An Efficient Hardware Accelerator for Structured Sparse Convolutional Neural Networks on FPGAs

Chaoyang Zhu, *Graduate Student Member, IEEE*, Kejie Huang, *Senior Member, IEEE*, Shuyuan Yang, *Student Member, IEEE*, Ziqi Zhu, *Student Member, IEEE*, Hejia Zhang, and Haibin Shen

***Abstract*— Deep convolutional neural networks (CNNs) have achieved state-of-the-art performance in a wide range of applications. However, deeper CNN models, which are usually computation consuming, are widely required for complex artificial intelligence (AI) tasks. Though recent research progress on network compression, such as pruning, has emerged as a promising direction to mitigate computational burden, existing accelerators are still prevented from completely utilizing the benefits of leveraging sparsity due to the irregularity caused by pruning. On the other hand, field-programmable gate arrays (FPGAs) have been regarded as a promising hardware platform for CNN inference acceleration. However, most existing FPGA accelerators focus on dense CNN and cannot address the irregularity problem. In this article, we propose a sparsewise dataflow to skip the cycles of processing multiply-and-accumulates (MACs) with zero weights and exploit data statistics to minimize energy through zeros gating to avoid unnecessary computations. The proposed sparsewise dataflow leads to a low bandwidth requirement and high data sharing. Then, we design an FPGA accelerator containing a vector generator module (VGM) that can match the index between sparse weights and input activations according to the proposed dataflow. Experimental results demonstrate that our implementation can achieve 987-, 46-, and 57-imag/s performance for AlexNet, VGG-16, and ResNet-50 on Xilinx ZCU102, respectively, which provides 1.5×–6.7× speedup and 2.0×–6.0× energy efficiency over previous CNN FPGA accelerators.**

***Index Terms*— Dataflow, deep convolutional neural networks (CNNs), field-programmable gate arrays (FPGAs), hardware accelerator, structured pruning.**

I. Introduction

THE remarkable performance improvement in various domains [1]–[3] achieved by convolutional neural networks (CNNs), such as AlexNet [4], VGG-16 [5], and ResNet [6], comes at the computational and data cost, which challenges both of on-chip storage and off-chip bandwidth in accelerator architecture design. Even though most of the operations in the CNN training and inference can be converted to matrix multiplication operations and accelerated with modern graphics processing units (GPUs), deploying CNNs on GPUs suffers from high power and area consumption. Customized accelerators have been regarded as a promising alternative, which is more flexible for considering the performance requirements and energy constraints [7]–[18].

Recently, the CNN pruning technique has been proved as an effective solution to reduce the computation and memory requirements of these models [19], [20]. For example, Han *et al.* [19] pointed out that pruning can lead to more than ten times the amount reduction of data with negligible accuracy loss. On the other hand, weight encoding, including quantization and entropy coding, has been proposed to further reduce the bit width of each weight, e.g., 4-bit per weight for AlexNet [20]. Unstructured pruning techniques, such as deep compression [20], have the weaknesses of imbalanced load and high irregularity. Therefore, structured pruning techniques [21]–[24] were proposed, which are more hardware friendly with a slightly lower compression ratio.

However, irregularity caused by sparsity prevents accelerators from fully leveraging the computation and data reduction. Exciting architectures on field-programmable gate arrays (FPGAs) for dense models are not efficient for sparse CNN models because a lot of weights are pruned so that most multiplication operations involve zero operands leading to low hardware efficiency [9], [25]–[29]. Sparse architecture on FPGAs has been investigated in recent years [30], [31]. Han *et al.* [30] designed the fully connected (FC) layers that use matrix-vector multiplication operations. In fact, the major operators in CNNs are convolution operations. Although the spatial convolution can be converted to matrix-vector multiplications, this will lead to a large memory footprint since the input feature map has to be copied multiple times when being flattened to a vector. Lu *et al.* [31] proposed a dataflow that exploits element-matrix multiplication as the key operation. However, this design holds low computation efficiency due to the imbalanced load of each processing engine (PE). Since this accelerator requires a large number of a lookup table (LUT) to buffer the input activations for nonzero weights, the performance is bounded by the number of LUT on FPGA, which leads to inefficient resource utilization.

To design an efficient FPGA accelerator for sparse CNN models, the following challenges have to be tackled.

Manuscript received May 4, 2020; accepted June 9, 2020. Date of publication July 1, 2020; date of current version August 26, 2020. This work was supported by the National Key Research and Development Program of China under Grant 2018YFB0904902. *(Corresponding author: Kejie Huang.)*

Chaoyang Zhu, Kejie Huang, Shuyuan Yang, Ziqi Zhu, and Haibin Shen are with the College of Information Science and Electronic Engineering, Zhejiang University, Hangzhou 310058, China (e-mail: 21760249@zju.edu.cn; huangkejie@zju.edu.cn; 21931061@zju.edu.cn; 21960370@zju.edu.cn; shen_hb@zju.edu.cn).

Hejia Zhang is with the Department of Computer Science, University of Southern California, Los Angeles, CA 90089 USA (e-mail: hejiazha@usc.edu).

Color versions of one or more of the figures in this article are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TVLSI.2020.3002779

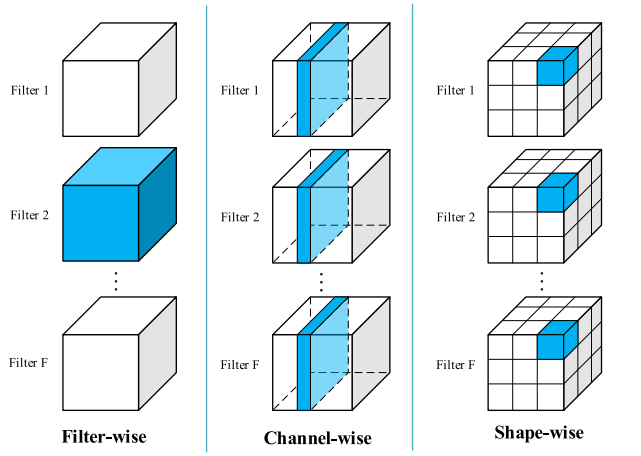


Fig. 1. Illustration of filterwise, channelwise, and shapewise structured pruning from left to right.

First, each output activation connects to several input activations through the sliding window (Kernel) in dense Convolutional (CONV) layers. The connection becomes irregular after pruning. Meanwhile, the sparse weights are encoded in compressed format, which results in extra coordinate computation to reconstruct the connection or to locate the output. Thus, it is challenging to design a dataflow to address the irregularity, whereas it efficiently leverages the data and computation reduction and maintains the high parallelism of FPGA.

Second, FPGAs can only provide limited on-chip memory and off-chip bandwidth. Although sparse CNNs have significant data reduction, it is difficult to save all the weights into on-chip memory for complex CNNs, such as VGG-16. Besides, different CNN models have different sizes, which results in high variability in the number of operations. A rigid accelerator architecture for CNNs may not fully utilize the FPGA's limited resources for every CNN model.

To address both challenges, we only focus on structured pruning to reduce the irregularity of sparse weights. On this basis, a sparsewise dataflow is proposed to address the remaining irregularity of sparse weights. With proposed dataflow, we do not need extra coordinate computation to reconstruct the connection or to locate the output. Furthermore, we minimize energy through zero-gating to avoid unnecessary computations if the input activations equal zero.

In conclusion, we make the following contributions: 1) we propose a sparsewise dataflow to skip the cycles of processing multiply-and-accumulates (MACs) that have zero weights and minimize energy through zero gating to further avoid unnecessary computations; 2) we propose a vector generator module (VGM) to reuse and generate the necessary input activations for sparse CNNs; and 3) we codesign both the accelerator architecture and the loop tiling to minimize off-chip memory accesses and maximize performance by slicing the input feature maps to best match the capacity of block random access memory (BRAM) on FPGA.

Experiments demonstrate that the proposed accelerator can achieve 987-, 46-, and 57-imag/s performance for AlexNet, VGG-16, and ResNet-50 on Xilinx ZCU102, respectively, which provides 1.5×–6.7× speedup and 2.0×–6.0× energy efficiency over previous CNN FPGA accelerators.

II. Background

CNNs consist of multiple types of layers, including convolutional layers, pooling layers, and FC layers. Through these layers, inputs are processed and propagated and, thus, to be classified or recognized. The convolution operation uses an $R \times R$ window to slide through the input feature map to extract features. At every location, the input activations inside the window are multiplied by corresponding weights, and the products are accumulated to compute the partial sum of output activation. Note that the partial sums in different input channels are accumulated to compute the output activation.

A. Network Pruning

CNNs have achieved remarkable success in various applications [1]–[3], [32], [33] at the cost of a huge amount of computations. The weights pruning method has been proved as an effective solution to reduce the computation and memory burden of these models with insignificant accuracy loss [19]–[24]. Structured sparsity learning (SSL) [34] proposes to learn structured sparsity at the levels of filters, channels, filter shapes, and so on. Fig. 1 illustrates three structured pruning schemes on the CONV layers of CNN: filterwise pruning, channelwise pruning, and shapewise pruning, which remove the whole filter(s), whole channel(s), and all neurons at the same location in each filter of a layer, respectively. Recently, a systematic solution framework [24] has been developed based on the powerful optimization tool alternating direction methods of multipliers (ADMM). Furthermore, AutoCompress [35] adopts the state-of-the-art ADMM-based structured weight pruning as the core algorithm and proposes an automatic structured pruning framework. Cambricon-S [23] applies the coarse-grained pruning (a kind of shapewise pruning) to prune weights on the both CONV and FC layers.

Compared with unstructured pruning techniques, structured pruning techniques are more regular and hardware friendly at the cost of a slightly higher sparsity. Table I shows the sparsity and accuracy comparison of deep compression [20] and coarse-grained pruning [23]. The difference in the sparsity between deep compression and coarse-grained pruning is almost negligible. However, coarse-grained pruning could achieve a very promising compression ratio compared with deep compression.

B. Loop Operation

In a typical CNN, convolutional layers take up about 90% of the computation in the inference procedure. A convolutional layer can be characterized by six parameters: F—the number of output feature maps (output channels); C—the number of input features maps (input channels); U—the height of output feature map; V—the width of output feature map; R—the kernel size; and S—the stride size, as shown in Fig. 2. The computation can be described in a deeply nested loop, as illustrated in Algorithm 1. Feature map-related loops f and

TABLE I
SPARSITY, COMPRESSION RATE, AND ACCURACY COMPARISON

model	Deep Compression			Coarse-grained pruning		
	Sparsity (%)	Compression Rate	Top1-E (%)	Sparsity (%)	Compression Rate	Top1-E (%)
AlexNet	11.15	35×	42.78	11.03	79×	42.72
VGG-16	7.61	49×	31.17	8.07	98×	31.33
ResNet-152	55.00	8×	24.40	55.83	10×	25.05

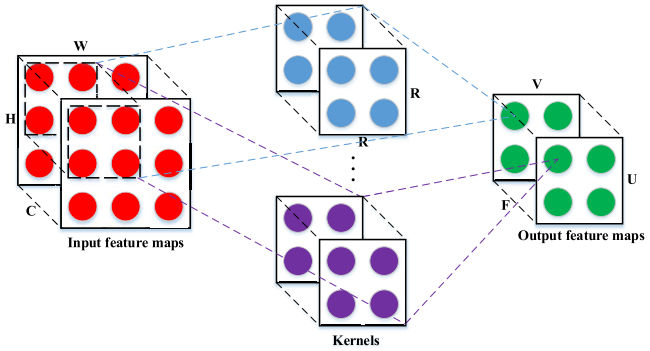


Fig. 2. Logical graph of convolutional operation. Convolutional layers conduct 2-D convolutions on a set of input feature maps and add the results to get output feature maps.

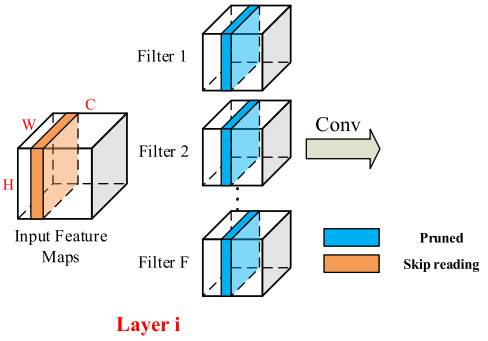


Fig. 3. Accelerating channelwise-pruned CONV layer by optimizing the memory reading and loop control.

Algorithm 1 Pseudocode of the Convolutional Layer

```
1  for f = 0; f<F; f++ do
2      for c = 0; c<C; c++ do
3          for u = 0; u<U; u++ do
4              for v = 0; v<V; v++ do
5                  for kh = 0; kh<R; kh++ do
6                      for kw = 0; kw<R; kw++ do
7                          Y^f_{u,v} += W^{f,c}_{kh,kw} × X^c_{u+kh,v+kw};
8                      end
9                  end
10             end
11         end
12     end
13 end
```

c index the output and input channels, respectively. Activation-related loops u and v index each activation of feature maps. Finally, weight-related loops kh and kw index weights of each kernel. Obviously, convolution operation exhibits high parallelism in the channel, activation, and weight levels.

To achieve high performance, the abovementioned deep-nested loop is unrolled and mapped to parallel hardware. This involves loop tiling and loop interchange strategy [36]. Loop tiling keeps all the input data of a loop tile stored on the chip to reduce external memory access. External memory access happens when the operation of a new tile begins. Besides, a part of the data may be temporally reused across the adjacent tiles. The loop interchange strategy decides the order of the loop tiles. Both loop tiling and loop interchange strategy decide the dataflow of hardware.

C. Accelerating Filterwise- and Channelwise-Pruned CNNs

Both channelwise sparsity and filterwise sparsity can be accelerated on hardware by optimizing the memory reading and loop control. Fig. 3 illustrates the concept of channelwise hardware acceleration. In this example, the third channel of each filter in Layer i is pruned, so the upper bound of the second loop (in Algorithm 1) is reduced to (C-1). Therefore, the corresponding computations can be skipped if the memory reading of the third channel in input feature maps is skipped. Fig. 4 is the example of the filterwise hardware acceleration. If the second filter in Layer i is pruned, the upper bound of the first loop (in Algorithm 1) is reduced to (F-1). The corresponding computations in Layer i can be skipped if the sparse filters have been compressed into a dense format. Meanwhile, the succeeding Layer i+1 turns into the channelwise pruning, where the corresponding computations can be skipped, as discussed earlier.

Therefore, the computations in both filterwise and channelwise pruning can be accelerated on a dense accelerator without adding extra hardware circuits, which is achieved by optimizing memory reading of the activations or weights and loop control. However, the acceleration cannot be achieved by skipping to read channel-level activations or weights in shapewise pruning. Extra hardware circuit is required to decide which part of activations needs to be loaded into PEs or to locate the output activations. To this end, we propose a sparsewise dataflow and a set of architecture optimization techniques in this design.

III. SPARSEWISE DATAFLOW

There have been dense CNNs dataflows on FPGA [9], [25], [37]. However, these dataflows cannot leverage the benefits of sparse CNNs since most multiplication operations involve zero weights that will not contribute to the corresponding output feature map. Therefore, it is extremely essential to skip the cycles of processing MACs that have zero weights.

Recently, designing dataflows for sparse CNNs on ASIC platforms has attracted attention from the research community.

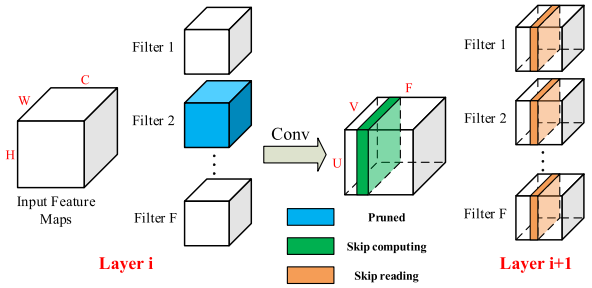


Fig. 4. Accelerating filterwise-pruned CONV layers by optimizing the memory reading and loop control.

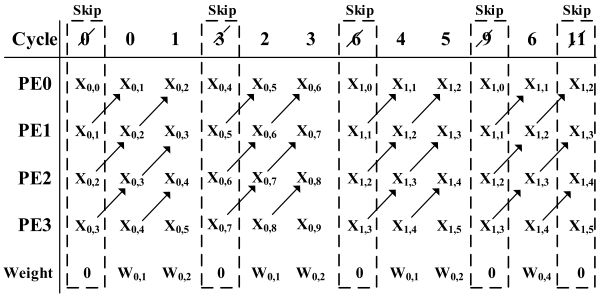


Fig. 5. Sparsewise dataflow for CONV layers. When weights equal to zero, the cycles of processing MACs will be skipped by controlling the upper bound of the corresponding loop illustrated in Algorithm 2.

However, these dataflows will not be efficient for FPGA platforms due to the architecture difference between ASIC and FPGA. For example, SCNN architecture [38] applies input-stationary dataflow where the inner computation is a Cartesian production. In SCNN architecture, there are N PEs, and each PE contains an $I \times D$ multiplier array. Consequently, it simultaneously requires $N \times I$ input activations and $N \times D$ weights for computation. Although this dataflow temporally reuses input activations, it still requires to update $N \times D$ weights in each cycle. Furthermore, this method first requires significant coordinates computation to locate output activations. Then, using the Cartesian production, this dataflow returns multiple partial sums that are needed to be arbitrated before being saved. Cambricon-S [23] applies output-stationary dataflow where the inner computation is a vector dot product. Cambricon-S implements a centralized indexing module to select input activations and only transfers the selected activations and indexes to PEs without extra coordinates' computation. However, this dataflow only performs the parallel computation in channel dimensions by spatially sharing input activations and, thus, requires M input activations and $N \times M$ weights for computation, which will lead to poor parallelism on FPGAs.

We propose a sparsewise dataflow to accelerate CNNs on FPGA (see Algorithm 2). For a convolutional layer with input feature maps Ifamp(C, H, W) and kernels Kernel(F, C, R, R), we select and fetch a vector of T_{oc} nonzero weights from T_{oc} adjacent kernels at each access. Similarly, we also fetch a vector of T_{om} activations that are in the same row of input feature map. Meanwhile, we compute T_{oc} element-vector multiplications on the PE array. After shapewise structured pruning, the coordinates of nonzero weights in different kernels are uniformed. Thus, each element-vector multiplication involves a weight and a shared vector of T_{om} activations. For example, in step 1, the weight vector is $[W_{0,0}^{0,0}, \ldots, W_{0,0}^{T_{\text{oc}},0}]$, and the input vector is $[X_{0,0}^{0}, \ldots, X_{0,T_{\text{om}}}^{0}]$. In step 2, we fetch weight vector $[W_{0,1}^{0,0}, \ldots, W_{0,1}^{T_{\text{oc}},0}]$ and input vector $[X_{0,\text{jump}}^{0}, \ldots, X_{0,T_{\text{om}}+\text{jump}}^{0}]$ for the next computation and compute T_{oc} element-vector multiplications in the pipeline mode. The variable jump equals to the number of zeros between adjacent nonzero weights in the same row of a kernel.

Convolution windows that produce adjacent outputs share part of input activations. Therefore, the two successive fetched input vectors from the same row of input feature map involve partially shared data. We add a dedicated VGM (will be introduced in Section IV) to update the input register for reusing data. We describe the convolution operation execution pattern in Fig. 5. Using the pattern in Fig. 5, the four PEs compute four adjacent output activations and skip the cycles of processing MACs that have zero weights.

For an FC layer with input vector I_vector(C, 1) and weight matrix $W(F, C)$, the weight buffer (WB) delivers a vector of T_{om} nonzero weights from the adjacent rows of matrix W at

Algorithm 2 Pseudocode of Our Dataflow

Input :
The nonzero weight array $W_{kh,kw}^{oc,ic}$;
The index array $Index[\,]$;
The row pointer of index $R\_pointer[\,]$;
The number of nonzero weights in each input channel $Offset[\,]$;
The input activation $X_{ih,iw}^{ic}$;
The Kernel size R×R×C×F;
The Output feature map size U×V×F;
Output:
The output activation $Y_{oh,ow}^{oc}$;

```
Initialize the parameter kw = 0 , kh = 0, i = 0 and jump = 0;
for oh = 0; oh<U; oh = oh + U_t do
    for oc = 0; oc<F; oc = oc + N do
        for ic = 0; ic <C; ic++ do
            if ic>0 then
                i = i + Offset[ic − 1];
            end
            for T_oh = oh; T_oh<oh + U_t; T_oh + + do
                for ow = 0; ow <V; ow = ow + M do
                    if kh = R&&ow>0 then
                        i = i − Offset[ic];
                    end
                    for kh = 0; kh <R; kh++ do
                        kh_count = ic × R + kh;
                        kw_max = R_pointer[kh_count];
                        if kh>0 then
                            i = i + R_pointer[kh_count − 1];
                        end
                        jump = 0;
                        for kw = 0; kw <kw_max; kw++ do
                            jump = jump + Index[i + kw];
                            //Loop unroll
                            for T_oc = oc; T_oc < oc + N; T_oc + + do
                                for
                                  T_ow = ow; T_ow<ow + M; T_ow + +
                                  do
                                    Y^{T_oc}_{T_oh,T_ow} = W^{T_oc,ic}_{kh,kw} ×
                                      X^{ic}_{T_oh+kh,T_ow+kw+jump};
                                end
                            end
                        end
                    end
                end
            end
        end
    end
end
```

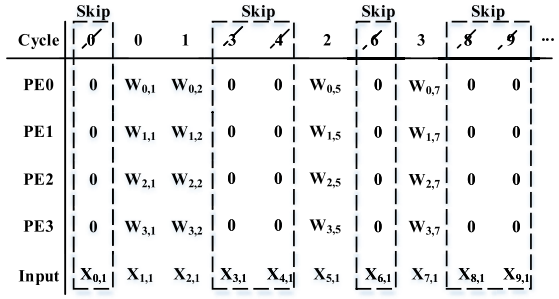


Fig. 6. Execution pattern of FC layers. Similar to CONV layers, the cycles of processing MACs will be skipped when weights equal to zero.

Algorithm 3 Pseudocode of Our Dataflow for 1×1 CONV Layers

Input :
The nonzero weight array $W(oc, ic)$;
The index array $Index[\,]$;
The number of nonzero weights in each filter $Offset$;
The input activation $X^{ic}_{ih,iw}$;
The Kernel size 1×1×Offset×F;
The Output feature map size U×V×F;
Output:
The output activation $Y^{oc}_{oh,ow}$;

```
Initialize the parameter index = 0;
for oh = 0; oh<U; oh = oh + U_t do
    for oc = 0; oc<F; oc = oc + N do
        for ic = 0; ic<Offset; ic++ do
            if ic = 0 then
                index = Index[ic];
            end
            else
                index = index + Index[ic] + 1;
            end
            //Update activation_read_address according to index for
              T_oh = oh; T_oh<oh + U_t; T_oh + + do
                for ow = 0; ow<V; ow = ow + M do
                    //Loop unroll
                    for T_oc = oc; T_oc < oc + N; T_oc + + do
                        for T_ow = ow; T_ow<ow + M; T_ow + + do
                            Y^{T_oc}_{T_oh,T_ow} = W(T_oc, ic) × X^{index}_{T_oh,T_ow};
                        end
                    end
                end
            end
        end
    end
end
```

each access (see Fig. 6). The input register only delivers a single activation.

Especially, in 1×1 CONV layers, shapewise pruning is equivalent to channelwise pruning since there is only one activation in each channel. Therefore, the dataflow (see Algorithm 3) for 1×1 CONV layers is quite different from what is described in Algorithm 2. Consequently, there are two steps to accelerating 1×1 CONV layers: step 1—compress the filters to get step index array Index[ ] and Offset; step 2—set upper bound of the third loop (in Algorithm 3) into Offset and update activation_read_address according to the index.

This dataflow requires a group of buffers to save partial sums produced by convolution operations. Considering a convolutional layer with output feature map Ofamp(F, U, V), we applied a $N \times M$ PE array. Make $T_{\text{oc}} = N$ and $T_{\text{om}} = M$ to prevent partial sums from being saved to and restored from DRAM; the depth of the partial sum (PSB) should be at least

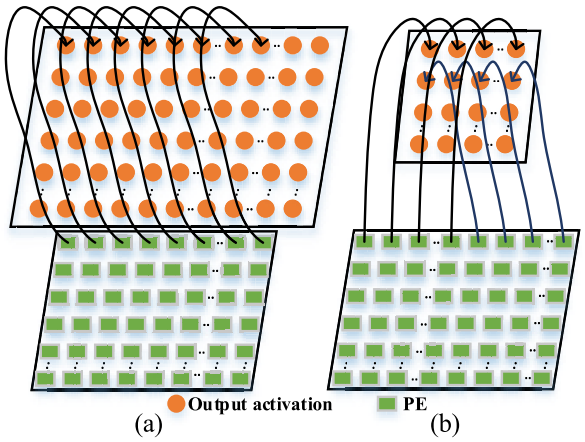


Fig. 7. Two situations of network mapping. (a) When $V \geq M$, one PU computes M partial sums of M output activations that are in the same row. (b) When $V < M$, one PU computes M partial sums of M output activations that are in the different rows.

$\lceil U \times V/M \rceil$. However, some CNNs, such as VGG-16, involve a quite large product of $U \times V$ in the first two layers. Therefore, we partition the input feature map into $\lceil H/H_t \rceil$ tiles, where

$$H_t = (U_t - 1) \times \text{Stride} + R = \left(\left\lfloor \frac{M \times \text{Slice}_{\text{BRAM}}}{V} \right\rfloor - 1 \right) \times \text{Stride} + R. \quad (1)$$

The parameter $\text{Slice}_{\text{BRAM}}$ represents the capacity of one block of BRAM on FPGA. The adjacent tiles share $(R - \text{Stride})$ rows because the sliding-window nature of the convolution operation introduces data dependence at tile edges.

When the width of output feature map $V < M$, we map a $\lfloor M/V \rfloor \times V$ tile to one processing unit (PU) for efficiency because each PE computes one output activation. Otherwise, we map a $1 \times M$ tile to one PU, as shown in Fig. 7. The PE array consists of N Pus, and each PU consists of M PEs in the same row. The input activations are shared across PUs. Different PUs compute output activations that are from different output channels.

By spatially sharing both input activations and weights, and temporally reusing partial input activations, we reduce the bandwidth requirement to $N \times B + M \times A$. Therefore, the bandwidth requirement of our proposed accelerator is much smaller than that of Cambricon-S $(M \times A + N \times M \times B)$ and SCNN $(N \times D \times B)$, where A and B represent the widths of activation and weight, respectively.

IV. Architecture of Sparsewise Accelerator

In this section, we introduce the detailed architecture of our accelerator to address the remaining irregularity of shapewise-pruned CNNs.

Overview: Fig. 8 depicts the overall block diagram of our proposed accelerator. Following the proposed dataflow, we design a VGM to address the sparsity with shared indexes. We design a PU that has multiple PEs to compute adjacent output activations in parallel. Multiple PUs constitute an array to compute multiple output activations across output channels

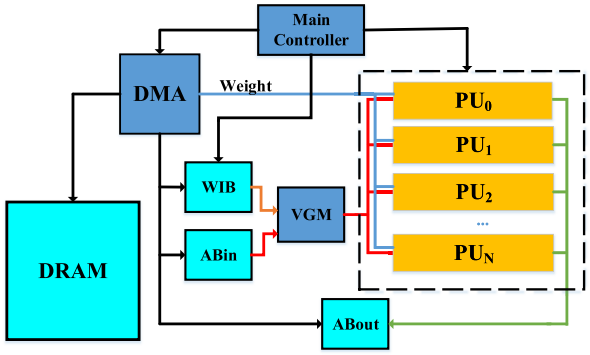


Fig. 8. Accelerator architecture. The DRAM is implemented with double data rate random access memory in PS on Xilinx FPGA.

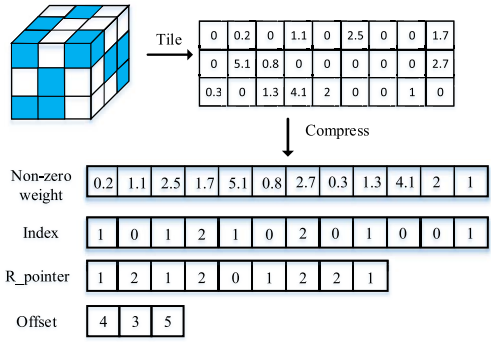


Fig. 9. Index representation of weights in CONV layers. Index: number of pruned weights between two nonzero weights. R_pointer: number of remaining weights in each row. Offset: remaining weights in each channel of one kernel.

in parallel. The storage module consists of two activation buffers (ABin and ABout), a weight index buffer (WIB), N WBs, and $N \times M$ PSBs. The main controller decodes instructions and weight indexes into detailed control signals for all other modules.

A. Addressing With Sparsity

The accelerator is designed to exploit structured sparsity for performance gain and energy reduction. In our accelerator, sparsity is processed by VGM and PU together. The VGM receives input activations from ABin, decodes weight indexes from WIB, and then produces the selected activations that are broadcast to all the PUs. Meanwhile, these input activations will be cached for the next selection in VGM since there is overlapping when the kernel slides across the input feature map. Each PU receives the read address of weight from the main controller and reads out the needed weight following the principles described in proposed dataflow, thus avoiding unnecessary computations.

1) Index: Before elaborating on the VGM and PU, we clearly explain how we store and index the sparse weights. We store the sparse weights that result from shapewise pruning using a compressed sparse row (CSR) format, which only requires $2a + R \times C + C$ numbers, where a is the number of nonzero weights, R is the number of rows, and C is the number

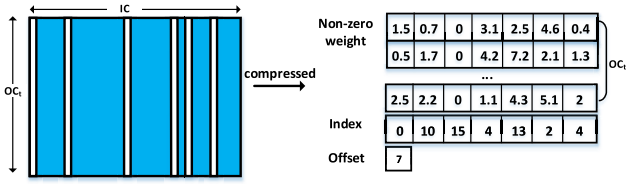


Fig. 10. Index representation of weights in FC layers. Padding filler zero to prevent overflow.

of input channels. Although different filters can share the same index after structured pruning, each 3-D filter can still be tiled in an irregular sparse matrix. To compress further, we store the step index instead of the absolute position. We encode both step index and R_pointer in 4 bits and encoded Offset in 16 bits, as illustrated in Fig. 9.

The 4-bit index is large enough in convolutional layers; however, the situation is different in FC layers. When we need an index larger than the bound, we will pad a filler zero to prevent overflow. Regarding the example in Fig. 10, when the step index exceeds the largest 4-bit unsigned number, we pad a filler zero.

2) VGM: The VGM module processes the sparsity by selecting the needed input activations and transfers the selected input activations to all the PUs (see Fig. 11). We design a central VGM shared by multiple PUs to more efficiently process shared indexes from structured sparsity. For example, first, when index is "1," the activations in register REG_0 will be shifted by A bits to the left (the most left A bits, where A represents the width of activation), and the data with index $(\text{REG}_0[0], \text{REG}_0[\text{Stride}], \ldots, \text{REG}_0[(M-1) \times \text{Stride}])$ will be broadcasted to all the PUs. In cycle 1, when index is "0," the activations in register REG_0 will be further shifted by A bits. PUs do MACs with previously selected input activations and cache the activations selected by VGM in this cycle. In cycle 2, the row number of kernel kh (see Algorithm 2) increases. Therefore, REG_0 reloads activations from REG_1 and does data-shift according to the new weight index. The depths of both REG_0 and REG_1 are set to $(M-1) \times \text{stride} + R$. To leverage the overlap between activation selection and memory access, the activations in REG_1 will be updated after REG_0 finally reloading activations from REG_1.

As PUs share the same indexes of weights due to shapewise pruning, the module for selecting activations (VGM) is shared by all the PUs, thus reducing the indexing module overhead and bandwidth requirement between VGM and PUs.

3) PU: The PU processes all operations in CNNs. Each PU consists of M zero-value discriminators, M homogeneous PEs, M homogeneous PSBs, a WB, pooling, normalization, and activation module (see Fig. 12). Weights can be stored separately in PUs as output activations from different channels involve independent kernels. A selected activation first streams into a zero-value discriminator and then will be used in PE if it does not equal to zero. Meanwhile, the discriminator produces an enable signal to control the clock gating of PE. When the selected activation equals to zero, the corresponding PE is clock gated. To minimize data communication, the partial

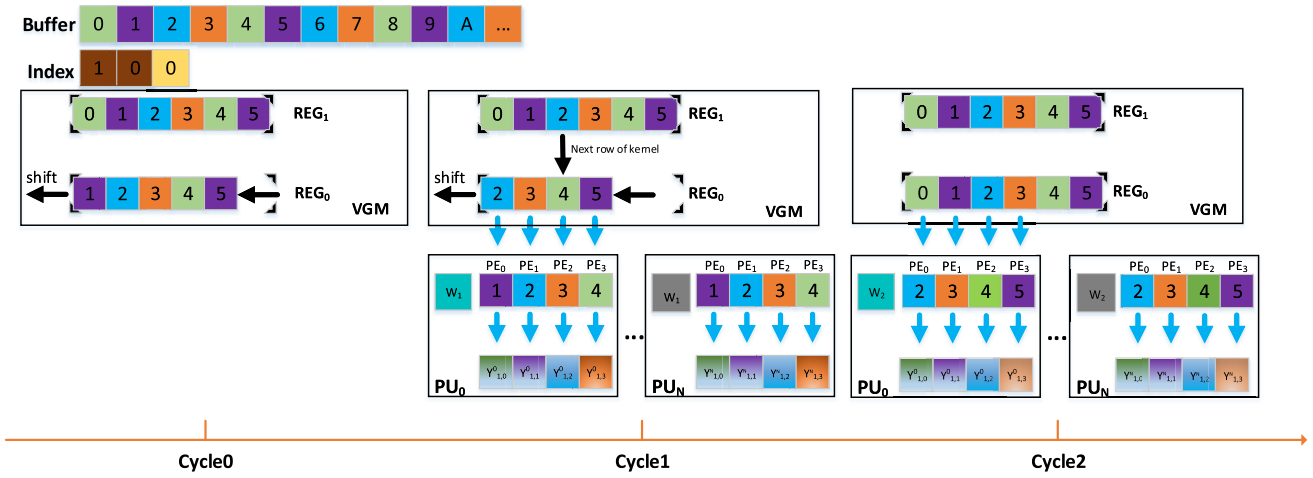


Fig. 11. The VGM buffers and selects input activations to reuse them and to address the weight sparsity. It is shared by all the PUs.

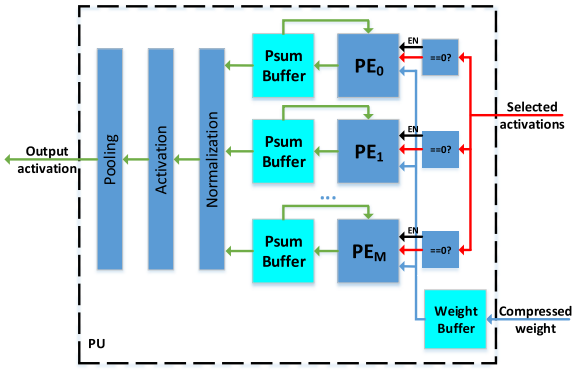


Fig. 12. Architecture of the PU. The PU processes all operations on CNNs. It contains M homogeneous PEs, and the number of PEs can be configured according to the CNN model and FPGA platform.

sums produced by PE will be saved in a local PSB that is placed next to the PE. Until the entire computation of output activation is done, the final partial sum will be transferred to the activation and pooling module.

As there is neither spatially sharing nor temporally reuse for weights in the FC layer, it requires quite large input bandwidth. Although all the PUs can be active in the FC layer, the required off-chip memory bandwidth cannot be satisfied on the FPGA platform. Thus, we only keep one PU active when M is large enough in the FC layer.

B. Optimize PE for Quantization

One of the most common methods for model compression is to quantize both weights and activations. The low-bit activations and weights require small bandwidth, which benefits to improve the throughput if the accelerator is a computation bounded design [15]. However, the proposed design is BRAM limited. Directly increasing the size of the PE array will lead to a failure of implementation on FPGA because the number of required BRAM will easily exceed the available number. Therefore, we maintain the datapath of activations and optimize the structure of PE. For example, the activations and weights are quantized to 8 bit. We dispatch two selected 8-bit activations into one PE. Due to the proposed dataflow, the two activations multiply with the same weight in two digital signal processing (DSP) slice, respectively. Then, the partial sums can be concatenated and stored in the buffer.

C. Storage

As the data processed in our accelerator have different behaviors, we split storage into five parts: an ABin, an ABout, a WIB, N WBs, and $N \times M$ PSBs.

For the ABin and ABout, we set the width as $16 \times (M - 1 + R)$ bit and $16 \times M$ bit, respectively, so as to provide $(M - 1 + R)$ input activations for VGM and to fetch M output activations from PU at each access. Benefiting from the proposed dataflow, N PUs share the same input activations, and we read input activations and produce output activations row-by-row. Thus, we set a small depth for both ABin and ABout.

For the WB in each PU, we use a dual-port ram for which we select the read width as 16 bit for one port and $16 \times M$ bit for the other port. In particular, the write width of both two ports is $16 \times M$ bit. In the convolutional layer, only one weight will be read out and be broadcasted to all the PEs, whereas, in the FC layer, M weights will be fetched and be transferred to the corresponding PE.

For the WIB, we select the width as 16 bit as we use CSR format where we deploy 16, 4, and 4 bit for Offset, Index, and R_pointer, respectively. Thus, Index and R_pointer are stored aligning to 4 bits. We divide the WIB into three parts for the three components of the compressed weights index.

For the PSB, we set the width to 32 bit. We store the partial sums and cache output activations in PSB, as the ABout fetches output activations from PUs in turns. We map each PSB to one BRAM. If we map each PSB to two or more BRAMs, the on-chip memory resources will easily be the bottleneck of the available peak performance, leading to the damping of throughput.

The size of the five buffers is decisive to overall performance and energy consumption. For example, the size of the

TABLE II
RESOURCE UTILIZATION BREAKDOWN

	BRAM	LUT	DSP	FF
PU	28	6407	28	5463
DMA	111	26875	0	6940
Controller	0	28392	6	1282
VGM	0	27963	0	7285
ABin	2	0	0	0
ABout	1	0	0	0
WIB	2	0	0	0
Total	1460(80%)	390K(65%)	1350(53%)	278K(51%)
Available	1824	600K	2520	550K

PSB decides the number of tiles. Small size of PSB requires a large number of tiles that lead to costly off-chip memory access, whereas a large size of PSB leads to unscalability for small layers in CNNs. Thus, we generally deploy 2 KB, 2 KB, 4 KB, 512 B, and 2 KB for ABin, ABout, WIB, WB, and PSB, respectively. The configuration of these buffers can be adjusted according to the CNNs.

V. EXPERIMENT

A. Experiments Setup

We evaluate our design on the Xilinx ZCU102 evaluation kit consisting of an Ultrascale FPGA, quad ARM Cortex-A53 processors, and 4-GB processing system (PS) DDR4. In this article, we use Verilog for RTL implementation and employ Xilinx Vivado (v2017.2) to compile the source code to the bitstream. The design method is inspired by DNNWEAVER [7]. Our FPGA implementation is synthesized at 200-MHz frequency. We use a GPIO to USB adapter to read the power directly from the PMbus in the FPGA board. We comprehensively apply [23], [24] methods to train the CNN models. In our experiment, we test typical CNNs including Lenet, AlexNet, and VGG-16 and achieve 11.85%, 32.92%, and 36.75% sparsity of Lenet, AlexNet, and VGG-16 without significant accuracy loss.

B. Resource Utilization

Table II shows the resource utilization breakdown with the configuration ($N = 48$ and $M = 28$). BRAMs are mainly used to construct the buffers and first in–first out (FIFOs). The parameter N determines the number of WBs in PUs and that of output FIFOs in direct memory access (DMA). The product of parameters M and N determines the number of PSBs. Each DSP can address a 16 bit $\times$ 16 bit multiplication or an 8 bit $\times$ 8 bit multiplication. The number of DSP can be calculated as $N \times M + 6$ when the width of the operand is 16. The six DSPs are used to calculate the index address for WIB and the number of shift bit for VGM. Besides the buffers, the rest modules consume LUT and flip-flop (FF).

Fig. 13 shows the resource utilization of different parallelism factors obtained from the Xilinx Vivado tool (v.2017.2). The LUT utilization increases as the total number of PE $N \times M$ increases. The utilization of BRAM is mainly used to construct buffers and FIFOs. When (N, M) increases to a certain extent, some large FIFOs are implemented by LUTs and FFs rather than BRAM to meet the timing constraints.

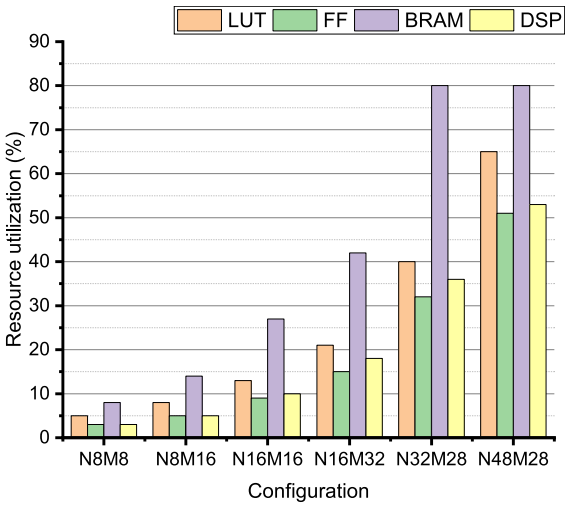


Fig. 13. Resource utilization of the accelerator under different configurations.

C. Computation Efficiency

In our scheme, all PUs are executed in parallel and each PU is assigned to calculate one channel in the output feature map. Benefiting from the shapewise pruning, the load of each PU is balanced. When we map the network onto PEs, the inefficiency of our design mainly comes from two aspects: dynamic activation inefficiency (DAI) and dataflow mapping inefficiency (DMI). First, there are zero-activations in input feature maps, which leads to some PEs gated. This pattern is designed to save energy deliberately. Second, the size of the feature map cannot be divided by M evenly, where M is the number of PE in each PU. The DAI is dynamic and variable dependent on the datasheet but irrespective of the proposed dataflow, whereas the DMI is dependent on the proposed dataflow and can be computed as the following equations.

We assume that the size of a 3-D output feature map is $U \times V \times F$. According to our dataflow, when $V > M$, the average computation efficiency is shown as follows:

$$\text{Compute}_{\text{eff}} = \frac{V}{\lceil V/M \rceil M}. \tag{2}$$

When $V < M$, the average computation efficiency is shown as follows:

$$\text{Compute}_{\text{eff}} = \frac{U \times V}{\left\lceil \frac{U}{\lfloor \frac{M}{V} \rfloor} \right\rceil M}. \tag{3}$$

We measure the DMI on AlexNet, VGG-16, and ResNet-50. In Fig. 14, the computation efficiency involves in DMI across different layers with different parallelism factors. According to Fig. 14(a)–(c), the computation efficiency is high if M is close to a factor of the product of U and V. For example, both U and V in VGG-16 can be divided by 14, and the computation efficiency keeps 100% when M equals to 14 or 28. As for 1×1 Conv layers in ResNet-50, the computation efficiency is low when M increases because the performance is bounded by the off-chip memory bandwidth. According to Fig. 14(d)–(f), the computation efficiency is reduced when N equals to 48 because 48 is not a factor of the parameter

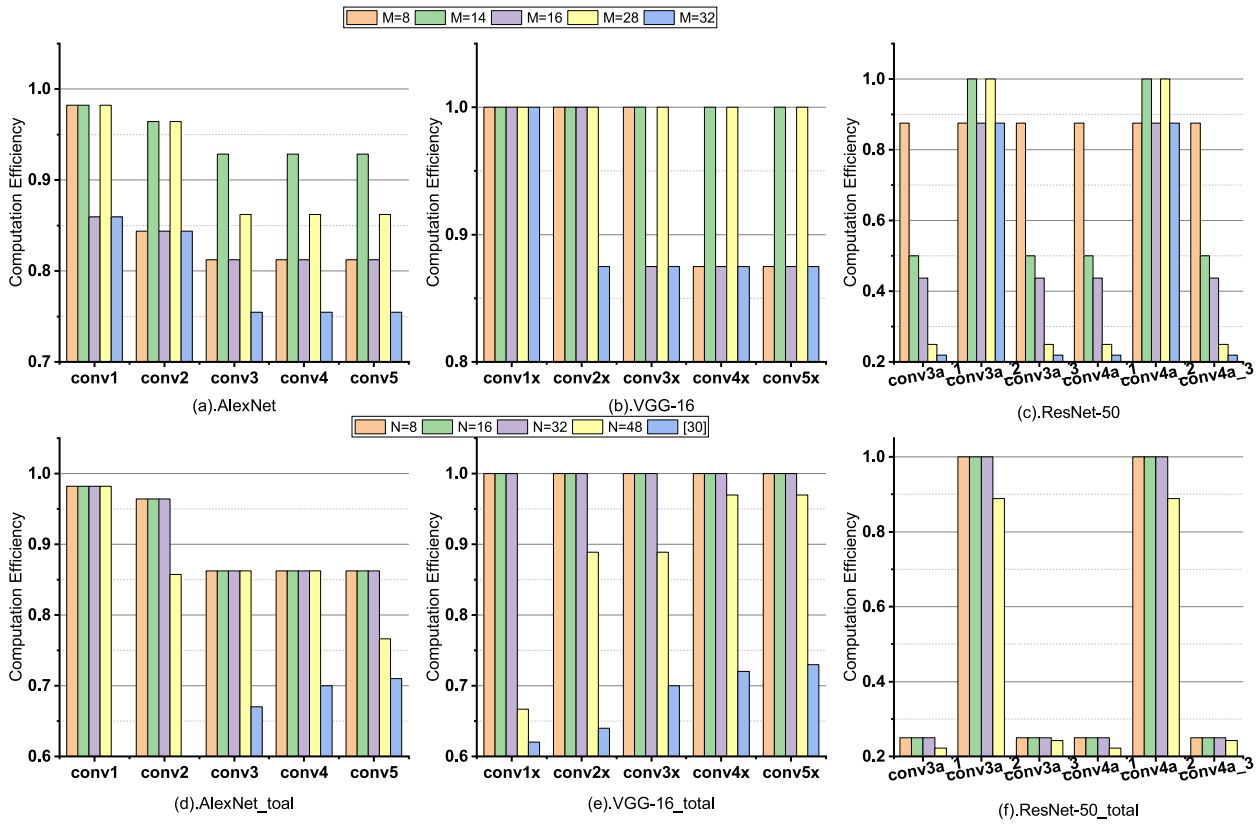


Fig. 14. (a)–(c) Computation efficiency of AlexNet, VGG-16, and ResNet-50 (partial layers) under different M values, respectively. N is kept to 1 for all three figures. Specifically, in (c), conv3a_2 and conv4a_2 are 3×3 Conv layers, and others are 1×1 Conv layers. (d)–(f) Computation efficiency of AlexNet, VGG-16, and ResNet-50 (partial layers) under different N values, respectively. M is set to 28 for all the three figures. Besides, (d) and (e) elaborate the computation efficiency of design [31].

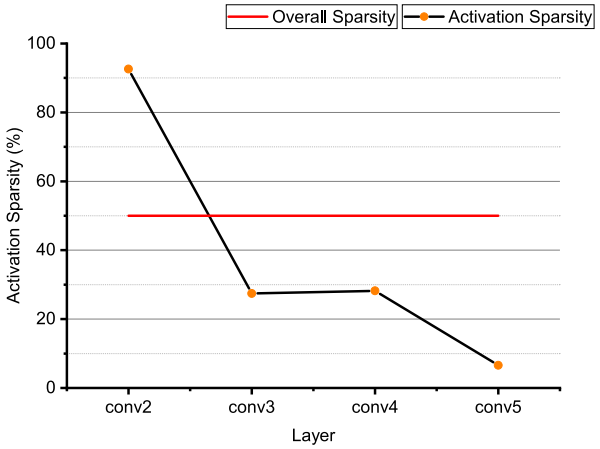


Fig. 15. Activation sparsity of CONV layers in AlexNet. The overall sparsity is about 39.6%.

"F." However, the computation efficiency is still better than that of design [31]. In conclusion, our sparsewise dataflow can maintain high computation efficiency for different neuron networks.

To analyze DAI, we first count sparse activations on convolutional layers of AlexNet and VGG-16 by using Pytorch vision. The data set is ImageNet 2012. As shown in Fig. 15, layer conv5 shows the lowest activation sparsity below 10%, and layer conv2 shows the highest activation sparsity over 90%. The overall sparsity is about 39.6%. As for VGG-16, Fig. 16 depicts that the last seven convolutional layers show a low sparsity below 30% and layer conv2_1 shows the highest sparsity about 75%. In the mass, the overall activation sparsity is about 39.5%. According to the activation sparsity, we can estimate the DAI.

Then, we measure the DAI on conv2_1 of VGG-16 by simulation. On each compute cycle, we sample the total number of active PEs and calculate the efficiency. As the number of total samples is too large, we calculate an average efficiency of every 100 samples. Fig. 17 shows the proportion of ungated PE. Indeed, a part of PEs is gated to save energy. The proportion of active PE on conv2_1 is positively related to the activation sparsity. The sample circuit is designed for simulation when the configuration parameter $\langle N, M \rangle = \langle 16,28 \rangle$ and does not be implemented on FPGA.

D. Performance Analysis

In this section, we adopt the well-known roofline model [15] for exploring the impact of insufficient off-chip bandwidth on performance. We set the bit width of the AXI data bus, which connects to DDR as 128 bit. First, we do not quantize the weight and change the configuration to find the optimal parameters of mapping the FC6 layer of VGG-16 onto our accelerator. We normalize all the performance numbers to

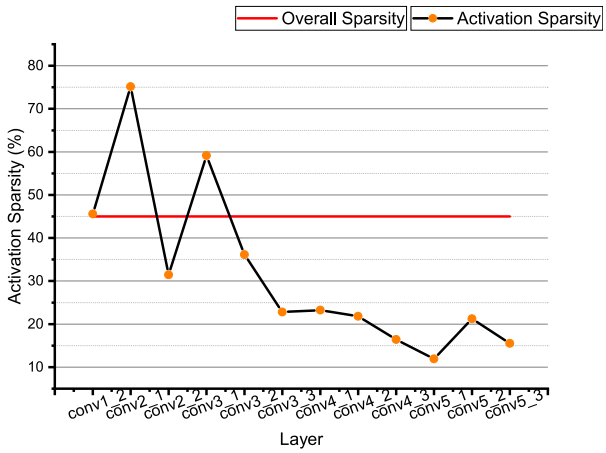


Fig. 16. Activation sparsity of CONV layers in VGG16. The overall sparsity is about 39.5%.

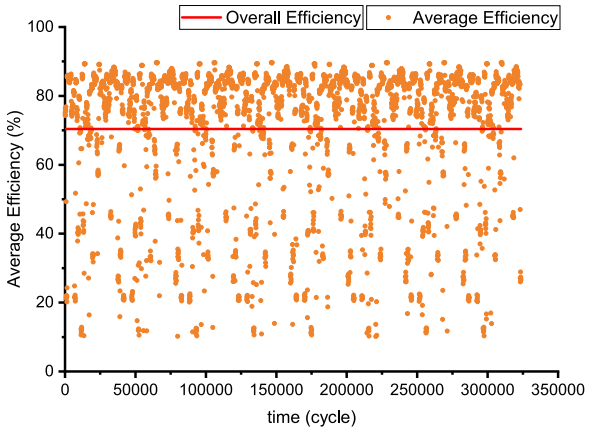


Fig. 17. Proportion of active PEs on conv2_1 of VGG-16. We sample the number of active PEs by simulation under the configuration of $\langle N, M \rangle = \langle 32,28 \rangle$. Each point represents the average efficiency of 100 samples.

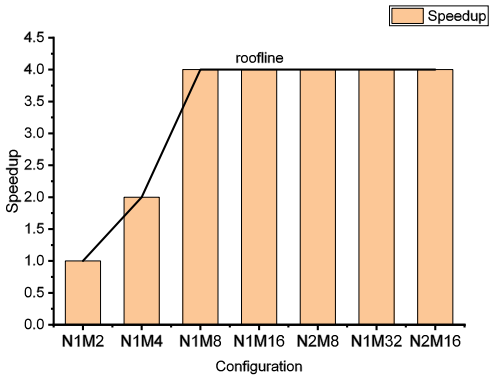


Fig. 18. Roofline model for 16 bit fixed in FC layer under different configurations. When the number of PEs increases to 8, the performance touches the roof because the off-chip bandwidth is fully utilized.

that of "N1M2." As shown in Fig. 18, when the number of PE reaches 8, the performance touches the roof. Second, we quantize the weights to 8 bit. We find that the performance touches the roof until the number of PE reaches to 16, and the roof becomes higher in Fig. 19 because there is neither weight share nor weight reuse in the FC layer. When the bit

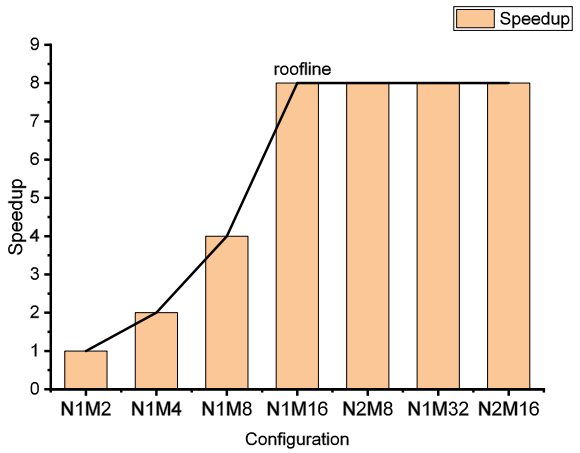


Fig. 19. Roofline model for 8-bit int in FC layer under different configurations. The width of weight is 8 bit, so the performance touches the roof until the number of PEs increases to 16.

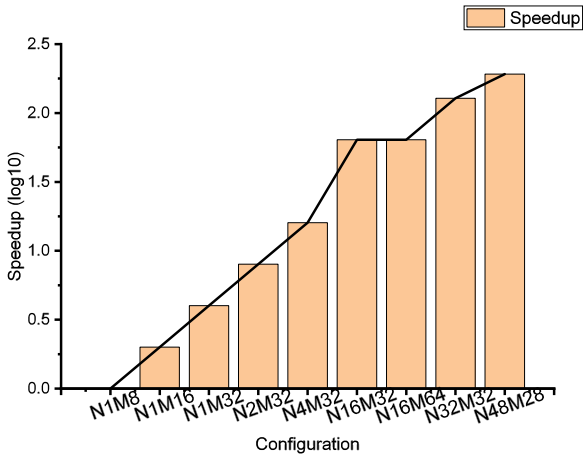


Fig. 20. Roofline model for 16 bit fixed in CONV layer under different configurations. When the number of PEs in a PU increases from 32 to 64, the performance does not improve.

width of weight decreases by half, the number of transferred weights doubles in each DDR access. Thus, the speedup also doubles.

After that, we map the conv2_1 layer of VGG-16 onto our accelerator to explore the optimal parameters. To get a high compute efficiency, we set N as the divisor of the number of output channel F. We normalize all the performance numbers to that of "N1M8." In Fig. 20, configuration "N48M28" achieves the best peak performance.

The total power consumption is about 15.4 W. Fig. 21 depicts the power consumption breakdown. Specifically, PS (containing DDR4) consumes about 18% (2.8 W) of power, DSP consumes about 11% (1.7 W) of power, BRAM consumes nearly 15% (2.3 W) of power, LOGIC consumes about 42% (6.5 W) of power, CLOCK consumes nearly 8% (1.2 W) of power, and other parts consume 6% (0.9 W) of power. Concretely, the power consumption of DDR4 is about 1.4 W.

Finally, we analyze the performance of our implementation. We set the PE array size as $\langle N, M \rangle = \langle 48,28 \rangle$, which consists of 1344 PEs. In this configuration, the peak throughput can

TABLE III
PERFORMANCE COMPARISON WITH PREVIOUS IMPLEMENTATION

	[39]	[40]	[31]	Ours	[31]	[26]	Ours	Ours	[41]	Ours
CNN type	AlexNet	AlexNet	AlexNet	AlexNet	VGG-16	VGG-16	VGG-16	VGG-16	ResNet-50	ResNet-50
Device	Zynq ZC706	Virtex XC7VX690T	Zynq ZCU102	Zynq ZCU102	Zynq ZCU102	Arria-10 GX1150	Zynq ZCU102	Zynq ZCU102	Arria-10 GX1150	Zynq ZCU102
Accelerator type	sparse	dense	sparse	sparse	sparse	dense	sparse	sparse	dense	sparse
Sparsity(%)	-	100.00	10.80	32.92	11.70	100.00	36.75	36.75	100.00	45.00
MAC Reduction(%)	-	-	65.1	66.7	67.4	0.0	65.4	65.4	0.0	52.3
Frequency(MHz)	-	200	200	200	200	150	200	200	200	200
Precision	-	16bit fixed	16bit fixed	16bit fixed	16bit fixed	16bit fixed	16bit fixed	8bit int	16bit fixed	16bit fixed
DSP Utilization	-	1436(40%)	1144(45%)	1352(53%)	1144(45%)	1518(100%)	1352(53%)	2520(100%)	1046(69%)	1352(53%)
Logic Utilization	-	468K(67%)	552K(92%)	390K(65%)	552K(92%)	161K(38%)	390K(65%)	405K(67%)	128K(30%)	390K(65%)
BRAM	-	423(39%)	912(48%)	1460(80%)	912(48%)	1900(70%)	1460(80%)	1460(80%)	2167(80%)	1460(80%)
Performance(imag/s)	147	548[1]	446	987	31	22	46	92	36	57
Power(W)	9.6	17.3	23.6	15.4	23.6	45.0	15.4	17.1	-	15.4
Efficiency(imag/s/W)	15.3	31.7	18.9	64.1	1.3	0.5	3.0	5.4	-	3.7

[1] This design uses unified Winograd-Gemm architecture.

TABLE IV
PERFORMANCE DENSITY COMPARISON WITH PREVIOUS IMPLEMENTATION

	[40]	Ours	[31]	[42]	[26]	Ours	Ours
CNN type	AlexNet	AlexNet	VGG-16	VGG-16	VGG-16	VGG-16	VGG-16
Device	XC7VX690T	Zynq ZCU102	Zynq ZCU102	XC7VX690T	Arria-10 GX1150	Zynq ZCU102	Zynq ZCU102
Frequency(MHz)	200	200	200	200	150	200	200
Precision	16bit fixed	16bit fixed	16bit fixed	8bit BFP	16bit fixed	16bit fixed	8bit int
Performance(GOP/s)	270	476.7	309.0	281.5	232.2	495.4	990.8
DSP Efficiency(GOP/s/DSP)	0.188	0.353	0.270	0.275	0.153	0.367	0.393
Logic cell Efficiency(GOP/s/K cells)	0.577	1.222	0.560	1.213	1.442	1.270	2.446

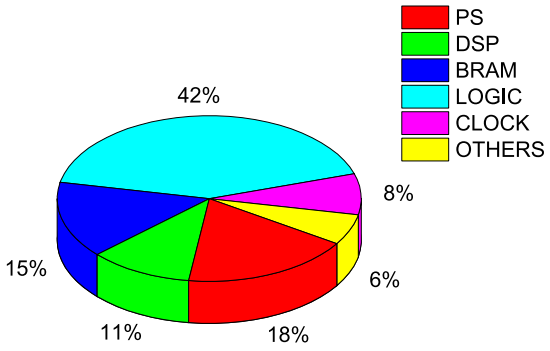


Fig. 21. Power consumption breakdown.

be calculated as 2×0.2 GHz $\times 28 \times 48 = 537.6$ GOP/s when the width of the operand is 16. Especially, the proposed design supports to perform two 8 bit $\times$ 8 bit multiplications in a PE with two DSPs, which leads to 1075.2-GOP/s peak performance.

We compare our design with convolutional FPGA accelerators in Table III. The performance in Table III represents the effective throughput. References [26], [40], and [41] are dense CNN accelerators, and [31] and [39] are sparse CNN accelerators. Both [31] and [39] only address the weight sparsity but do not address the activation sparsity, so we compute our throughput with DMI that is defined in previous subsection. For the dense accelerator, the performance is computed by dividing the effective throughput with computation of dense network. According to Table III, our accelerator achieves 987-imag/s effective performance on structured sparse AlexNet, which shows 1.8×–6.7× speedup and 2.0×–4.2× energy efficiency compared with [31], [39], and [40]. As for VGG-16, our implementation achieves 46-imag/s performance, which is 1.5×–2.1× speedup and 2.3×–6.0× energy efficiency compared with [26] and [31]. Compared with design [41], the proposed design achieves 1.6× speedup on ResNet-50. The performance of accelerating ResNet-50 is lower than accelerating AlexNet and VGG-16 because the 1×1 Conv layers occupy nearly half the computation burden. In 1×1 Conv layers, input activations cannot be temporary reuse since the size of the kernel equals that of the input feature map. Thus, the performance is bounded by the off-chip memory bandwidth. For the case of 8-bit int, we achieve 92-imag/s performance that is 3.0×–4.2× speedup and 4.1×–10.8× energy efficiency compared with [26] and [31].

The reason for the speedup is because our dataflow can effectively skip the sparse weight multiplications. In addition, this dataflow flexibly maps network onto PEs, which leads to the high utilization of on-chip resources. Previous works cannot efficiently exploit zeros or involve low compute efficiency. We take design [31] as an example to show the advantages of our proposed scheme. The nonstructured pruning in design [31] causes the remaining weights in different filters unevenly distributed. As a result, the latency is always bounded by the filter with the maximum number of nonzeros. To balance the load of each PE, this design aligns the weights with invalid data, by sacrificing the computation efficiency. In contrast, the load of each PU is balanced in our proposed design by doing shapewise pruning. Moreover, the dataflow in design [31] divides the output feature map into 8×8 tiles, resulting in an uneven division in some layers. Take the conv4x of VGG-16 as an example, the feature map size is 28×28 leading to a 23.4% waste of computation. Considering the abovementioned two points, the computation efficiency of [31] in VGG-16 is nearly 75%. Therefore, the effective

DSPs are only 768 when 1024 DSPs are configured as ⟨ 8, 8, 16 ⟩. In contrast, the computation efficiency of our design in VGG-16 is 100% when 896 DSPs are configured as a ⟨ 28, 32 ⟩ PE array. On the other hand, only half DSPs are utilized in design [31], and the performance is bounded by the logic resource. Benefited from the proposed dataflow and architecture, we can utilize 1344 DSPs in the PE array (1.31× more than that of [31]). Moreover, we apply the clock gate on unused PEs when the input activations equal to zero, resulting in higher energy efficiency than previous works.

The proposed design achieves a higher performance of resource efficiency because we leverage the sparsity and achieve a high mapping efficiency, as tabulated in Table IV. The accelerator [31] also leverages the sparsity, but the DSP efficiency is encumbered by the low mapping efficiency due to the unbalanced load of PE. In addition, the logic cell efficiency is only 0.560-GOP/s/K cells because the TLUT and CMUX [31] consume a large number of logic resources. Reducing the bit width can help to improve the resource utilization efficiency. The design [42] achieves a performance of 0.275-GOP/s/DSP and 1.213-GOP/s/K cells because it uses an 8-bit block float point to represent activations and weights so that two multiplication operations can be carried out in a DSP slice. In the proposed design, if the width of data is 8 bit, the logic cell efficiency nearly improves 100%.

VI. Conclusion

In this article, we have proposed a sparse CNN FPGA accelerator with a sparsewise dataflow to skip zero weights computations. Moreover, we have exploited data statistics to minimize energy through zeros gating to avoid unnecessary computations. In addition, we have proposed a set of architecture optimization techniques for sparse CNNs. Experiments demonstrated that our implementation could achieve 987-, 46-, and 57-imag/s performances for AlexNet, VGG-16, and ResNet-50 on Xilinx ZCU102, respectively, which provides 1.5×–6.7× speedup and 2.0×–6.0× energy efficiency over previous CNN FPGA accelerators.

References

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification," in *Proc. IEEE Int. Conf. Comput. Vis. (ICCV)*, Dec. 2015, pp. 1026–1034.

[2] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2014, pp. 580–587.

[3] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 779–788.

[4] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. 25th Int. Conf. Neural Inf. Process. Syst.*, NY, USA, pp. 1097–1105.

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

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

[7] H. Sharma *et al.*, "From high-level deep neural models to FPGAs," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2016, pp. 1–12.

[8] Y. Ma, N. Suda, Y. Cao, S. Vrudhula, and J.-S. Seo, "ALAMO: FPGA acceleration of deep learning algorithms with a modularized RTL compiler," *Integration*, vol. 62, pp. 14–23, Jun. 2018.

[9] H. Zeng, R. Chen, C. Zhang, and V. Prasanna, "A framework for generating high throughput CNN implementations on FPGAs," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, Feb. 2018, pp. 117–126.

[10] Y. Ma, Y. Cao, S. Vrudhula, and J.-S. Seo, "An automatic RTL compiler for high-throughput FPGA implementation of diverse deep convolutional neural networks," in *Proc. 27th Int. Conf. Field Program. Log. Appl. (FPL)*, Sep. 2017, pp. 1–8.

[11] 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.

[12] A. Podili, C. Zhang, and V. Prasanna, "Fast and efficient implementation of convolutional neural networks on FPGA," in *Proc. IEEE 28th Int. Conf. Appl.-Specific Syst., Architectures Processors (ASAP)*, Jul. 2017, pp. 11–18.

[13] K. Guo *et al.*, "Angel-eye: A complete design flow for mapping CNN onto embedded FPGA," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 1, pp. 35–47, Jan. 2018.

[14] H. Li, X. Fan, L. Jiao, W. Cao, X. Zhou, and L. Wang, "A high performance FPGA-based accelerator for large-scale convolutional neural networks," in *Proc. 26th Int. Conf. Field Program. Log. Appl. (FPL)*, Aug. 2016, pp. 1–9.

[15] 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. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays (FPGA)*, 2015, pp. 161–170.

[16] M. Motamedi, P. Gysel, V. Akella, and S. Ghiasi, "Design space exploration of FPGA-based deep convolutional neural networks," in *Proc. 21st Asia South Pacific Design Autom. Conf. (ASP-DAC)*, Jan. 2016, pp. 575–580.

[17] L. Lu, Y. Liang, Q. Xiao, and S. Yan, "Evaluating fast algorithms for convolutional neural networks on FPGAs," in *Proc. IEEE 25th Annu. Int. Symp. Field-Programmable Custom Comput. Mach. (FCCM)*, Apr. 2017, pp. 101–108.

[18] Y. Ma, N. Suda, Y. Cao, J.-S. Seo, and S. Vrudhula, "Scalable and modularized RTL compilation of convolutional neural networks onto FPGA," in *Proc. 26th Int. Conf. Field Program. Log. Appl. (FPL)*, Aug. 2016, pp. 1–8.

[19] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Proc. Adv. Neural Inf. Process. Syst.*, 2015, pp. 1135–1143.

[20] 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: http://arxiv.org/abs/1510.00149

[21] C. Ding *et al.*, "CirCNN: Accelerating and compressing deep neural networks using block-circulant weight matrices," in *Proc. 50th Annu. IEEE/ACM Int. Symp. Microarchitecture*, Oct. 2017, pp. 395–408.

[22] C. Deng, S. Liao, Y. Xie, K. K. Parhi, X. Qian, and B. Yuan, "PermDNN: Efficient compressed DNN architecture with permuted diagonal matrices," in *Proc. 51st Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2018, pp. 189–202.

[23] X. Zhou *et al.*, "Cambricon-S: Addressing irregularity in sparse neural networks through a cooperative software/hardware approach," in *Proc. 51st Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2018, pp. 15–28.

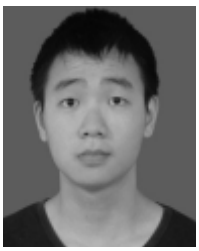
[24] T. Zhang *et al.*, "StructADMM: A systematic, high-efficiency framework of structured weight pruning for DNNs," 2018, *arXiv:1807.11091*. [Online]. Available: http://arxiv.org/abs/1807.11091

[25] J. Zhang and J. Li, "Improving the performance of OpenCL-based FPGA accelerator for convolutional neural network," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays (FPGA)*, 2017, pp. 25–34.

[26] Y. Ma, Y. Cao, S. Vrudhula, and J.-S. Seo, "Optimizing loop operation and dataflow in FPGA acceleration of deep convolutional neural networks," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays (FPGA)*, 2017, pp. 45–54.

[27] Y. Guan *et al.*, "FP-DNN: An automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates," in *Proc. IEEE 25th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, Apr. 2017, pp. 152–159.

[28] X. Zhang *et al.*, "AccDNN: An IP-based DNN generator for FPGAs," in *Proc. IEEE 26th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, Apr. 2018, p. 210.
[29] X. Wei *et al.*, "Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs," in *Proc. 54th Annu. Design Autom. Conf.*, Jun. 2017, p. 29.
[30] S. Han *et al.*, "ESE: Efficient speech recognition engine with sparse LSTM on FPGA," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, 2017, pp. 75–84.
[31] L. Lu, J. Xie, R. Huang, J. Zhang, W. Lin, and Y. Liang, "An efficient hardware accelerator for sparse convolutional neural networks on FPGAs," in *Proc. IEEE 27th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, Apr. 2019, pp. 17–25.
[32] H. Zhang, P.-J. Lai, S. Paul, S. Kothawade, and S. Nikolaidis, "Learning collaborative action plans from YouTube videos," in *Proc. Int. Symp. Robot. Res. (ISRR)*, Hanoi, Vietnam, 2019, pp. 1–16.
[33] H. Zhang and S. Nikolaidis, "Robot learning and execution of collaborative manipulation plans from YouTube cooking videos," 2019, *arXiv:1911.10686*. [Online]. Available: http://arxiv.org/abs/1911.10686
[34] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2016, pp. 2074–2082.
[35] N. Liu, X. Ma, Z. Xu, Y. Wang, J. Tang, and J. Ye, "AutoCompress: An automatic DNN structured pruning framework for ultra-high compression rates," 2019, *arXiv:1907.03141*. [Online]. Available: http://arxiv.org/abs/1907.03141
[36] 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: http://arxiv.org/abs/1712.08934
[37] J. H. Ko, B. Mudassar, T. Na, and S. Mukhopadhyay, "Design of an energy-efficient accelerator for training of convolutional neural networks using frequency-domain computation," in *Proc. 54th Annu. Design Autom. Conf.*, Jun. 2017, pp. 1–6.
[38] A. Parashar *et al.*, "SCNN: An accelerator for compressed-sparse convolutional neural networks," in *Proc. ACM/IEEE 44th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2017, pp. 27–40.
[39] S. Li, W. Wen, Y. Wang, S. Han, Y. Chen, and H. Li, "An FPGA design framework for CNN sparsification and acceleration," in *Proc. IEEE 25th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, Apr. 2017, p. 28.
[40] S. Kala, B. R. Jose, J. Mathew, and S. Nalesh, "High-performance cnn accelerator on FPGA using unified winograd-GEMM architecture," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 27, no. 12, pp. 2816–2828, Dec. 2019.
[41] Y. Ma, M. Kim, Y. Cao, S. Vrudhula, and J.-S. Seo, "End-to-end scalable FPGA accelerator for deep residual networks," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2017, pp. 1–4.
[42] 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.



Chaoyang Zhu (Graduate Student Member, IEEE) received the B.S. degree from the College of Physics and Technology, Central China Normal University, Wuhan, China, in 2017. He is currently working toward the master's degree at the College of Information Science and Electronic Engineering, Zhejiang University, Hangzhou, China.

His research interest includes hardware acceleration of neural networks.

Kejie Huang (Senior Member, IEEE) received the Ph.D. degree from the Department of Electrical Engineering, National University of Singapore (NUS), Singapore, in 2014.

He has been a Principal Investigator with the College of Information Science Electronic Engineering, Zhejiang University (ZJU), Hangzhou, China, since 2016. Prior to joining ZJU, he spent five years in the IC design industry, including Samsung and Xilinx, two years in the Data Storage Institute, Agency for Science Technology and Research (A*STAR), Singapore, and another three years in the Singapore University of Technology and Design (SUTD), Singapore. He holds four granted international patents and another eight pending ones. He has authored or coauthored more than 30 scientific papers in international peer-reviewed journals and conference proceedings. His research interests include low-power circuits and systems design using emerging nonvolatile memories, architecture, and circuit optimization for reconfigurable computing systems and neuromorphic systems, machine learning, and deep learning chip design.

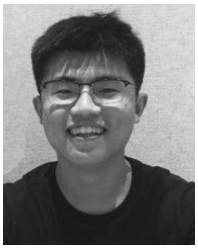
Dr. Huang is an Associate Editor of the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS II: EXPRESS BRIEFS.

Shuyuan Yang (Student Member, IEEE) received the B.S. degree in electronic science and technology from the Huazhong University of Science and Technology, Wuhan, China, in 2019. He is currently working toward the M.S. degree in electronic science and technology at Zhejiang University, Hangzhou, China.

His current research interests include deep learning accelerators and network on chip.

Ziqi Zhu (Student Member, IEEE) received the B.S. degree in electronic and information engineering from Zhejiang University, Hangzhou, China, in 2019, where he is currently working toward the M.S. degree in integrated circuits.

His current research interests include computer vision and 3-D object detection.



Hejia Zhang received the B.E. degree in bioengineering from Zhejiang University, Hangzhou, China, in 2017. He is currently working toward the Ph.D. degree in computer science at the University of Southern California, Los Angeles, CA, USA.

His current research interests include robot learning from videos and submodular optimization for active learning.

Haibin Shen is currently a Professor with Zhejiang University, Hangzhou, China and a member of the second level of 151 Talents Project of Zhejiang Province and the Key Team of Zhejiang Science and Technology Innovation. His research achievement has been used by many major enterprises. He has published more than 100 articles in academic journals, and he has been granted more than 30 patents of invention. His research interests include learning algorithm, processor architecture, and modeling.

Dr. Shen was a recipient of the First Prize of Electronic Information Science and Technology Award from the Chinese Institute of Electronics, and has won a Second Prize at the Provincial Level.