High-Performance Acceleration of 2-D and 3-D CNNs on FPGAs Using Static Block Floating Point

Hongxiang Fan, Shuanglong Liu, Zhiqiang Que, Xinyu Niu, and Wayne Luk, *Fellow, IEEE*

***Abstract*— Over the past few years, 2-D convolutional neural networks (CNNs) have demonstrated their great success in a wide range of 2-D computer vision applications, such as image classification and object detection. At the same time, 3-D CNNs, as a variant of 2-D CNNs, have shown their excellent ability to analyze 3-D data, such as video and geometric data. However, the heavy algorithmic complexity of 2-D and 3-D CNNs imposes a substantial overhead over the speed of these networks, which limits their deployment in real-life applications. Although various domain-specific accelerators have been proposed to address this challenge, most of them only focus on accelerating 2-D CNNs, without considering their computational efficiency on 3-D CNNs. In this article, we propose a unified hardware architecture to accelerate both 2-D and 3-D CNNs with high hardware efficiency. Our experiments demonstrate that the proposed accelerator can achieve up to 92.4% and 85.2% multiply-accumulate efficiency on 2-D and 3-D CNNs, respectively. To improve the hardware performance, we propose a hardware-friendly quantization approach called static block floating point (BFP), which eliminates the frequent representation conversions required in traditional dynamic BFP arithmetic. Comparing with the integer linear quantization using zero-point, the static BFP quantization can decrease the logic resource consumption of the convolutional kernel design by nearly 50% on a field-programmable gate array (FPGA). Without time-consuming retraining, the proposed static BFP quantization is able to quantize the precision to 8-bit mantissa with negligible accuracy loss. As different CNNs on our reconfigurable system require different hardware and software parameters to achieve optimal hardware performance and accuracy, we also propose an automatic tool for parameter optimization. Based on our hardware design and optimization, we demonstrate that the proposed accelerator can achieve 3.8–5.6 times higher energy efficiency than graphics processing unit (GPU) implementation. Comparing with the state-of-the-art FPGA-based accelerators, our design achieves higher generality and up to 1.4–2.2 times higher resource efficiency on both 2-D and 3-D CNNs.**

Manuscript received February 5, 2021; revised June 14, 2021; accepted September 17, 2021. This work was supported in part by the U.K. EPSRC under Grant EP/L016796/1, Grant EP/N031768/1, Grant EP/P010040/1, Grant EP/V028251/1, and Grant EP/S030069/1; in part by the National Natural Science Foundation of China under Grant 62001165; in part by the Hunan Provincial Natural Science Foundation of China under Grant 2021JJ40357; in part by the Changsha Municipal Natural Science Foundation under Grant kq2014079; and in part by the funds from Corerain, Maxeler, Intel, Xilinx, and the State Key Laboratory of Space-Ground Integrated Information Technology (SGIIT). *(Corresponding author: Shuanglong Liu.)*

Hongxiang Fan, Zhiqiang Que, and Wayne Luk are with the Department of Computing, Imperial College London, London SW7 2AZ, U.K.

Shuanglong Liu is with the School of Physics and Electronics, Hunan Normal University, Changsha 410081, China (e-mail: liu.shuanglong@hunnu.edu.cn).

Xinyu Niu is with Shenzhen Corerain Technologies Company Ltd., Shenzhen 518048, China.

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TNNLS.2021.3116302.

Digital Object Identifier 10.1109/TNNLS.2021.3116302

***Index Terms*— Field-programmable gate array (FPGA), static block floating point (BFP), three-dimensional convolutional neural network (3-D CNN).**

I. Introduction

In RECENT years, deep neural networks (DNNs), especially convolutional neural networks (CNNs), have demonstrated their great potential in various computer vision (CV) applications. In particular, 2-D CNNs, which perform 2-D convolution in the spatial domain to extract 2-D features, have achieved state-of-the-art accuracy in a wide range of CV tasks, including image classification [1] and object detection [2]. Besides 2-D CNNs, 3-D CNNs [3], due to their ability to incorporate the 3-D information based on 3-D convolution, have been also adopted in various 3-D CV scenarios, such as human action recognition [4] and 3-D medical imaging segmentation [5].

Nevertheless, the memory and computational complexity of 2-D and 3-D convolutions put a heavy burden on their hardware performance on general-purpose processors, which restrains their application in real-life scenarios [6]. For instance, a classical 3-D CNN designed for human action recognition called $C3D$ requires nearly 78 GOPs, and thus, achieves only 951 ms per inference for a 16-frame video on an Intel i5 CPU, which cannot meet the requirement of real-time processing [7]. Therefore, there is a great demand for domain-specific accelerators for both 2-D and 3-D CNNs. Different hardware platforms, including graphics processing units (GPUs), application-specific integrated circuits (ASICs), and field-programmable gate arrays (FPGAs), have been used to accelerate both 2-D and 3-D CNNs. Among all these hardware platforms, FPGAs are gaining popularity because of their better flexibility than ASICs and higher energy efficiency than GPUs [8], [9]. In spite of these advantages, there are several challenges when accelerating both 2-D and 3-D CNNs on FPGA:

1) To improve the hardware performance, most FPGA-based accelerators tend to utilize low-precision weights or activations [8]. However, previous work either shows significant accuracy loss, such as fixed-point [10] and logarithm arithmetic [11], or requires a large amount of hardware resource on implementing quantization modules, such as linear quantization [12] and dynamic BPF quantization [13].
2) Convolution operations, especially 3-D convolution, are highly memory and computation-intensive, making it

hard to achieve high performance on FPGAs with limited memory and computational resources [7].

3) Unifying 2-D and 3-D CNNs onto one hardware architecture is challenging since they have different data locality, and improper design may cause a high degree of data replication [14].

To address these challenges, this article proposes a high-performance reconfigurable architecture for accelerating 2-D and 3-D CNNs. To improve hardware efficiency, the design unifies both 2-D and 3-D convolution onto one computational engine and supports three different types of parallelism for different CNNs. To reduce the computational complexity, we propose a novel hardware-friendly quantization method called static block floating-point (BFP) quantization. The proposed static BFP quantization consumes fewer hardware resources than both linear integer quantization and traditional dynamic BFP quantization while achieving the same level of accuracy. Furthermore, as different CNNs on our proposed reconfigurable system require different software and hardware parameters to achieve optimal performance, an automatic tool is proposed to optimize these parameters given the running CNN and dataset. Therefore, our contributions in this article include the following.

1) A hardware-friendly quantization method called static BFP quantization. Without retraining, it can quantize both the weights and activations in 2-D and 3-D CNNs to 8-bit mantissa while maintaining the same level of accuracy. By using fixed shared exponents during inference, the proposed method eliminates the exponent calculation and the frequent FP-BFP and BFP-FP conversions required in traditional BFP arithmetic. Comparing with the integer linear quantization using zero point, our quantization can decrease the logic resource consumption of the convolutional kernel design by nearly 50% on an FPGA (Section III).
2) A uniform FPGA-based accelerator for both 2-D and 3-D CNNs with high hardware performance. By unifying the 2-D and 3-D CNNs into one computational pattern and providing different types of parallelism for different CNNs, the design is able to achieve high hardware efficiency (Section IV). An automatic tool optimizes the accuracy and hardware performance by determining the proper software and hardware parameters. On an Intel Xeon Silver 4110 CPU, the tool only consumes a few minutes for the whole optimization process (Section V).
3) Extensive experiments evaluating a wide range of 2-D and 3-D CNNs on the proposed design in terms of accuracy and hardware performance. Compared with the other state-of-the-art FPGA-based designs and GPU implementation, our design shows higher generality, power efficiency, and resource efficiency (Section VI).

II. Background

This Section introduces the basic operations used in both 2-D and 3-D CNNs. Then, a brief review of FPGA-based accelerators for 2-D and 3-D CNNs is presented. The basic knowledge and applications of BFP will be also illustrated.

TABLE I
NOTATION OF PARAMETERS USED IN THIS ARTICLE

Parameter	Description
H	The height of input feature map
W	The width of input feature map
K_s	The kernel size in spatial dimension
K_t	The kernel size in temporal dimension
N_c	The channel number
N_f	The filter number
N_l	The frame number (length)
PS	The parallelism level along spatial dimension
PK	The parallelism level along kernel dimension
PC	The parallelism level along channel dimension
PF	The parallelism level along filter dimension
BW	Bandwidth between on-chip and off-chip memory
EFF_{io}	Communication efficiency of IO
CLK_{io}	Clock frequency of IO
CLK_{pe}	Clock frequency of processing engine

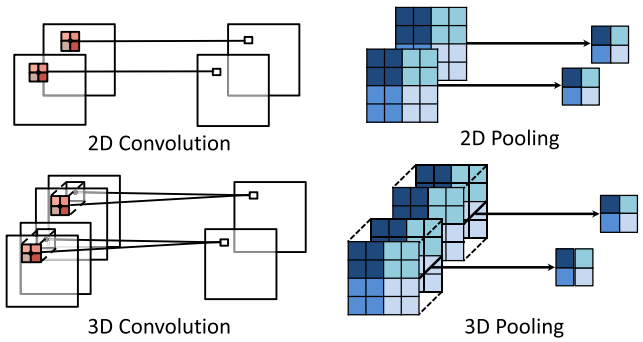


Fig. 1. 2-D and 3-D convolution, channel dimension is not shown for simplicity.

A. Operations in 2-D and 3-D CNNs

In modern 2-D and 3-D CNNs, the commonly used operations include 2-D and 3-D convolution, 2-D and 3-D pooling [3], batch normalization (BN) [15], shortcut addition (SC) [16] and rectified linear unit (ReLU). This part introduces the difference between 2-D and 3-D operations.

1) 2-D and 3-D Convolution: Fig. 1 shows the basic operation of 2-D and 3-D convolution. For each input frame, 2-D CNNs apply one 2-D convolution to generate one output frame without considering the information that existed in the time dimension. However, in order to incorporate the third-dimension information, 3-D convolution applies convolution on multiple frames simultaneously and aggregates the results from different frames to generate one output frame. Therefore, 3-D convolution is more memory and computation-intensive compared with 2-D convolution. Accelerating both 2-D and 3-D convolutions on one uniform accelerator is challenging since their different degrees of data locality require well-designed architecture to avoid data replication [14].

2) 2-D and 3-D Pooling: As shown in Fig. 1, 2-D pooling is applied on one single frame, which is used to decrease the activation in height and width dimensions. Different from 2-D pooling, 3-D pooling is performed in totally 3-D with another dimension as a time dimension. Therefore, for a 3-D pooling

with $K_s \times K_s \times K_t$ kernel size, it receives K_t consecutive frames as input and generates only one output frame. The spatial dimension of this generated output frame is also reduced by $K_s \times K_s$.

Note that, since 2-D convolution is one special case of 3-D convolution with $K_t = 1$, we denote the convolution mentioned in this article to have a kernel size $K_s \times K_s \times K_t$ to represent both 2-D and 3-D convolution. The same meaning is applied to pooling, where the kernel size is $K_s \times K_s \times K_t$ to represent both 2-D and 3-D pooling.

B. Block Floating Point

1) Arithmetic Representation: A BFP number is represented by mantissa bits, exponent bits, and a sign bit. Unlike FP representation which assigns a separate exponent to each number, BFP shares a common exponent in one block of BFP numbers. There are two benefits of BFP in comparison with FP. First, as different BFP numbers belonging to one block share a common exponent, the memory consumption of storing the BFP numbers can be significantly decreased. Second, since BFP-based addition is implemented using the exponent alignment with the fixed-point addition [6], [17], and BFP-based multiplication is implemented using fixed-point multiplication and addition, the computational complexity of BFP is significantly reduced [17].

However, the benefits of BFP also come with burdens. Because the BFP numbers within one block share a common exponent and different layers may use different block sizes and shared exponents, their mantissa parts need to be aligned. During this process, the precision loss may occur, which introduces a decrease in inference accuracy. The situation is getting worse when the variance of BFP numbers increases. In the worst case, the shifting bits required by the exponent alignment may be larger than the mantissa bits, which causes severe precision loss [6].

2) Dynamic BFP Quantization Approach: According to the BFP arithmetic, previous work has proposed a dynamic BFP quantization technique to accelerate both CNN inference [18], [19] and training [8]. The operation of the convolutional layer using the dynamic BFP quantization scheme is shown in Fig. 2(a). As we can see, the dynamic BFP quantization uses the maximal exponent as the shared exponent. Because the maximal exponents may vary for different iterations of inference, the shared exponents need to be determined during the runtime. However, this runtime process requires frequent FP-BFP and BFP-FP conversions, which puts a heavy overhead on both resource consumption and hardware performance. In addition, dynamic BFP quantization requires extra hardware resources for the computation and storage of exponent parts, which further hinders its application in hardware platforms with limited resources.

C. Related Work

1) FPGA-Based CNN Accelerator: Due to the reconfigurability and high energy efficiency, FPGAs have become a popular platform in accelerating CNNs [20]. Various hardware architectures have been proposed, and the initial research effort

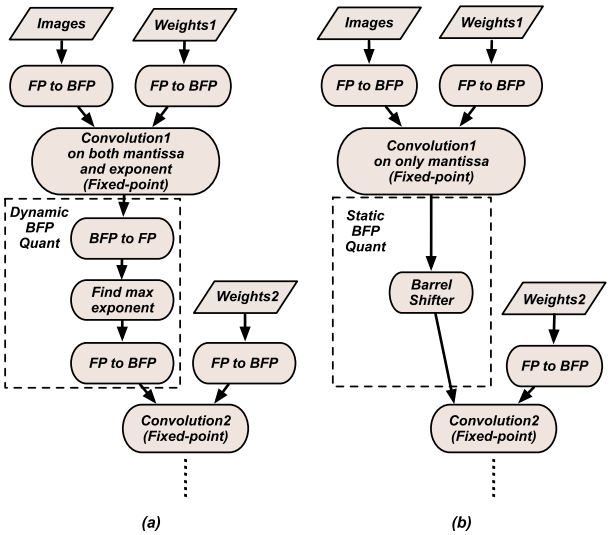


Fig. 2. Operation of convolutions with dynamic and static BFP quantization. (a) Convolution using dynamic BFP quantization. (b) Convolution using static BFP quantization.

focuses on accelerating 2-D CNNs. Venieris and Bouganis [21] proposed a synchronous dataflow (SDF) model for mapping 2-D CNNs to FPGAs based on a streaming architecture. However, the resource consumption of a streaming architecture can become very large when the number of layers increases, which makes it impossible to map large 2-D and 3-D CNNs onto a single FPGA device. Another design methodology is to deploy a single processing engine (PE) for CNN-like operations on the FPGA and accelerate the CNN layer-by-layer sequentially using the same hardware engine [22]. Based on this design methodology, Ma *et al.* [23] further improved the performance by exploring the convolution loop optimization for 2-D CNNs. However, the BFP used in their design requires retraining to recover accuracy. Xing *et al.* [24] also proposed a high-performance FPGA-based design and compiler to accelerate 2-D CNNs. The common drawback in these 2-D CNNs accelerators is that they cannot be used to accelerate 3-D CNNs with high performance [4]. For example, the ordinary convolutional approach adopted in these 2-D CNNs accelerators will cause higher computation complexity when they are used to accelerate 3-D CNNs [14].

Although accelerating 3-D CNNs on FPGA is challenging, several previous works have been proposed to address this problem. Fan *et al.* [4] proposed an FPGA-based architecture called *F-C3D* to accelerate 3-D CNNs. Sun *et al.* [25] adopted hardware-aware pruning to decrease the memory and computational required by 3-D CNNs, which achieved 2.3 times speedup and 2.3 times power efficiency compared with other FPGA implementations. Yang *et al.* [26] proposed a unified dynamically reconfigurable accelerator using a novel Winograd decomposition algorithm to accelerate 2-D CNNs, but its support for 3-D CNNs is unknown. Shen *et al.* [14] first attempted to provide a uniform template-based architecture for both 2-D and 3-D CNNs based on the Winograd algorithm. However, the use of the Winograd algorithm consumes a large number of logical resources on transforming the input and out-

put matrices. Liu *et al.* [10] proposed a uniform architecture based on 2-D multiply-accumulate (MAC) array. Nevertheless, the design uses low-bitwdith fixed-point arithmetic and the accuracy result cannot be guaranteed.

2) Low-Bitwidth Quantization: Quantization [27] and sparsity exploiting [28] are two main-streaming techniques to reduce the algorithmic complexity of CNNs. Since this article mainly focuses on quantization, we refer the reader to a review of the sparsity exploiting by Wang *et al.* [8].

A comprehensive survey on quantization methods for DNNs has been summarized in [29]. There are three main-streaming quantization approaches, i.e., fixed-point, logarithm, and linear quantization. Courbariaux *et al.* [30] explore the use of fixed-point arithmetic on CNNs for both training and inference. However, it is only validated on small datasets and may introduce significant accuracy loss on large datasets such ImageNet [31]. Miyashita *et al.* [11] proposed a logarithmic quantization method which decreased the precision of the VGG-16 [32] and AlexNet [1] to low bitwidth without significant accuracy loss. Nevertheless, this quantization approach has not been evaluated on InceptionV4 and MobileNetv2, which cannot demonstrate its effectiveness on these light-weight models. Jacob *et al.* [12] proposed an integer-only quantization using zero-point to maintain accuracy. They demonstrated in their experiments that the integer quantization with zero-point only introduced negligible accuracy loss on a wide range of CNN models. However, the use of zero-point also introduces a heavy burden on the memory and computational resources, which limits the overall hardware performance. Jain *et al.* [33] proposed a power-of-two scaling quantization with trainable quantization thresholds. However, the process requires time-consuming retraining and its performance on 3-D CNNs is unknown.

The dynamic BFP quantization mentioned in Section II-B is first proposed in [17] and applied for CNN inference. It is able to compress both the activations and weights to 8 bits for ResNet-50 [16] with only negligible accuracy loss on small datasets. Based on the dynamic BFP quantization, Lian *et al.* [13] proposed a high-performance CNN accelerator on FPGA. Although the design uses 8-bit mantissa BFP for the main computation, the precision used in the on-chip and off-chip communication is still 16-bit and it requires frequent conversion between BFP and FP, which brings a heavy burden on the memory usage and bandwidth resource. Different from these works, our proposed quantization scheme uses a fixed shared exponent for different inputs to eliminate the frequent conversion between BFP and FP. The shared exponent is determined before CNN inference by minimizing the KL divergence. We also introduce an automatic tool that optimizes the shared exponents, the bitwidth of mantissa, and the exponent by balancing the tradeoff between accuracy and hardware performance. Our prior work [34] explored the application of static BFP quantization on 2-D CNNs. However, its accuracy performance on 3-D CNNs was unknown. In addition, this article only evaluated the kernel design for 2-D convolution without running the actual CNN models, and it did not study the unified hardware architecture for both 2-D and 3-D CNNs.

III. Static Block Floating-Point Quantization

In this Section, we first introduce the quantization approach and blocking strategy of our static BFP quantization. The operations of 2-D and 3-D CNNs under static BFP will then be presented.

A. Quantization Approach

The quantization approach should consider not only the accuracy performance but also the hardware implementation and performance. As mentioned in Section II-B2, the frequent FP-BFP and BFP-FP conversions required by the dynamic BFP quantization put a heavy overhead on its resource consumption and hardware performance. In order to eliminate the process of finding the shared exponents at runtime like dynamic BFP quantization, our static quantization scheme fixes the shared exponent for different inputs and determines the shared exponents before CNN inference. To achieve this, it is required to collect certain amounts of intermediate results by running the CNNs on different inputs and find the properly shared exponents by minimizing the precision loss. Although simply using the maximal exponents in the collected intermediate statistics is one approach to determine the shared exponents, we found that, on the ImageNet dataset [31], this maximal strategy will cause significant accuracy drop on the CNN models using depthwise convolution, such as MobileNetv2 [35]. To address this issue, we propose another strategy that determines the shared exponent by minimizing the KL divergence [36], which describes the difference between two distributions.

Algorithm 1 Static BFP Quantization Using KL Divergence

```
Run the FP-based CNNs using different inputs
for Each block b do
    Fetch FP statistics and build the target distribution D_t
    Find the maximum exponent e_max
    Use maximal exponent, e^b_opt = e_max
    Initialize D_max as the maximal FP value
    for e_offset = 0 to i do
        e_cur = e_max − e_offset
        Apply BFP quantization using e_cur
        Build the BFP-quantized distribution D_q
        Compute the KL divergence between D_t and D_q
        if D_t < D_max then
            e^b_opt = e_cur, D_max = D_t
```

As illustrated in Algorithm 1, in the beginning, we separate the intermediate results into several blocks according to the block size. Note that the block size is a hyperparameter in our static BFP quantization and we will discuss it in Section III-B. For each block, a histogram can be drawn based on the collected intermediate FP statistics by running CNNs on example inputs, which are used to record the original distribution. Then, the BFP quantization is applied on i blocks based on i different exponents, $e_{\max}$ to $e_{\max}-i+1$, which produces i different BFP-quantized distributions. Our experiments demonstrate that $i = 3$ is enough to find the proper exponents in most cases.

When the BFP quantization is done, we compute the KL divergence between the BFP-quantized distribution and FP distribution for i different exponents. Then, the shared exponent is set to be the exponent with the minimal KL divergence. We iteratively perform this process on each layer to obtain the shared exponents for the whole network. In our experiments later, we found that different CNNs require different strategies to achieve higher accuracy. Therefore, an automatic tool will be proposed in Section V, which determines different strategies of finding shared exponent for different CNNs and models.

B. Blocking Strategy

The block size decides the tradeoff between the precision loss and hardware performance. Although a large block size can decrease the number of shared exponents and the memory consumption, it may also increase the precision loss since the variance within one block becomes large. Therefore, our proposed blocking strategy aims at achieving a balance between precision loss and hardware performance.

In a typical convolutional layer, the shape of the weight tensor is $N_c \times N_f \times K_s \times K_s \times K_t$ and the activation tensor has the size $N_l \times N_c \times H \times W$. In terms of activations, we can block the tensor along the spatial ($H \times W$), channel (N_c) and temporal (N_l) dimensions. However, the convolution needs to accumulate data from different spatial positions and channels, which means different exponents in these two dimensions may cause frequent exponent realignment, and thus, degrades the hardware performance. Since the spatial and channel dimensions are not suitable for blocking, this article blocks the activations along the temporal dimension, which generates N_l shared exponents for each activation tensor. For weights, the blocking can be performed along the kernel ($K_s \times K_s \times K_t$), channel ($N_c$), and filter ($N_f$) dimensions. Since the weights from different kernel positions and channels need to be accumulated together, we only block the weights along the filter dimension, which produces N_f shared exponents for every weight tensor. Using our proposed blocking strategy, there is no need to perform exponent realignment for a single convolutional layer.

To visualize the effect of static BFP quantization while using the proposed exponent strategies (Section III-A) and blocking strategy, Fig. 3 presents the normalized histograms of the output activations of the ninth and 29th convolutional layers in *ResNet–50* using original FP data and quantized data. To compare two different exponent strategies, we quantize the activations using the maximal exponent (max) and shared exponents obtained by minimizing the KL divergence (kl). As we can see, naively using the maximum exponents causes significant precision loss in the output activations of the ninth convolutional layer, making it a different distribution from the original FP data. However, while using the KL divergence to determine the shared exponents, the quantized activations follow a similar distribution as the FP data.

C. CNNs With Static BFP Quantization

Most operations in modern CNNs, such as SC, 2-D, and 3-D convolutions, have different BFP-based implementations

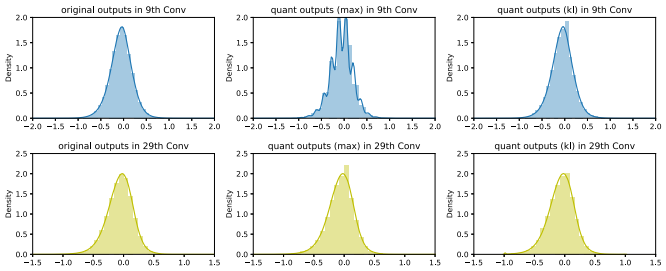


Fig. 3. Normalized histogram of original FP and quantized data in the ninth and 29th convolutional layers of *ResNet–50* on *ImageNet* dataset.

from their FP counterparts. This section introduces how these operations are implemented using the static BFP quantization scheme.

1) Convolution: Fig. 2(b) shows the basic operations of both 2-D and 3-D convolutions when the static BFP quantization is applied. There are two improvements in comparison with the dynamic BFP quantization.

1) Because the proposed quantization scheme determines the shared exponents before the runtime, the frequent FP-BFP and BFP-FP conversions can be replaced by a simple shift operation. For instance, given two consecutive convolutional layers with the shared exponents a and b, it only needs to perform the exponent realignment that shifts the mantissa parts by $a-b$ bits, which avoids the trivial data conversions.
2) Since the shared exponents are already known, the required shifting bits for each layer can be precomputed before runtime. Therefore, the need of calculating exponents can be eliminated under the static BFP quantization scheme, which significantly decreases the usage of the memory and computational resources.

2) Shortcut Addition: In modern CNNs, such as ResNet, SC [16], [37] has been widely used for residual learning. The computation of a typical ResNet-like block with FP arithmetic is presented in Fig. 4. The SC adds the outputs of the second convolution and the original inputs together to obtain the final results. However, since the exponents of both inputs and outputs are different under the static BFP quantization scheme, the addition cannot be simply performed in an original way. To implement SC under static BFP quantization, a shift operation is required before the SC to align the exponents of the two tensors, which is presented in Fig. 4.

To find the properly shared exponents for BFP-based SC, one straightforward approach is to simply use the maximal exponents in the output tensor. However, we found that it will bring a significant accuracy drop on the CNN models using the depthwise convolution, such as MobileNetv2. To address this issue, we observe that the shared exponents of the SC should be determined based on the output tensor as well as the original input tensor. Therefore, both inputs and outputs are concatenated into one tensor, and the shared exponents are determined by minimizing the KL divergence of this concatenated tensor.

3) Batch Normalization, Pooling, and ReLU: BN has been widely used in modern CNNs to address the internal covariate

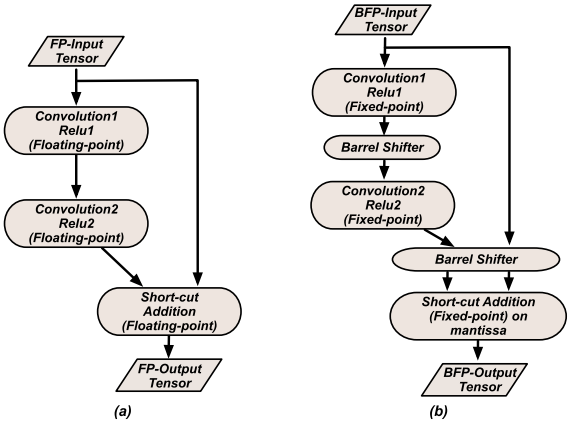


Fig. 4. SC using FP and BFP. (a) SC using floating point. (b) SC using BFP.

shift issue caused during training [15]. However, as the parameters of BN are fixed during inference, we adopt the fused BN optimization [38] to merge the BN parameters into the weights and biases of convolution, which saves a lot of resources for designing the BN module in hardware.

The ReLU and pooling, including both 2-D and 3-D pooling, are mainly composed of addition and comparison which operate along the spatial and temporal dimensions. As mentioned in Section III-B, since our blocking is only performed along the filter dimension, the activations belonging to the same spatial and temporal dimensions share the same exponents. Therefore, ReLU and pooling under static BFP quantization can still be simply performed in a fixed-point manner on their mantissa parts.

IV. Hardware Implementation

In this section, we first explore a uniform computational pattern for both 2-D and 3-D CNNs. Then, a unified hardware architecture is proposed based on static BFP. Several hardware optimization techniques will be also presented to improve the hardware performance and efficiency.

A. Uniform Computational Pattern

The pseudocode of 2-D and 3-D convolution is presented in Algorithm 2, where 2-D convolution is one special case of 3-D convolution when K_t equals 1. The key problem in accelerating CNNs is how to allocate hardware resources on FPGA spatially and how to temporally assign the required computation to each hardware module. For instance, the design proposed in [4] and [6] contains W numbers of MAC units and each MAC unit is composed of $K_t \times K_s \times K_s$ multipliers followed by an adder tree. Therefore, the computation of the four innermost loops is mapped into the computational engine spatially and the rest of the calculation in the other loops is performed temporally by running on the same engine sequentially. However, the main drawback in these designs is that K_t and K_s vary in different layers and networks, which significantly reduces the generality and performance of these accelerators.

Fig. 5 visualizes the computation of convolution as matrix multiplication in two different manners. Previous work adopts

Algorithm 2 Pseudocode of 2-D and 3-D Convolution

```
for (fr = 0; fr < Nl; fr++)
  for (fi = 0; fi < Nf; fi++)
    for (c = 0; c < Nc; c++)
      for (h = 0; h < H; h++)
        for (w = 0; w < W; w++)
          for (kt = 0; kt < Kt; kt++)
            for (kh = 0; kh < Ks; kh++)
              for (kw = 0; kw < Ks; kw++)
                O[fr][fi][h][w] += W[kt][fi][c][kh][kw] × I[fr + kt][fi][c][h + kh][w + kw]
```

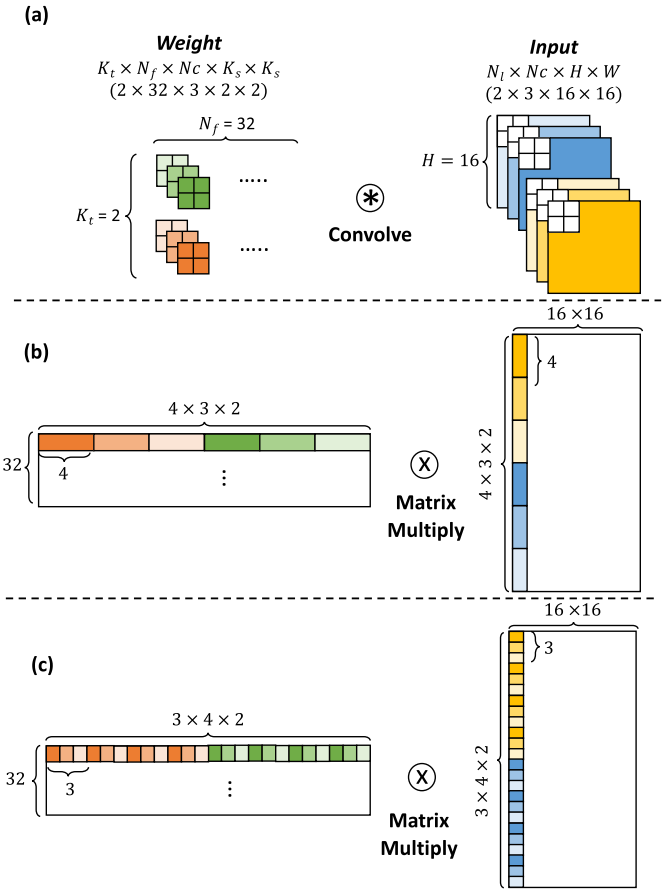


Fig. 5. 2-D convolution in different sequence. (a) Convolution. (b) Kernel-major MM. (c) Channel-major MM.

kernel-first matrix multiplication which performs the calculation belonging to one kernel first [4], [6]. To improve the generality of the accelerator for 2-D and 3-D CNNs with different kernel sizes, we rearrange the data and computational sequence to perform the matrix multiplication in a channel-first manner where the computation along the channel dimension is calculated first. Based on this computational manner, our hardware is designed to exploit channel and filter parallelisms by computing multiple channels PC and filters PF spatially, and mapping the computation in k_t, k_h, and k_w loops temporally, which brings two benefits: 1) the proposed design is general enough for convolutions with different kernel sizes.2) Both

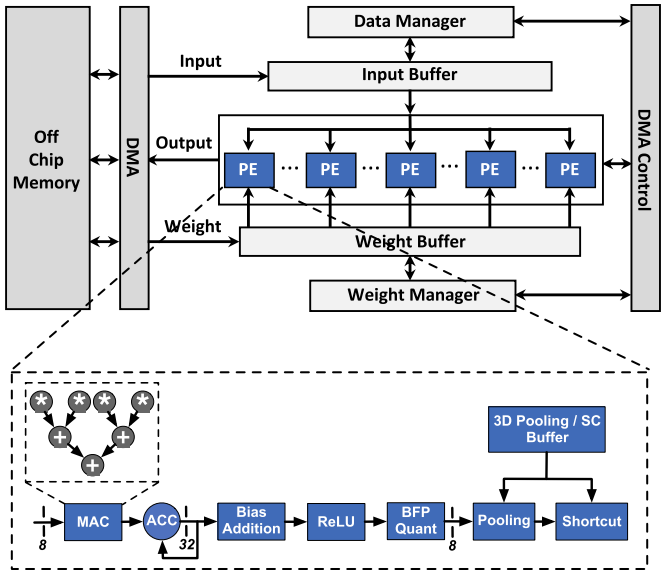


Fig. 6. Design overview of the proposed FPGA-based accelerator.

2-D and 3-D convolution are unified onto the one hardware design as all the computation related to the third dimension is performed temporally.

B. Hardware Design

1) Architecture Overview: An overview of our proposed accelerator is presented in Fig. 6. It is mainly composed of off-chip memory, an input buffer, a weight buffer, and PF numbers of PEs which compute PF filters in parallel as mentioned in Section IV-A. The PF is a reconfigurable hardware parameter, which is optimized for different CNN models by the automatic tool introduced in Section V. The input data are shared among different PEs, cached in the input buffer, and controlled by a data manager. The weight manager transfers the weights of different filters from the weight buffer to each PE accordingly to perform convolution. The outputs generated by PEs are transferred back to the off-chip memory directly in order to decrease the on-chip memory consumption. In this manner, the weights and intermediate results of each layer only require to be loaded from off-chip memory to on-chip buffer once to perform one feedforward pass. To overlap the time spent on loading the weights of each layer, the double buffer technique is used in the weight buffer. The PE is designed to perform a sequence of layers in a pipeline, which is shown in Fig. 6. It mainly consists of a *MAC*, *ReLU*, BFP quantization (*BFP Quant*), pooling, and shortcut (*SC*) modules and one buffer.

2) 2-D and 3-D Convolution: The computation of 2-D and 3-D convolution is performed in *MAC* units. Each *MAC* contains PC numbers of multipliers followed by a $\log_2 PC$-level adder tree. As mentioned in Section IV-A, the PC is a hardware parameter similar to PF, which can be reconfigured for different CNN models. Because the static BFP quantization uses the fixed shared exponents during runtime, we only need to perform computation on the corresponding mantissa parts. In addition, as the weights are blocked along the filter

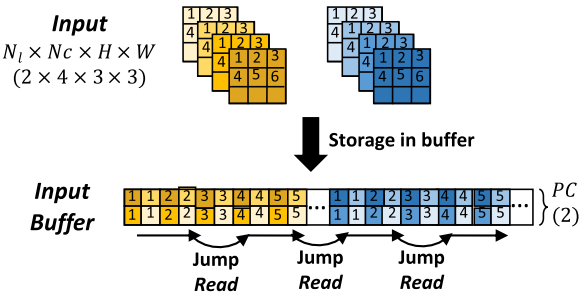


Fig. 7. Data storage in input buffer and data access pattern.

dimension according to our blocking strategy mentioned in Section III-B, the intermediate results coming from different spatial positions and channels can be accumulated together. Therefore, after the last level of the adder tree, there is a fixed-point accumulator designed for channel accumulation. To avoid overflow, the precision of the accumulator is 32 bits with the 1-bit sign bit, 16-bit fractional bit, and 15-bit integer bit. Note that the bitwidth of inputs, weights, and outputs are reconfigurable to meet different users' needs. A control signal is connected to the accumulator, which is used to indicate when the results are ready. To keep the bitwidth of the next layer's inputs as 8-bit, we design a BFP quantization module (BFP-Quant) before applying the pooling operation, which maps the 32-bit accumulated outputs to 8 bits. The implementation details of the BFP-Quant module will be shown in Section IV-B4.

As each PE receives PC numbers of pixels from different channels and all PEs share the same inputs, the data are stored in the input buffer in the manner as shown in Fig. 7. For input data with 2 frames, 3 channels, and 3×3 spatial size, the first-frame data belonging to the same spatial position from different channels are first stored sequentially, followed by the data from the next spatial position in the same frame. The data of the second frame are stored in the same manner after the storage of the first frame. Therefore, the input buffer can output PC number of pixels from different PC channels each time to PEs as required by our computation pattern.

3) 2-D and 3-D Pooling: The pooling operations include max pooling and average pooling, which are implemented using comparators and adders followed by a 16-bit divider. Because 2-D pooling is applied on spatial dimension and the data belonging to the same channel share one exponent under our blocking scheme, there is no need for exponent alignment. Therefore, 2-D pooling can be simply performed in the corresponding mantissa part without using extra operations. When it comes to 3-D pooling, the buffer in the PE is utilized to cache the data from adjacent frames. An example of 3-D pooling with $K_t = 3$ is shown in Fig. 8. In the beginning, the result of the first pooling is cached in the buffer. While processing the second frame, the 3-D pooling is performed between the newly generated second-frame results and the first-pooled data cached in the buffer. The pooling results are then flown back to the buffer again and overwrite the first-pooled data in the buffer. In the end, the pooling is performed between the third frame results and the second-pooling data, and the results are transferred to the off-chip memory.

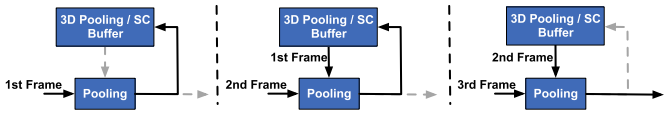


Fig. 8. Data flow of 3-D Pooling when K_t is 3.

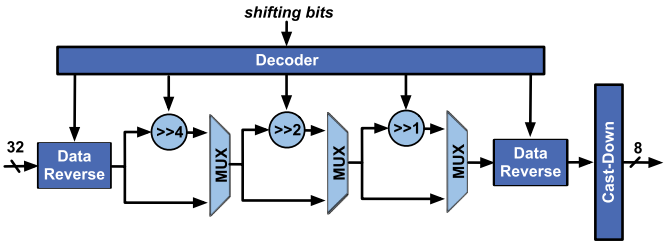


Fig. 9. Hardware design of BFP quantization module.

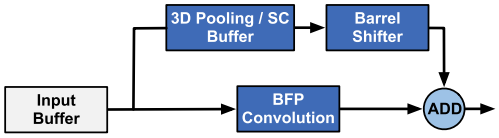


Fig. 10. BFP shortcut.

4) BFP Quantization: Fig. 9 presents the hardware design of our BFP quantization module. Because the shared exponents may vary in different layers, we need to align the outputs generated from the current convolutional layer using the shared exponents belonging to the next layer. In addition, since the shared exponents of the current layer may be greater or smaller than the one used in the next layer, our BFP quantization module needs to support both right and left shifting operations. In this article, we adopt the mask-based data-reversal barrel shifter design [39] to support both right and left shift with any given shift amounts parsed by a decoder. To truncate the 32-bit shifted results to 8 bits, we put a cast-down module at the end of the BFP quantization module to generate 8-bit results. The cast-down module simply extracts the higher 8 bits from the 32-bit shifted results.

5) Shortcut Addition: The SC is first proposed in [16]. Since the exponents of the second convolutional layer's outputs may be different from the original inputs as introduced in Section III-C2, it is required to perform the bit shifting in the shortcut module. Therefore, we design a barrel shifter to perform the required exponent alignment after the data buffer, which is shown in Fig. 10. When both of the tensors are aligned, the fixed-point SC can be simply performed on their mantissa parts.

C. Hardware Optimization

To improve the hardware efficiency, it is important to keep the computational resources as busy as possible for useful calculation. However, we observed that when the channel number is small, especially in the very initial layers where the channel number is 3, most of the multipliers and adders are idle because PC is always set as a relatively large value, such as 32 or 64. To address the hardware inefficiency caused by small channel concurrence in some layers, we design our *MAC* in each PE to support two modes with different combinations of parallelism, which is shown in Fig. 11.

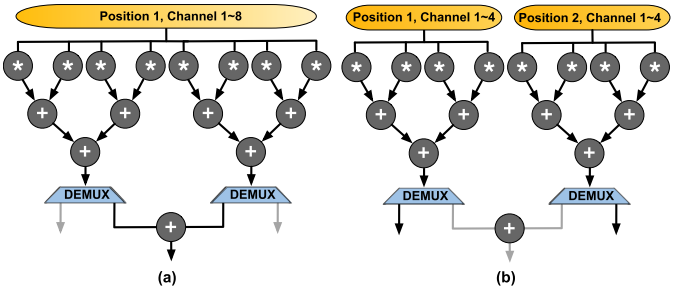


Fig. 11. *MAC* design with different two modes. (a) MAC under PC mode. (b) MAC under $PC\&PS$ mode.

In each PE, we partition the *MAC* design into several subtrees and put a *DEMUX* module at the tail of each subtree. When the channel number N_c is greater and divisible by PC, the *MAC* is configured as PC mode where the results from each subtree are aggregated together by controlling the *DEMUX* modules. When the channel number is smaller than PC, we exploit the parallelism in spatial dimension by feeding the data from different positions into different subtrees. Then, the *MAC* design is configured as $PC\&PS$ mode using *DEMUX*s and the results from each subtree are outputted directly without going through the adder at the tail of the *MAC* unit. By switching between two different modes for different layers, the hardware efficiency is improved even when the number of channels is small in some very initial layers of CNNs.

V. Automatic Tool

This section proposes a tool that automatically optimizes software and hardware hyperparameters for different CNNs and datasets. Several techniques, such as fake quantization, resource model, and latency model, will be presented to speed up the optimization process.

A. Overview of Tool

As mentioned in Section-III, there are different software and hardware hyperparameters in our static BFP quantization and hardware design. These hyperparameters can be reconfigured for different CNNs to improve the accuracy and hardware performance. Therefore, an automatic tool is proposed in this article to perform the hardware and software optimizations, which is shown in Fig. 12. The inputs of the tool include the network architecture of the target CNN model, the resource information of the underlying hardware platform, and the validation dataset used for evaluation. Since the main optimization is implemented using PyTorch [40], we use ONNX [41] tool at the beginning to convert the models generated from other frameworks, such as TensorFlow [42] and MXNet [43], to PyTorch models. There are two optimizations included in our automatic tool: 1) hardware optimization, which determines the hardware parameters, such as parallelism level including PC and PF, according to the resource model, latency model, and the available resources of the hardware platform. 2) Software optimization, which determines the

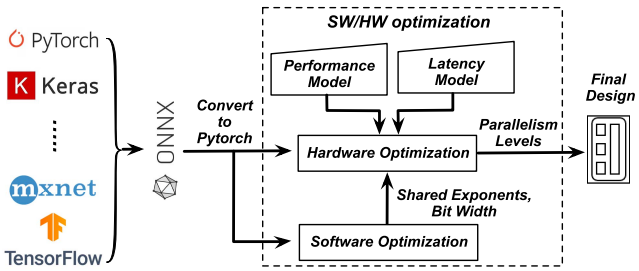


Fig. 12. Overview of automatic tool.

software parameters, such as the shared exponents, for each layer to optimize the accuracy of the given dataset.

B. Hardware Optimization

For a given CNN model, we optimize the hardware performance a by iterating all the possible hardware parameters $\mathcal{A}$ and use greedy algorithm to determine the optimal hardware design, which can be formulated as

$$\min_{a} \text{LAT}(a), \quad \{a \in \mathcal{A} \mid \text{RES}(a) \leq \text{RES}_{\text{avl}}\}. \tag{1}$$

The $\text{LAT}(a)$ and $\text{RES}(a)$ represent the latency model and resource model, respectively, which are described as follows.

1) Resource Model: To estimate the resource consumption of the proposed FPGA-based accelerator, we introduce a resource model for the logic and memory resources. Table I summarizes the notation used in this article.

The logic resources include digital signal processor (DSP) blocks and other logic elements, such as LUTs or ALMs. In our design, we use one DSP together with some logic elements to implement two 8-bit multipliers. To save DSP resources, the adders are implemented using only logic elements. Since DSPs are the limiting resource for FPGA-based CNN accelerator [9], we only consider DSP consumption in this article. The DSP consumption can be described by

$$\text{DSP}_{\text{total}} = (PC * \text{PF})/2. \tag{2}$$

The memory resources are mainly consumed by the input buffer and weight buffer. As the input buffer needs to cache the input feature maps from K_t frames in the current ith layer, its usage can be represented as

$$\text{MEM}_{\text{in}} = \max_{i=1,\ldots,l} \left(N_c^i * H^i * W^i * K_t^i\right) * \text{DW}. \tag{3}$$

In terms of the weight buffer, it only needs to cache the current PF filters with PC channels, so the memory consumption can be formulated as

$$\text{MEM}_{\text{weight}} = \max_{i=1,\ldots,l} \left(N_c^i * \text{PF} * PC * K_s^i * K_s^i * K_t^i\right) * \text{DW}. \tag{4}$$

As the use of ping-pong buffer technique, the total memory consumption is

$$\text{MEM}_{\text{total}} = 2 * \left(\text{MEM}_{\text{in}} + \text{MEM}_{\text{weight}}\right). \tag{5}$$

2) Latency Model: Since we run the network on the proposed accelerator layer-by-layer, the total latency $\text{LAT}_{\text{total}}$ can be calculated using $\sum_{i=0}^{N} \text{LAT}_i$, where N represents the total number of layers and LAT_i denotes the latency consumed in the ith layer. Based on the computational pattern introduced in Section IV-A, it takes totally $\text{BF} = N_f/\text{PF}$ batches for the accelerator to finish the computation, and each batch produces $\text{PF} \times H_{\text{out}} \times W_{\text{out}}$ output pixels. Therefore, LAT_i can be formulated as $\sum_{j=0}^{\text{BF}} \text{LAT}_{i,j}$, where j denotes the jth batch.

The latency $\text{LAT}_{i,j}$ is composed of three parts: 1) $\text{LAT}_{i,j}^{\text{load}}$—time of loading the activations and weights from off-chip memory to the on-chip buffers. 2) $\text{LAT}_{i,j}^{\text{comp}}$—time used for the computation required by the jth batch in layer i. 3) $\text{LAT}_{i,j}^{\text{store}}$—time spent in storing the outputs back to the off-chip memory. We calculate the latency in terms of clock cycles to achieve a more accurate estimation. These three parts can be formulated as follows.

1) Loading time, which consists of the weights loading time $\text{LAT}_{i,j}^{\text{weight}}$ and the input data loading time $\text{LAT}_{i,j}^{\text{input}}$

$$\text{LAT}_{i,j}^{\text{input}} = \frac{N_l^i \times N_c^i \times W^i \times H^i}{\text{BW} \times \text{EFF}_{\text{io}} \times \text{CLK}_{\text{io}}} \tag{6}$$

$$\text{LAT}_{i,j}^{\text{weight}} = \frac{K_s^i \times K_s^i \times K_t^i \times N_c^i \times \text{PF}}{\text{BW} \times EFF_{\text{io}} \times \text{CLK}_{\text{io}}}. \tag{7}$$

Because we cache the input data in the on-chip buffer for data reuse, the input data loading time only exists when considering the first batch, which can be formulated as

$$\text{LAT}_{i,j}^{\text{load}} = \begin{cases} \text{LAT}_{j=1,i}^{\text{load}} & = \text{LAT}_{i,j}^{\text{input}} + \text{LAT}_{i,j}^{\text{weight}} \\ \text{LAT}_{j\neq 1,i}^{\text{load}} & = \text{LAT}_{i,j}^{\text{weight}} \end{cases} \text{into.} \tag{8}$$

2) Computational time. Since most of the computational time is occupied by the convolution operation (>99%), and the other operations (BN, Pool, ReLU, Pool, and SC) only consume a negligible portion of the total time, the computational time can be formulated as

$$\text{LAT}_{i,j}^{\text{comp}} = \frac{K_s^i \times K_s^i \times K_t^i \times N_c^i \times \text{PF} \times W^{i+1} \times H^{i+1}}{\text{PF} \times PC \times \text{CLK}_{\text{pe}}}. \tag{9}$$

3) Storing time, which is spent on transferring the outputs to the off-chip memory

$$\text{LAT}_{i,j}^{\text{store}} = \frac{\text{PF} \times W^{i+1} \times H^{i+1}}{\text{BW} \times \text{EFF}_{\text{io}} \times \text{CLK}_{\text{io}}}. \tag{10}$$

Since our accelerator is designed in the fully pipelined manner, these three parts are overlapped with each other, and the time spent on ith layer can be formulated as

$$\text{LAT}_i = \sum_{j=0}^{\text{BF}} \max\left(\text{LAT}_{i,j}^{\text{load}}, \text{LAT}_{i,j}^{\text{store}}, \text{LAT}_{i,j}^{\text{comp}}\right). \tag{11}$$

C. Software Optimization

As the design is based on static BFP arithmetic, the software optimization aims at finding the optimally shared exponent for each block to improve the accuracy. We perform the software optimization using a quantization tool, which is based on a fake BFP quantization technique.

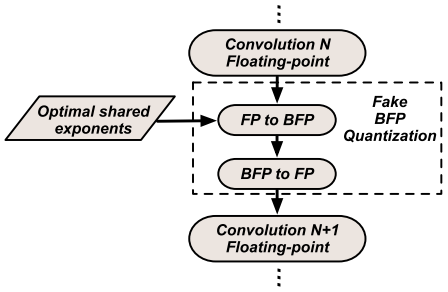


Fig. 13. Fake BFP quantization technique.

1) Fake BFP Quantization: PyTorch [40] is a machine learning framework, which provides high flexibility and generality. However, it cannot be directly used for our BFP calculation since most of the core operations, including both 2-D and 3-D convolutions, are implemented in floating-point arithmetic. In order to simulate the accuracy loss brought by the static BFP quantization using Pytorch, we propose a fake BFP quantization technique, which is shown in Fig. 13.

For a CNN model using static BFP quantization, the precision loss exists in both weights and activations. To capture the precision loss that existed in quantized activations, a fake BFP quantization layer, which contains two data type conversion operations, is placed between every two adjacent convolutional layers. The first conversion operation converts the FP data into BFP numbers, where the quantization error of activations is generated. Then, to output the results in FP format, the second conversion operation maps BFP numbers back to FP outputs such that the FP operations provided by the PyTorch library can be still used in the rest of the CNN layers. Note that the quantization error simulated in the first conversion layer will be kept because the FP numbers after the second conversion only represent the FP values quantized by BFP. The same fake quantization layers will be applied on weights to simulate quantization error. However, unlike activations that change over time, the weights are fixed during the inference. Therefore, it only requires applying fake quantization on weights once before the inference. Then the BFP quantized weights will be stored in off-chip memory and loaded to on-chip buffers during runtime. Note that these fake BFP quantization layers, for both activations and weights, are only applied during the software simulation. In the real hardware design and execution, these layers will be removed.

2) Quantization Tool: The workflow of our automatic tool[1] is presented in Fig. 14. The main computation is implemented using the high-performance FP functions provided by PyTorch, which guarantees the speed of the tool. The inputs of the tool include example images, the pretrained CNN models, the basic BFP configurations (exponent and mantissa bits), and the validation dataset. Then, the tool uses maximum and KL divergence strategies to find two sets of shared exponents as mentioned in Section III. According to these shared exponents, we construct the BFP quantized models by inserting the fake BFP quantization layers between every two adjacent convolutions in the FP models. By performing the inference using the BFP-quantized CNN models based on two sets of shared exponents, the final accuracy is obtained and the shared exponents with higher accuracy will be produced as outputs for the user.

[1]Our quantization tool is publicly available at: https://github.com/os-hxfan/Static_BFP_CNN

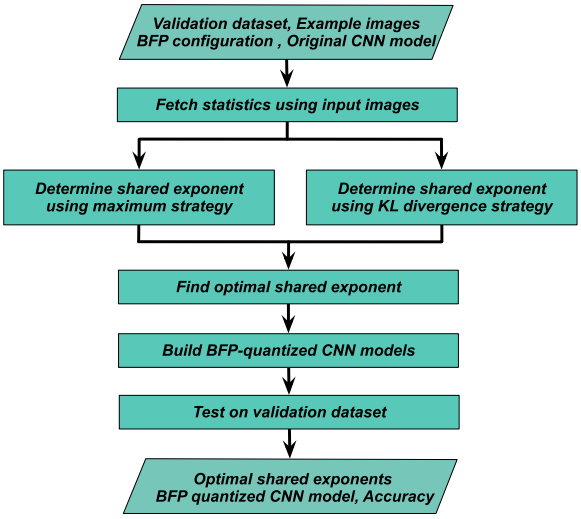


Fig. 14. Workflow of the BFP quantization tool.

TABLE II
BENCHMARK 2-D AND 3-D CNN MODELS

	Network	Workloads (GOP)	Model Size (MB)
2D CNN	*ResNet–50* [16]	7.7	98
	VGG–16 [32]	30.9	528
	InceptionV4 [45]	27.6	46
	MobileNetV2 [35]	0.6	14
3D CNN	*C3D* [3]	144.3	321
	R3D-18 [46]	87.4	136
	R3D-34 [46]	156.7	261

VI. EXPERIMENTAL RESULTS AND DISCUSSION

To evaluate the accuracy and hardware performance, various 2-D and 3-D CNNs are used as benchmark models in our experiments, which are listed in Table II. We evaluate 2-D CNNs on ImageNet [31] dataset with 1000 object categories for image classification and 3-D CNNs on UCF101 [44] dataset with 101 human action classes for video recognition. Intel Arria 10 GX1150 is used as a platform for hardware implementation and Quartus 17 Prime Pro is used for synthesis. All the design space explorations are performed at the software level using our automatic framework, while the final accuracy and hardware performance are evaluated at board level.

A. Accuracy Comparison

We conduct four experiments in this part for accuracy comparison. We first evaluate the accuracy of our static BFP quantization with different exponent and mantissa bits. The second

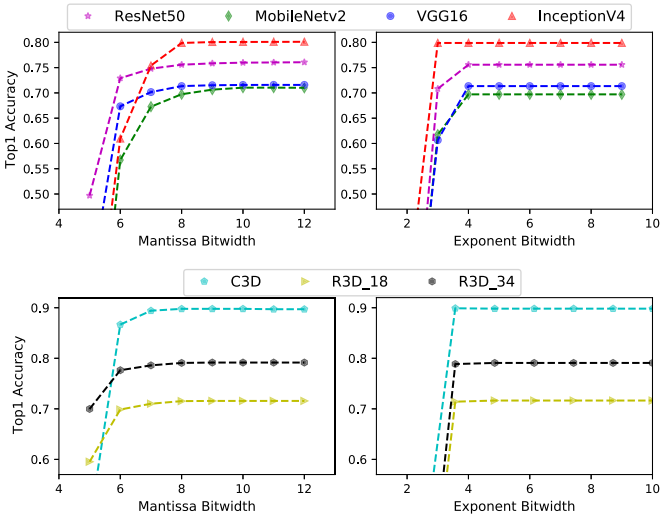


Fig. 15. Accuracy of 2-D (the first row) and 3-D (the second row) CNNs under different mantissa and exponent bitwidth.

experiment explores the effectiveness of different strategies while finding the shared exponents. Then, we demonstrate the software efficiency of our automatic tool. In the fourth experiment, we compare the dynamic BFP quantization [13] and the integer linear quantization [12] with our proposed static BFP quantization.

1) Bitwidth Exploration: This experiment is used to explore the relationship between the bitwidth and the accuracy performance. We first evaluate our benchmark 2-D and 3-D CNNs using different mantissa bitwidths ranging from 4 to 13. To eliminate the effect of exponent bits, the bitwidth of the exponent is set to be 8 for controlling variates. It can be clearly seen from Fig. 15 that the accuracy keeps steady when the mantissa bits are greater than 8-bit. A significant accuracy drop can be observed when the bitwidth of the mantissa is smaller than 6-bit. On *MobileNetV2* and *InceptionV4*, the accuracy reduces remarkably by 15% and 20% when the mantissa bits are decreased from 8 to 6. Therefore, to maintain the accuracy, we set the bitwidth of mantissa as 8.

To find the proper bitwidth of exponents, the benchmark CNN models are evaluated using different exponent bitwidths. The mantissa bits are set to be 8 for controlling variates. According to the results shown in Fig. 15, the accuracy almost keeps the same when the exponent bits are greater than 3. Therefore, based on the experimental results shown in this part, we set the exponent and mantissa bits to be 4 and 8, respectively, like the tradeoff between hardware performance and accuracy in the rest of the experiments.

2) Maximum Versus KL Divergence: We evaluate our benchmark CNNs using different strategies while determining shared exponents, and the results are shown in Table III. It can be clearly seen that the KL divergence strategy shows higher accuracy on all the 2-D CNNs. In particular, it shows significant accuracy improvement on MobileNetV2 (+3.406%) of using the KL divergence strategy. However, maximum strategy shows higher accuracy on *C3D* and *R3D-34*. Therefore, the optimal strategies of achieving higher accuracy vary on different CNNs and it will be optimized by our tool (Section V).

TABLE III
ACCURACY OF THE STAIC BFP QUANTIZATION USING DIFFERENT STRATEGIES AND TIME COST OF TOOL

	Maximal Exponent		Minimizing KL Divergence	
	Accuracy	Time Cost (second)	Accuracy	Time Cost (second)
ResNet–50	75.308%	32.38	**75.758%**	30.56
VGG–16	71.24%	56.13	**71.260%**	50,96
InceptionV4	79.766%	43.26	**79.856%**	38.59
MobileNetV2	66.308%	68.67	**69.714%**	61.97
C3D	**90.596%**	9.24	89.930%	8.96
R3D-18	71.307%	8.57	**71.567%**	7.23
R3D-34	**78.860%**	14.16	69.456%	11.63

3) Software Efficiency: On an Intel Xeon Silver 4110 CPU, we measure the time cost of the quantization process. The results are shown in Table III. Since it does not require any time-consuming retraining or finetuning, our tool only consumes a few minutes to finish the whole optimization process, which demonstrates the high software efficiency.

4) Comparison With Other Quantization Approaches: We compare our approach against three state-of-the-art quantization methods, i.e., dynamic BFP quantization [13], integer linear quantization [12], and logarithm quantization [11]. To demonstrate our quantization scheme is also applicable on complex artificial intelligence tasks, we evaluate the object detection using a more complicated CNN model named single shot detector [2] (*SSD*) on the FDDB [47] dataset with 5171 human faces. We set the bitwidths of exponent and mantissa as 4 and 8 for both static and dynamic BFP quantization. For a fair comparison, all the quantization methods use 8 bits on activations. Table IV presents the accuracy results.[2] Among these quantization schemes, only logarithm quantization requires time-consuming retraining to recover its accuracy performance. Except for *MobileNetV2*, the accuracy loss of our static BFP quantization on all these models is less than 0.5%. Although the static BFP quantization on *MobileNetV2* suffers nearly 2% accuracy loss, it still performs better than the integer linear quantization and BFP dynamic quantization. On *InceptionV4*, *VGG-16*, and *ResNet-50*, dynamic BFP quantization shows higher accuracy than static BFP quantization. However, the frequent FP-BFP and BFP-FP conversions in dynamic BFP quantization place a heavy burden on the hardware performance and resources consumption, which is not suitable for FPGA implementation. Comparing with the integer linear quantization [12], our static BFP quantization demonstrates the higher accuracy on all the 2-D CNN models. However, since the accuracy of the integer linear quantization on 3-D CNNs is not reported [12], we are not able to compare our method with integer linear quantization on 3-D CNNs. Compared with the logarithm quantization [11], our approach achieves higher

[2]The per-class accuracy is available at: https://github.com/os-hxfan/Static_BFP_CNN

TABLE IV
ACCURACY OF CNN MODELS UNDER DIFFERENT QUANTIZATION SCHEMES

	Re-train	*ResNet-50*	*VGG–16*	*InceptionV4*	*MobileNetV2*	*C3D*	*R3D-18*	*R3D-34*	*SSD*
Floating Point	No	76.13%	71.592%	80.118%	71.768%	90.596%	71.529%	78.971%	83.5%
Logarithm Quant [11]	Yes	74.81%	70.82%	–	–	–	–	–	–
Linear Quant [12]	No	74.34%	71.412%	75.84%	69.668%	–	–	–	83.5%
Dynamic BFP [13]	No	76.408%	71.516%	79.802%	69.238%	89.930%	71.196%	77.231%	83.3%
Static BFP	No	75.758%	71.260%	79.856%	69.714%	90.077%	71.566%	78.860%	83.4%

TABLE V
RESOURCE CONSUMPTION OF THE CONVOLUTIONAL KERNEL MODULE USING DIFFERENT QUANTIZATION ON INTEL ARRIA 10 GX1150

	ALMs	DSPs
Static BFP Quantization	58,368	1,345
Integer Quantization with zero-point [12]	117,632	1,473
Total Savings	59,264	128
Savings Percentage	50.3%	8.7%

TABLE VI
RESOURCE CONSUMPTION OF THE WHOLE DESIGN ON ARRIA 10 GX1150

Resources	ALMs	Registers	DSPs	M20K
Used	263,569	901,673	1,345	2,334
Total	427,200	1,708,800	1,518	2,713
Utilization	61%	53%	88.6%	86%

accuracy on both *ResNet-50* and *VGG–16*. In addition, our approach shows higher generality since logarithm quantization does not evaluate on 3-D CNNs and lightweight models, such as *MobileNetV2*.

B. Resource Efficiency

To demonstrate the kernel design based on the static BFP quantization has high hardware efficiency, we implement the BFP-based convolutional kernel (Section IV) using Verilog on an Intel Arria 10 GX1150 FPGA. As presented in Section IV, only dynamic BFP quantization [13] and integer linear quantization [12] can achieve comparable accuracy with our approach. Therefore, the analysis in this part only considers dynamic BFP and linear quantization. Compared with dynamic BFP quantization, since our static BFP quantization eliminates the frequent data conversion between BFP and FP, it saves a large number of resources in implementing BFP-FP convertors. To compare with [12], we implement a convolutional kernel based on integer linear quantization on the same hardware platform. The PC and PF are set as 64 on both of the designs. We implement the adders using ALMs and multipliers by both DSPs and ALMs as mentioned in Section V.

Table V presents the synthesis results of both kernel designs. In comparison with the integer linear quantization, our approach saves the DSP usage by 5.8%. The savings come from the implementation of the barrel shifter module which uses logic resources to quantize the accumulated 32-bit results into 8-bit outputs. At the same time, our static BFP quantization uses nearly two times fewer ALMs than the integer linear quantization. The large ALMs savings come from nonzero-point representation in static BFP quantization. Therefore, our quantization is more hardware-friendly than integer linear quantization.

To measure the hardware performance and resource usage of the whole system, we implement all hardware modules on the same FPGA (Intel Arria 10 GX1150) using Verilog with both PC and PF being 64. The hardware is clocked at 220 MHz. A dual-core ARM Cortex-A9 processor (1.5 GHz) is installed on the platform, which is used to configure the parameters of each layer while running the network. A 2-GB DDR4 memory is installed as the off-chip memory to store the weights and intermediate results of CNNs. Quartus 17 Prime Pro is used for synthesis and implementation. The resource utilization of the final design is presented in Table VI. The DSP and memory resources become the limiting resources in our design, which consume over 80% of the total resources.

C. Hardware Performance

We evaluate benchmark CNN models on our accelerator with respect to latency, energy consumption, and hardware efficiency, which is presented in Table VII. The parallelism levels, PC and PF, are optimized for different models using our tool introduced in Section V. Since *MobileNetV2* contains depthwise convolution which does not exhibit concurrence in the channel dimension, the design chooses a relatively small PC and large PF for higher performance. We also measure the computational efficiency using MAC efficiency and the energy efficiency using giga-operations per second per watt (GOP/s/W).

As shown in Table VII, in most of models which do not use depthwise convolutions, the throughput of our accelerator is 1.33∼1.66 TOP/s (tera-operations per second) and MAC efficiency is 78.5%∼92.4% depending on different CNN models. The high computational efficiency comes from several factors.

1) Without frequent BFP and FP conversions, the proposed static BFP quantization uses low-precision data in both computation and storage, which efficiently utilizes the computational, bandwidth, and memory resources.
2) The hardware architecture supports four categories of fine-grained parallelisms, including PS, PK, PC, and PF, which fully utilize the extensive concurrence exhibited by both 2-D and 3-D CNNs.

TABLE VII
HARDWARE PERFORMANCE OF OUR DESIGN ON DIFFERENT 2-D AND 3-D CNN MODELS

Task	Dataset	Model	$\{PC, PF\}$	Latency (ms)	Throughout (GOP/s)	Energy Efficiency (GOP/s/W)	MAC Efficiency
Image Classification	*ImageNet*	*ResNet–50*	$\{64, 64\}$	4.62	1667	37.0	92.4%
		VGG–16	$\{64, 64\}$	23.18	1335	29.7	74.0%
		InceptionV4	$\{64, 64\}$	19.38	1423	31.6	78.9%
		MobileNetV2	$\{32, 128\}$	0.71	820	18.2	47.0%
Object Detection	*FDDB*	*SSD*	$\{64, 64\}$	3.21	1637	36.0	91.1%
Human Action Recognition	*UCF101*	*C3D*	$\{64, 64\}$	93.95	1536	34.1	85.2%
		R3D-18	$\{64, 64\}$	56.67	1541	34.2	85.5%
		R3D-34	$\{64, 64\}$	111	1416	31.5	78.5%

TABLE VIII
PERFORMANCE COMPARISON OF OUR FINAL FPGA DESIGN VERSUS CPU AND GPU PLATFORMS

	CPU		GPU		Our Work	
Platform	Intel Xeon E5-2680 v2		TITAN X Pascal		Intel Arria 10 GX1150	
Frequency	2.8 GHz		1.53 GHz		220 MHz	
Technology	22 nm		16 nm		20 nm	
Acceleration Library	MKLDNN		CuDNN, PyTorch 1.9.0		-	
Power (W)	135		248		45	
Model	*ResNet–50*	*R3D-18*	*ResNet–50*	*R3D-18*	*ResNet–50*	*R3D-18*
Latency (ms)	323.99	309.77	5.24	17.16	4.62	56.67 (45.57)
Throughput (GOP/s)	24	282	1470	5089	1667	1541
Energy Efficiency (GOP/s/W)	0.18	2.09	6.59	20.5	37.0	34.2

TABLE IX
PERFORMANCE COMPARISON OF OUR FINAL FPGA DESIGN VERSUS OTHER FPGA DESIGNS

	Ma *et al.* [48] (ICCAD 2018)	Fan *et al.* [6] (FPL 2018)	Venieris *et al.* [21] (TNNLS 2018)	Zhang *et al.* [49] (ICCAD 2018)	Shen *et al.* [14] (FPGA 2018)		Yu *et al.* [24] (TCAD 2019)	Our Work			
Platform	Intel Arria GX1150	Xilinx ZC706	Xilinx Zynq-7045	Xilinx ZC706	Xilinx VUS440		Xilinx ZU9	Intel Arria GX1150			
Technology	20 nm	28 nm	28 nm	28 nm	20 nm		16 nm	20 nm			
Frequency (MHz)	240	200	125	200	200		330	220			
DSPs	1518	780	900	680	1536		1542	1345			
Logic (ALMs/LUTs)	175K	83K	218K	114K	209K		433K	427K			
Model	*SSD*	*C3D*	*VGG16*	*VGG16*	*VGG16*	*C3D*	*ResNet-50*	*ResNet-50*	*VGG16*	*SSD*	*C3D*
Accuracy (Top-1)	76.94%	84.8%	–	63.7%	–	–	–	75.758%	71.260%	83.4%	90.077%
Throughput (GOP/s)	1032	159	124	524	821	785	1380	1667	1335	1637	1536
Resource Efficiency (GOP/s/DSP)	0.680	0.204	0.14	0.77	0.535	0.511	0.895	1.24	0.992	1.22	1.14

3) By analyzing the computation of a variety of 2-D and 3-D CNNs, the unified computational pattern proposed in this article to improve the resource efficiency.
4) By utilizing the reconfigurability of our accelerator, the automatic tool is able to deeply optimize the hardware designs for different CNN models case by case.

D. Performance Comparison

1) Comparison With CPU and GPU: We also compare our FPGA-based design with CPU and GPU implementations, which is shown in Table VIII. *ResNet–50* and *R3D-18* are chosen as our benchmark models to represent 2-D and 3-D CNNs, respectively. The batch size on all three implementations is set to be one for a fair comparison. Compared with the CPU implementation, our accelerator achieves 6–70 times higher throughput on both 2-D and 3-D CNNs. Comparing with the GPU implementation, our design is 1.5 times more energy efficient. Although GPU is faster on *R3D-18* using the 16-nm technology, our FPGA design (20 nm) can also achieve 45.57 ms when we scale the performance to the 16-nm technology by $16/20$ times.

2) Comparison With Other FPGA Designs: Table IX presents the comparison results between our accelerator with the state-of-the-art FPGA designs in terms of latency and throughput. Because these designs are implemented on different platforms with different hardware resources, their DSP consumption may vary from each one. Therefore, we measure the GOP/s/DSP of these designs for a fair comparison. The GOP/s/DSP is the platform-independent metric to evaluate the quality of hardware architecture, which represents the computing ability provided by one DSP. Compared with previous designs, our accelerator supports a wider range of benchmark models, including different 2-D and 3-D CNNs, which demonstrates its higher generality. At the same time, our accelerator also shows higher throughput and resource efficiency than all these state-of-the-art designs. In comparison with [24] which has the highest performance on *ResNet-50*, our design can achieve higher throughput and nearly 1.4 times higher resource efficiency. Note that [24] can only support 2-D CNNs and its performance for 3-D CNNs is unknown. Comparing with the unified accelerator which supports both 2-D and 3-D CNNs, we can achieve 1.6–2 and 1.9–2.2 times higher throughput and resource efficiency depending on different CNN models.

VII. Conclusion

This work proposes a uniform hardware architecture to accelerate both 2-D and 3-D CNNs with high hardware efficiency. The design is based on a hardware-friendly quantization method call static BFP. The proposed static BFP eliminates the frequent representation conversions required in traditional dynamic BFP arithmetic. Without using zero point, static BFP can achieve up to 50% logic resources saving on an FPGA compared with conventional integer linear quantization. Extensive experiments on various 2-D and 3-D CNNs demonstrate that the static BFP can decrease the bitwidth of mantissa to 8 with negligible accuracy loss. An automatic tool is also proposed to optimize the accuracy and hardware performance by determining the proper software and hardware parameters. Our hardware design together with optimizations achieves 3.8–5.6 times higher energy efficiency than GPU implementation. Compared with the state-of-the-art FPGA-based accelerators, our design can also achieve up to 1.4–2.2 times higher resource efficiency and higher generality on both 2-D and 3-D CNNs. Further work includes extending our hardware design to support the transformer and other neural networks and exploring the mixed-precision BFP on these networks to further improve the performance.

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2012, pp. 1097–1105.

[2] W. Liu *et al.*, "SSD: Single shot multibox detector," in *Proc. Eur. Conf. Comput. Vis.* Cham, Switzerland: Springer, 2016, pp. 21–37.

[3] D. Tran, L. Bourdev, R. Fergus, L. Torresani, and M. Paluri, "Learning spatiotemporal features with 3D convolutional networks," in *Proc. IEEE Int. Conf. Comput. Vis. (ICCV)*, Dec. 2015, pp. 4489–4497.

[4] H. Fan, X. Niu, Q. Liu, and W. Luk, "F-C3D: FPGA-based 3-dimensional convolutional neural network," in *Proc. 27th Int. Conf. Field Program. Log. Appl. (FPL)*, Sep. 2017, pp. 1–4.

[5] H. Lu, H. Wang, Q. Zhang, S. W. Yoon, and D. Won, "A 3D convolutional neural network for volumetric image semantic segmentation," *Proc. Manuf.*, vol. 39, pp. 422–428, Jan. 2019.

[6] H. Fan, H.-C. Ng, S. Liu, Z. Que, X. Niu, and W. Luk, "Reconfigurable acceleration of 3D-CNNs for human action recognition with block floating-point representation," in *Proc. 28th Int. Conf. Field Program. Log. Appl. (FPL)*, Aug. 2018, pp. 287–2877.

[7] H. Fan *et al.*, "F-E3D: FPGA-based acceleration of an efficient 3D convolutional neural network for human action recognition," in *Proc. IEEE 30th Int. Conf. Appl.-Specific Syst., Architectures Processors (ASAP)*, Jul. 2019, pp. 1–8.

[8] E. Wang *et al.*, "Deep neural network approximation for custom hardware: Where we've been, where We're going," 2019, *arXiv:1901.06955*. [Online]. Available: https://arxiv.org/abs/1901.06955

[9] S. Liu, H. Fan, X. Niu, H.-C. Ng, Y. Chu, and W. Luk, "Optimizing CNN-based segmentation with deeply customized convolutional and deconvolutional architectures on FPGA," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 11, no. 3, pp. 1–22, Dec. 2018.

[10] Z. Liu, P. Chow, J. Xu, J. Jiang, Y. Dou, and J. Zhou, "A uniform architecture design for accelerating 2D and 3D CNNs on FPGAs," *Electronics*, vol. 8, no. 1, p. 65, Jan. 2019.

[11] D. Miyashita, E. H. Lee, and B. Murmann, "Convolutional neural networks using logarithmic data representation," 2016, *arXiv:1603.01025*. [Online]. Available: https://arxiv.org/abs/1603.01025

[12] B. Jacob *et al.*, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2018, pp. 2704–2713.

[13] 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 Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 27, no. 8, pp. 1874–1885, Aug. 2019.

[14] J. Shen, Y. Huang, Z. Wang, Y. Qiao, M. Wen, and C. Zhang, "Towards a uniform template-based architecture for accelerating 2D and 3D CNNs on FPGA," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, 2018, pp. 97–106.

[15] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," 2015, *arXiv:1502.03167*. [Online]. Available: https://arxiv.org/abs/1502.03167

[16] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2016, pp. 770–778.

[17] Z. Song, Z. Liu, and D. Wang, "Computation error analysis of block floating point arithmetic oriented convolution neural network accelerator design," in *Proc. 32nd AAAI Conf. Artif. Intell.*, 2018, pp. 1–8.

[18] B. Wu, Y. Wang, P. Zhang, Y. Tian, P. Vajda, and K. Keutzer, "Mixed precision quantization of ConvNets via differentiable neural architecture search," 2018, *arXiv:1812.00090*. [Online]. Available: https://arxiv.org/abs/1812.00090

[19] G. Lacey, G. W. Taylor, and S. Areibi, "Stochastic layer-wise precision in deep neural networks," 2018, *arXiv:1807.00942*. [Online]. Available: https://arxiv.org/abs/1807.00942

[20] K. Guo, S. Zeng, J. Yu, Y. Wang, and H. Yang, "A survey of FPGA-based neural network accelerator," 2017, *arXiv:1712.08934*. [Online]. Available: https://arxiv.org/abs/1712.08934

[21] S. I. Venieris and C.-S. Bouganis, "FPGAConvNet: Mapping regular and irregular convolutional neural networks on FPGAs," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 30, no. 2, pp. 326–342, Jul. 2018.

[22] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *Proc. 2015 ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, 2015, pp. 161–170.

[23] Y. Ma, Y. Cao, S. Vrudhula, and J.-S. Seo, "Optimizing the convolution operation to accelerate deep neural networks on FPGA," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 26, no. 7, pp. 1354–1367, Jul. 2018.

[24] Y. Xing *et al.*, "DNNVM: End-to-end compiler leveraging heterogeneous optimizations on FPGA-based CNN accelerators," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 10, pp. 2668–2681, Oct. 2020.

[25] M. Sun, P. Zhao, M. Gungor, M. Pedram, M. Leeser, and X. Lin, "3D CNN acceleration on FPGA using hardware-aware pruning," in *Proc. 57th ACM/IEEE Design Autom. Conf. (DAC)*, Jul. 2020, pp. 1–6.

[26] C. Yang, Y. Wang, X. Wang, and L. Geng, "WRA: A 2.2-to-6.3 TOPS highly unified dynamically reconfigurable accelerator using a novel Winograd decomposition algorithm for convolutional neural networks," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 66, no. 9, pp. 3480–3493, Sep. 2019.

[27] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding," 2015, *arXiv:1510.00149*. [Online]. Available: https://arxiv.org/abs/1510.00149

[28] A. Aimar *et al.*, "NullHop: A flexible convolutional neural network accelerator based on sparse representations of feature maps," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 30, no. 3, pp. 644–656, Mar. 2018.

[29] T. Liang, J. Glossner, L. Wang, S. Shi, and X. Zhang, "Pruning and quantization for deep neural network acceleration: A survey," 2021, *arXiv:2101.09671*. [Online]. Available: https://arxiv.org/abs/2101.09671

[30] M. Courbariaux, Y. Bengio, and J.-P. David, "Training deep neural networks with low precision multiplications," 2014, *arXiv:1412.7024*. [Online]. Available: https://arxiv.org/abs/1412.7024

[31] O. Russakovsky *et al.*, "ImageNet large scale visual recognition challenge," *Int. J. Comput. Vis.*, vol. 115, no. 3, pp. 211–252, Dec. 2015.

[32] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014, *arXiv:1409.1556*. [Online]. Available: https://arxiv.org/abs/1409.1556

[33] S. R. Jain, A. Gural, M. Wu, and C. H. Dick, "Trained quantization thresholds for accurate and efficient fixed-point inference of deep neural networks," 2019, *arXiv:1903.08066*. [Online]. Available: https://arxiv.org/abs/1903.08066

[34] H. Fan, G. Wang, M. Ferianc, X. Niu, and W. Luk, "Static block floating-point quantization for convolutional neural networks on FPGA," in *Proc. Int. Conf. Field-Program. Technol. (ICFPT)*, Dec. 2019, pp. 28–35.

[35] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted residuals and linear bottlenecks," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2018, pp. 4510–4520.

[36] Wikipedia Contributors. (2020). *Kullback-Leibler Divergence*. [Online]. Available: https://en.wikipedia.org/wiki/Kullback%E2%80%93Leibler_divergence

[37] K. He, X. Zhang, S. Ren, and J. Sun, "Identity mappings in deep residual networks," in *Proc. Eur. Conf. Comput. Vis.* Cham, Switzerland: Springer, 2016, pp. 630–645.

[38] H. Fan *et al.*, "A real-time object detection accelerator with compressed SSDLite on FPGA," in *Proc. Int. Conf. Field-Program. Technol. (FPT)*, Dec. 2018, pp. 14–21.

[39] M. R. Pillmeier, M. J. Schulte, and E. G. Walters III, "Design alternatives for barrel shifters," *Proc. SPIE*, vol. 4791, pp. 436–447, Dec. 2002.

[40] A. Paszke *et al.*, "PyTorch: An imperative style, high-performance deep learning library," in *Proc. Adv. Neural Inf. Process. Syst.*, 2019, pp. 8024–8035.

[41] *ONNX Framework*. Accessed: Feb. 1, 2021. [Online]. Available: https://github.com/onnx/onnx

[42] M. Abadi *et al.* (2015). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. [Online]. Available: https://www.tensorflow.org/

[43] T. Chen *et al.*, "MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems," 2015, *arXiv:1512.01274*. [Online]. Available: https://arxiv.org/abs/1512.01274

[44] K. Soomro, A. R. Zamir, and M. Shah, "UCF101: A dataset of 101 human actions classes from videos in the wild," 2012, *arXiv:1212.0402*. [Online]. Available: https://arxiv.org/abs/1212.0402

[45] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, Inception-ResNet and the impact of residual connections on learning," in *Proc. 31st AAAI Conf. Artif. Intell.*, 2017, pp. 1–7.

[46] K. Hara, H. Kataoka, and Y. Satoh, "Can spatiotemporal 3D CNNs retrace the history of 2D CNNs and ImageNet?" in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2018, pp. 6546–6555.

[47] V. Jain and E. Learned-Miller, "FDDB: A benchmark for face detection in unconstrained settings," Univ. Massachusetts, Amherst, MA, USA, Tech. Rep. UM-CS-2010-009, 2010.

[48] Y. Ma, T. Zheng, Y. Cao, S. Vrudhula, and J.-S. Seo, "Algorithm-hardware co-design of single shot detector for fast object detection on FPGAs," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Nov. 2018, pp. 1–8.

[49] X. Zhang *et al.*, "DNNBuilder: An automated tool for building high-performance dnn hardware accelerators for FPGAs," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Nov. 2018, pp. 1–8.

Hongxiang Fan received the B.S. degree in electronic engineering from Tianjin University, Tianjin, China, in 2017, and the master's degree from the Department of Computing, Imperial College London, London, U.K., in 2018, where he is currently pursuing the Ph.D. degree in machine learning and high-performance computing.

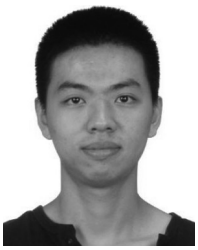
His current research interests include efficient algorithms and acceleration for machine learning applications.

Shuanglong Liu received the B.Sc. and M.Sc. degrees from the Department of Electronic Engineering, Tsinghua University, Beijing, China, in 2010 and 2013, respectively, and the Ph.D. degree in electric engineering from the Imperial College London, London, U.K, in 2017.

From 2017 to 2020, he was a Research Associate with the Department of Computing, Imperial College London. He is currently a Distinguished Professor with the School of Physics and Electronics, Hunan Normal University, Changsha, China. His current research interests include reconfigurable and high-performance computing for deep neural networks.

Zhiqiang Que received the B.S. degree in microelectronics and the M.S. degree in computing science from Shanghai Jiao Tong University, Shanghai, China, in 2008 and 2011, respectively. He is currently pursuing the Ph.D. degree with the Department of Computing, Imperial College London, London, U.K.

From 2011 to 2016, he worked on the microachitecture design and verification of ARM CPUs with Marvell Semiconductor Ltd., Shanghai. He is currently a Research Assistant with the Department of Computing, Imperial College London. His research interests include computer architectures, high-performance computing, and computer-aided design tools for hardware design optimization.



Xinyu Niu received the B.A. degree from Fudan University, Shanghai, China, in 2010, and the M.Sc. and D.Phil. degrees in computing science from Imperial College London, London, U.K., in 2011 and 2015, respectively.

He is currently a Co-Founder and the CEO of Shenzhen Corerain Technologies Company, Ltd., Shenzhen, China. His current research interests include developing applications and tools for reconfigurable computing that involve runtime reconfiguration.

Wayne Luk (Fellow, IEEE) received the B.A., M.Sc., and D.Phil. degrees in engineering and computing science from the University of Oxford, Oxford, U.K., in 1984, 1985, and 1989, respectively.

He founded and currently leads the Custom Computing Group, Department of Computing, Imperial College London, London, U.K., where he is also a Professor of computer engineering. He was a Visiting Professor with Stanford University, Stanford, CA, USA.

Dr. Luk is a fellow of the Royal Academy of Engineering and the British Computer Society (BCS). He had 15 papers that received awards from international conferences. He has been a member of the steering committee and the program committee of various international conferences. He received a Research Excellence Award from the Imperial College London. He was the Founding Editor-in-Chief of the *ACM Transactions on Reconfigurable Technology and Systems*.