.

[58] Yunxuan Yu, Tiandong Zhao, Kun Wang, and Lei He. 2020. Light-OPU: An FPGA-based overlay processor for lightweight convolutional neural networks. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 122–132.

[59] Yunxuan Yu, Tiandong Zhao, Mingyu Wang, Kun Wang, and Lei He. 2020. Uni-OPU: An FPGA-based uniform accelerator for convolutional and transposed convolutional networks. *IEEE Trans. VLSI Syst.* 28, 7 (2020), 1545–1556.

[60] Chen Zhang, Guangyu Sun, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. 2016. Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. 1–8.

[61] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. 2016. Cambricon-x: An accelerator for sparse neural networks. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*. 1–12.

[62] Xiaofan Zhang, Junsong Wang, Chao Zhu, Yonghua Lin, Jinjun Xiong, Wen-mei Hwu, and Deming Chen. 2018. DNNBuilder: An automated tool for building high-performance DNN hardware accelerators for FPGAs. In *Proceedings of the International Conference on Computer-Aided Design*. ACM, 56.

[63] Tiandong Zhao, Yunxuan Yu, Kun Wang, and Lei He. 2021. Heterogeneous dual-core overlay processor for lightweight CNNs. In *Proceedings of the IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'21)*. IEEE, 264–264.

[64] Xuda Zhou, Zidong Du, Qi Guo, Shaoli Liu, Chengsi Liu, Chao Wang, Xuehai Zhou, Ling Li, Tianshi Chen, and Yunji Chen. 2018. Cambricon-S: Addressing irregularity in sparse neural networks through a cooperative software/hardware approach. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*. 15–28.

[65] Dan Zuras, Mike Cowlishaw, Alex Aiken, Matthew Applegate, David Bailey, Steve Bass, Dileep Bhandarkar, Mahesh Bhat, David Bindel, Sylvie Boldo, et al. 2008. IEEE standard for floating-point arithmetic. IEEE Std 754-2008 (2008), 1–70.

Received January 2021; revised May 2021; accepted July 2021