2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)

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

Liqiang Lu[1], Jiaming Xie[1], Ruirui Huang[2], Jiansong Zhang[2], Wei Lin[2], Yun Liang[1]*
[1] Center for Energy-efficient Computing and Applications, School of EECS, Peking University, Beijing, China
[2] Alibaba Group
Email: {liqianglu, jmxie, ericlyun}@pku.edu.cn
{ruirui.huang, muduan.zjs, weilin.lw}@alibaba-inc.com

***Abstract*—Deep convolutional neural networks (CNNs) have achieved remarkable performance at the cost of huge computation. As the CNN models become more complex and deeper, compressing CNNs to sparse by pruning the redundant connection in the networks has emerged as an attractive approach to reduce the amount of computation and memory requirement. On the other hand, FPGAs have been demonstrated to be an effective hardware platform to accelerate CNN inference. However, most existing FPGA architectures focus on dense CNN models. These dense architectures are inefficient when executing sparse models as most of the arithmetic operations involve addition and multiplication with zero operands.**

**In this work, we propose a hardware accelerator for sparse CNNs on FPGAs. To efficiently deal with the irregular connections in the sparse convolutional layers, we propose a weight-oriented dataflow that exploits element-matrix multiplication as the key operation. Each weight is processed individually which yields low decoding overhead. Then we design an FPGA architecture which features with a tile look-up table and a channel multiplexer. The tile look-up table (TLUT) is designed to match the index between sparse weights and input pixels. Using TLUT, the runtime decoding overhead is mitigated by using an efficient indexing operation. Moreover, we propose a weight layout to enable efficient on-chip memory access without conflicts. To cooperate with the weight layout, a channel multiplexer (CMUX) is inserted to locate the address. Experiments demonstrate that our accelerator can achieve 223.4-309.0 GOP/s for the modern CNNs on Xilinx ZCU102, which provides a 2.4X-12.9X speedup over previous dense CNN accelerators on FPGAs.**

## I. Introduction

Inspired by the biological nervous system, deep learning has recently achieved remarkable accuracy improvement. Convolutional neural networks (CNNs), the most commonly used model in deep learning, have been adopted in various domains, including image and speech recognition [1–4]. The significant accuracy improvement of CNNs comes at the cost of huge computational complexity as it requires a comprehensive assessment of all the regions across the feature maps.

Pruning deep neural networks has been proved as an effective solution to reduce the overall computation and memory requirements of these models while maintaining high accuracy [5, 6]. For example, Han et al. [5, 6] have shown that there is significant redundancy (up to 90%) for certain DNNs, which can be pruned without sacrificing the accuracy. Pruning techniques theoretically reduce the number of operations in the convolution algorithm, which potentially provide the opportunities for faster inference process.

However, existing architectures on FPGAs for dense models are not suitable for sparse CNN models. Most of these works optimize their dataflows based on loop operations like loop interchange and loop unrolling [7, 8]. The dense architecture can result in high hardware inefficiency since most multiplication operations involve zero operands [7–16]. Implementation of sparse DNNs has been studied in recent years on FPGAs[17]. These architectures mainly focus on the FC layers, which mainly use matrix-vector multiplication operations and are used in RNNs and LSTMs. However, the major operators of the modern CNN's computation are convolution operations. For example, the convolution operations occupy 90% of the total computation in GoogLeNet. Although the spatial convolution can be mapped to matrix-vector multiplications, this will increase the local memory requirement since the pixels in the input feature maps have to be copied multiple times when being flattened to a vector.

To design an efficient FPGA accelerator for sparse CNN models, it is very challenging due to the following reasons,

- The convolutional layers involve complex connections between input feature maps and output feature maps for sparse CNNs. Clearly, each output pixel is connected with part of the input pixels through the sliding kernels. The connection becomes irregular when the network becomes sparse. It is difficult to design a dataflow to deal with the irregularity but can leverage the high parallelism of FPGA and maintain FPGA efficiency.
- The sparse weights are encoded in sparse format, which requires extra coordinate computation to locate the weights. However, the distribution of the sparse weights (non-zeros) is irregular, which leads to inefficient memory access and low on-chip bandwidth utilization.

To address the first challenge, we propose a weight-oriented dataflow where each PE performs element-matrix multiplication instead of spatial convolution. Here, the element refers to the sparse weight and the matrix refers to the input tile. In this dataflow, the sparse weights are processed separately. By doing this, we successfully avoid the design issues related to sparsity such as irregular connections and load imbalance, etc. For the second challenge, we propose a weight layout, which

*Corresponding Author.

2576-2621/19/$31.00 ©2019 IEEE
DOI 10.1109/FCCM.2019.00013



IEEE computer society

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on November 27,2020 at 14:13:41 UTC from IEEE Xplore. Restrictions apply.

```
{L_H} for  h = 0 to H {
{L_W}    for w = 0 to W {
{L_N}      for n = 0 to N {
{L_M}        for m = 0 to M {
{L_R}          for r = 0 to R {
{L_S}            for s = 0 to S {
                   weight* = weight(n, m, r, s);
                   in* = in(m, h + r, w + s);
                   out(n, h, w) += weight* × in*;
}}}}}}
```

Fig. 1. A typical convolutional layer

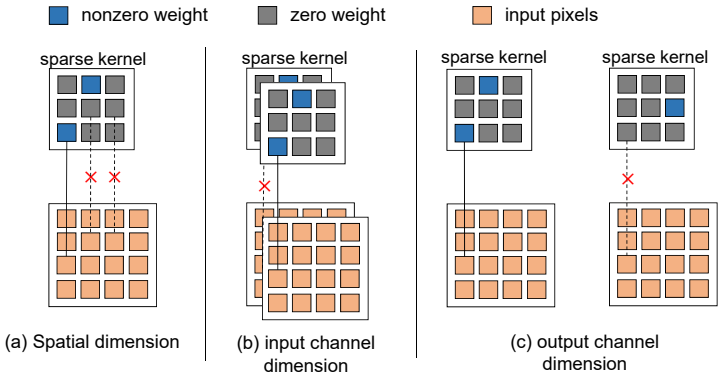


Fig. 2. Invalid computation caused by redundant connections in sparse CNNs

can enable efficient on-chip memory access of the weights. In this layout, the weights processed in parallel are stored continuously, and the results are accumulated from different BRAM banks to avoid access conflicts. Moreover, we design an efficient architecture for sparse CNNs that features with a tile look-up table (TLUT) and a channel multiplexer (CMUX). TLUT can reduce the overhead of runtime index matching and CMUX helps to locate the output address easily when updating the results.

In conclusion, this work makes the following contributions,

- We propose a dataflow with element-matrix multiplication as the key operation, where the element and the matrix refer to the sparse weight and input tile, respectively.
- We propose a weight layout which can enable efficient on-chip memory access. In this layout the weights used in parallel are stored continuously.
- We propose a set of architecture optimization techniques for sparse CNNs. We design a look-up table to match the weight with the corresponding input pixels and a channel multiplexer to locate the output address.

Experiments demonstrate that our accelerator can achieve 309.0, 223.4, 291.4 and 257.4 GOP/s for VGG, Alexnet, Resnet and GoogLeNet on Xilinx ZCU102, respectively. Our accelerator achieves a 2.4X-12.9X speedup over the previous dense CNN FPGA accelerators. Compared to TitanX GPU platform, our accelerator shows 7.56x energy-efficiency.

## II. Background

CNNs are a class of deep, feed-forward artificial neural networks, which are composed of a series of layers including convolutional layers, pooling layers and fully-connected layers (FC layer). The convolutional layer is the most important layer in which the kernels extract features from the input feature map. Figure 1 shows the typical convolution operation. The convolution operation uses a small $R \times S$ kernel to slide through the input feature map and the pixels inside the sliding window conduct a multiply-and-add operation with the weights in the kernel to compute a pixel value in the $H \times W$ output feature map. There are usually many input feature maps (aka input channels) and output feature maps (output channels) in a single convolutional layer, and the numbers of input maps and output maps are $M$ and $N$ as shown in Figure 1, respectively. Note that the convolution results in the different input channels are accumulated to obtain the output channel results.

TABLE I
Analysis of recent sparse CNN dataflow

| dataflow | type | inner computation | sparse format | Coordinate computation |
|---|---|---|---|---|
| SCNN[22] | pixel-oriented | Cartesian Product | # of in-between 0 | high |
| CambriconX[23] | kernel-oriented | Vector dot product | direct/stepping index | medium |
| Ours | weight-oriented | element-matrix multiplication | COO | low |

CNNs usually have a large number of weights, which could introduce the problem of over-fitting. The weights pruning techniques [5, 6] have been proven to be an effective method to reduce the computation and memory size while maintaining the overall model accuracy. For example, Deep Compression [5, 6] can reduce the number of weights in AlexNet [18] and VGG-16 [3] by 9X and 13X, respectively. These are known as unstructured pruning techniques. There are other pruning techniques which prune the weights with structured patterns [19–21]. The advantage of structured pruning techniques is they are hardware friendly. However, they often yield a low compression rate due to the strict mathematical formalization. The sparse CNN accelerators we propose can be used for both structured and unstructured pruning techniques.

## III. Weight-oriented dataflow

There have been prior efforts on designing dataflows for sparse CNNs on ASIC platforms. However, these dataflows will be inefficient for FPGA platforms due to the distinct architectures. In Table I, we classify prior ASIC designs based on the inner computation of the dataflow. SCNN architecture [22] applies the pixel-oriented dataflow where the innermost computation is a Cartesian product. Using Cartesian product, this dataflow multiplies input pixels with weights and returns



Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on November 27,2020 at 14:13:41 UTC from IEEE Xplore. Restrictions apply.

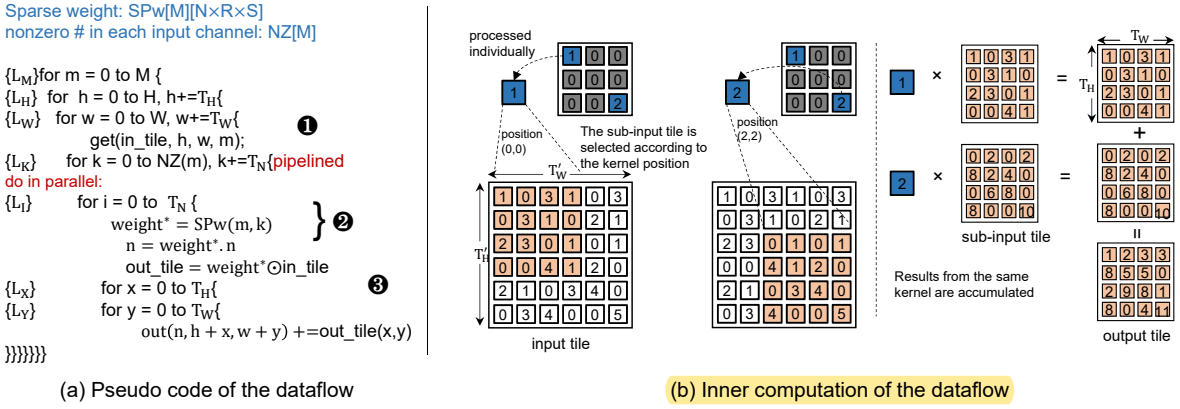


Fig. 3. Weight-oriented dataflow

multiple partial sums. This method requires significant coordinates computation to locate the sparse weights. Besides, the partial sums are connected with different output pixels, which bring great challenges for pipelining on FPGAs due to complex data dependency. Cambricon-X [23] design applies direct and step indexing technique to select the input pixels by detecting the nonzeros. Cambricon-X performs the vector dot product across channels by gathering the weights into a vector, which needs to dynamically select the input vector. This dataflow only performs parallel computation in channel dimensions, which will lead to poor parallelism on FPGAs.

There have been dense CNNs dataflows on FPGAs [8–10, 24, 25]. However, these dataflows will lead to invalid multiplications caused by the redundant connections between weights and input/ouput channels for sparse CNNs. As shown in Figure 2, the invalid multiplications can be from spatial kernel, input channel, and output channel dimensions, respectively. The input feature maps share the same index with the weight in the spatial kernel dimension and in the input channel dimension. In other words, the input pixel whose index matches the weight is needed when convolving the input with the kernel. Besides, different kernels are connected to different output feature maps, and the zero weight will not contribute to the corresponding output feature map.

We propose to transform the convolution computation to element-matrix multiplication by processing each weight as a single element. We compress the sparse weights into two arrays: (1) *SPw array*, where the nonzero weights in the same input channel are compressed into a vector. We use the coordinate list (COO) format to store sparse weights. More clearly, each element in the vector is 5-tuple $(n, r, s, value, valid)$ which represents the indices and the value of the weight; (2) *NZ array*, which records the number of non-zero weights in each input channel. One input channel is processed at a time. Figure 3 (a) shows the pseudo code of our dataflow which consists of three steps. In the **step 1**, we gather the necessary input pixels into an input tile according the position $(h, w, m)$. A $T_H \times T_W$ tile in the output feature map is connected with $T_H \times T_W$ pixels in the input feature map through a specific weight. Given a specific kernel size and the sliding stride, a $T_H \times T_W$ tile corresponds to a $T_{H'} \times T_{W'}$ tile in the input feature map as follows,

$$T_{H'} = R + stride \times (T_H - 1), \quad T_{W'} = R + stride \times (T_W - 1) \quad (1)$$

The input tile slides with a vertical stride $T_H$ and a horizontal stride $T_W$ as shown in Figure 3 (a). **Step 2** is the inner computation of our dataflow where $T_N$ weights are multiplied with the input tile in parallel.

Figure 3 (b) presents the details of the inner computation in the weight-oriented dataflow. Based on the position of the weight, we select a tile of input pixels that are connected with the weight. More clearly, given an output tile, each weight corresponds to a certain sub-input tile determined by the position of the weight in the kernel. For example, the value '1' in the top-left corner of the sparse weight multiplies with all the $4 \times 4$ top-left tiles of input feature maps. The weights are from different output channels. Finally, the multiplication results will be accumulated the output pixels according to the index $(n, h, w)$ in **Step 3**.

Our dataflow and its element-matrix multiplication inner computation has the following advantages. First, our dataflow processes the sparse weights one by one separately. By doing this, we can effectively exploit the sparsity and meanwhile reduce the sparsity deconding overhead. Second, our dataflow provides sufficient parallelism on FPGAs. More clearly, the output pixels in the spatial kernel and output channel dimensions are computed in parallel. Third, our dataflow has low data dependency overhead. The results from **Step 2** in Figure 3 are accumulated to different output pixels which have no read-and-write conflicts.

## IV. Architecture Optimization

In Section III, we transform the convolution operation to element-matrix multiplication. However, implementing this dataflow on FPGA arises two challenges. The first challenge is to select the necessary pixels for a specific weight. A single weight is connected to only part of the pixels in the input



Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on November 27,2020 at 14:13:41 UTC from IEEE Xplore. Restrictions apply.

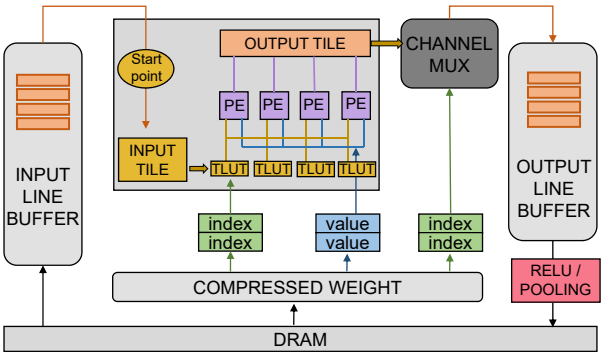


Fig. 4. Architecture overview

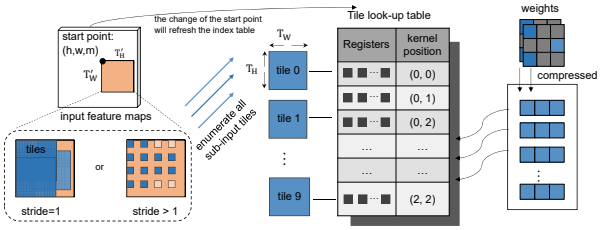


Fig. 5. Tile look-up table to locate the sub-input tile

feature maps, and the weight in the different position of the kernel is connected to different input pixels, as shown in **step 1** of Figure 3. Second, to ensure multiple results can be accumulated to the output buffer in parallel in **step 2**, a dedicated data layout is required under the hardware constraints of FPGA memory structure (e.g., dual-port BRAM). Furthermore, the PEs should be pipelined to increase the throughput.

Figure 4 shows the architecture overview. In Section IV-A, we use a tile look-up table (TLUT) to match the weight and the required input pixels. Using the tile look-up table can avoid runtime index matching. In Section IV-B, we propose a novel weight layout where the parallel weights are stored continuously. Besides, the layout can ensure the results from the PE array are accumulated to different output banks without data access conflicts in the pipeline. To cooperate with the layout, in Section IV-C, we propose a channel multiplexer (CMUX) to locate the channel address. The channel multiplexer receives the weight index in the sparse format and outputs which bank the results should be accumulated to. Since the weight distribution across output channels might be unbalanced, we analyze the load balancing problem in Section IV-D.

### A. TLUT module

As aforementioned, the weights represent the connections between the input feature map and the output feature map. However, when the weight is sparse, the connection loses its structured topology. To bridge the gap between irregular connections to input pixels and the regular PE array, we insert a tile look-up table between the input tile and PEs. Figure 5 depicts how the weight and the input pixels are paired. Given a position in the input feature maps and the number of

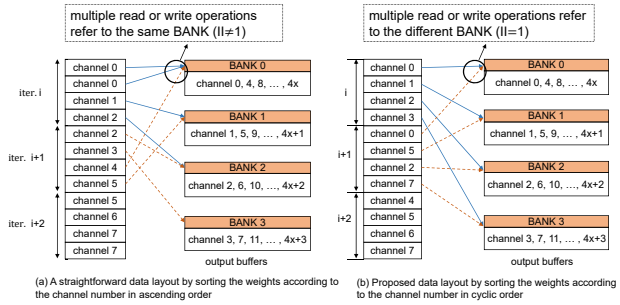


Fig. 6. Weight layout in the output channel dimension

output pixels that are computed in parallel, a region is selected according to Equation 1. Then, we enumerate all the possible sub-input tiles which need to be multiplied with the weights. Clearly, when the kernel is sliding in the input tile, the weight in a $R \times S$ kernel is connected to a set of input pixels in the input tile. These pixels are batched together into a new tile. When the stride is equal to 1, the selected pixels are adjacent to each other as shown in Figure 5. There are $R \times S$ tiles in total with $R \times S$ positions in the kernel which are stored separately in the tile look-up table. The tile look-up table replaces runtime index matching with a simple array indexing operation. This helps to save the logic resources significantly since the runtime index matching requires a large number of multiplexers. For example, in Figure 2, the position of the red weight is $(0,0)$ which corresponds the red tile, and we can directly fetch the pixels in the red tile which has been pre-fetched when the start point $(h,w,m)$ is determined.

### B. PE design and weight layout

The PE receives the decoded weight and the selected tile from the tile look-up table. We initiate a PE array with each PE conducting an element-matrix multiplication operation. In the **step 2** of our dataflow, we compute multiple output pixels from different output channels in parallel. There are $T_N$ homogeneous PEs process multiple weights and input tiles in parallel. Furthermore, we apply pipelining technique to our PE design. Pipelining allows multiple operations in **step 2** to process concurrently to increase throughput, and the pipeling efficiency is determined by the iteration interval. According to Figure 3, the iteration interval is bounded by the weight access bandwidth and output access bandwidth.

To enable simultaneous update of multiple output channels, the output buffer is partitioned to $T_N \times T_H \times T_W$ banks where each bank $i$ in the channel dimension stores the weights from the $(T_N \times x + i)$ output channel as shown in Figure 7. Traditionally, the weights are sorted in the ascending order of channels. If more than one weight need to be read from the same bank, this will lead to a long read latency. To address this problem, the weight layout is rearranged to cooperate with the partitioned output buffers. We define the quotient and remainder by dividing the output channel $n$ with $T_N$. In Figure 3, the compressed weights are stored in a 2D matrix $SPw$.



Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on November 27,2020 at 14:13:41 UTC from IEEE Xplore. Restrictions apply.

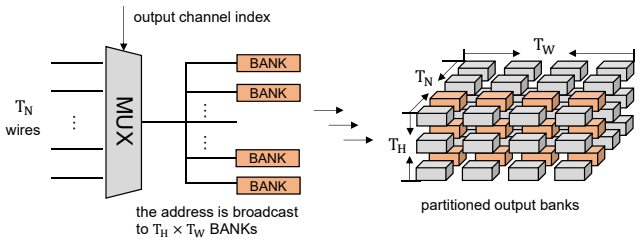


Fig. 7. Channel multiplexer to locate the output channel address

For each row in the matrix, we sort the weight according to the remainder. Specifically, we rearrange the weights to ensure that every consecutive $T_N$ weights have different remainders as shown in Figure 6, so that the results from the PE array are accumulated to the output buffers. For example, in Figure 6, 4 weights are processed in parallel. In our weight layout, the results from the PE array need to be accumulated to the output channel $(0,5,2,7)$ in iteration $i+1$, whose remainder are $(0,1,2,3)$. In this manner, multiple write operations refer to different banks, resulting in an improved iteration interval.

### C. CMUX module

In the PE array, each PE generates a tile of results that belong to a distinct output channel. The address that the results need to be accumulated to is determined by the index in the format. A channel multiplexer is inserted between the PE array and the output buffer to locate the address as shown in Figure 7. The channel multiplexer consists of $T_N$ input wires which represent the number of banks in the output channel dimension. The output channel index of weights is transferred to the channel multiplexer, then the channel multiplexer will output which bank the results need to be accumulated to.

### D. Load balancing analysis

In our architecture, PEs strictly process $T_N$ weights with different remainders together. However, the weights with different remainders cannot be evenly distributed. As a result, the latency is always bounded by the remainder with the maximum nonzeros. So we align the weights with invalid data among all the remainders so that the number of weights across different remainders is equal, as shown in Figure 8. There is a valid signal in COO format $(n,r,s,value,valid)$ to indicate whether the weight is valid. The computation efficiency can be computed as follows,

$$Compute_{eff} = \frac{\#\ of\ valid}{\#\ of\ valid + \#\ of\ invalid} \tag{2}$$

In the example of Figure 8, the parallelism factor $T_N$ is set to 4, 6, 8 with a fixed number of nonzeros 28. For each factor, the numbers of nonzeros with different remainders are different. We align the weight with each remainder to the same value and the grey points denote the invalid values which can cause inefficiency. For example, when $T_N = 8$, the computation efficiency is $\frac{28}{8\times5} = 70\%$. In the experimental section, we will analyze the computation efficiency using real networks.

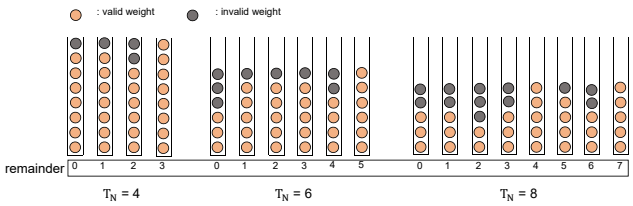


Fig. 8. Invalid computation under proposed weight layout

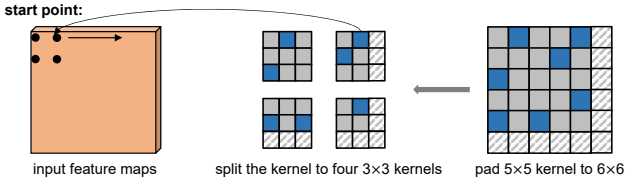


Fig. 9. An uniform design different kernel size

## V. Implementation Details

### A. Memory system

The on-chip memory of FPGAs is not large enough to hold all the channels of feature maps. Hence, we load and calculate feature maps at an unit of 128 channels. We apply line buffer techniques to store the input and output feature maps. There exist data reuse opportunities both horizontally and vertically since there is overlapping when the kernel slides across the input feature maps. Line buffer design is widely used in previous accelerators and can effectively reuse the input data. There are two input tiles working in a ping-pong manner to overlap the latency of the tile look-up table and the latency of the PE array. As shown in Figure 3, the latency of the PE array depends on the loop count of $L_K$ and the pipeline depth. In general, the latency of loop $L_K$ is much larger than the latency of the tile look-up table which is a constant.

### B. Uniform design

In general, the modern CNN networks contain different kernel size. For example, Resnet has $1\times1$ and $3\times3$ kernels in the residual block, and GoogLeNet has $1\times1$, $3\times3$ and $5\times5$ kernels in the inception module. Since each weight is processed independently in our dataflow, our architecture can flexibly handle different kernel size. To unify the structure of the tile look-up table, we transform all the kernels to the $3\times3$ kernel. Figure 9 shows an example that transforms the $5\times5$ kernel to the $3\times3$ kernel. The $5\times5$ kernel is padded to $6\times6$ kernel with zeros then split into four $3\times3$ kernels. Note that the splitted kernels start from different positions as shown in Figure 9. Apart from the convolutional layers, there are other layers in CNN models. In our architecture, we implement two widely-used layers: pooling layer and Rectified Linear Unit (ReLU) layer. Pooling layer outputs the maximum values in sub-regions of input feature maps. ReLU layer sets any input value less than zero to zero. These two layers are implemented by introducing comparison operators when writing the results to off-chip memory.



Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on November 27,2020 at 14:13:41 UTC from IEEE Xplore. Restrictions apply.

TABLE II
PERFORMANCE COMPARISON WITH PREVIOUS IMPLEMENTATION

| | [9] | [8] | [11] | [12] | [24] | Ours | Ours | Ours | Ours |
|---|---|---|---|---|---|---|---|---|---|
| CNN type | VGG | VGG | Resnet | Alexnet | Alexnet | VGG | Alexnet | Resnet | GoogLeNet |
| Device | Arria-10 GX1150 | Arria-10 GX1150 | Stratix-V GSMD5 | Stratix-V GSD8 | Zynq ZC706 | Zynq ZCU102 | Zynq ZCU102 | Zynq ZCU102 | Zynq ZCU102 |
| Frequency (MHz) | 150 | 370 | 150 | 120 | - | 200 | 200 | 200 | 200 |
| Precision | 16bit fixed | 32bit float | 16bit fixed | 16bit fixed | - | 16bit fixed | 16bit fixed | 16bit fixed | 16bit fixed |
| DSP Utilization | 1518 (100%) | 1320 (87%) | 1044 (66%) | - | - | 1144 (45%) | 1144 (45%) | 1144 (45%) | 1144 (45%) |
| Logic Utilization | 161K (38%) | 182K (43%) | 45.7K (27%) | - | - | 552K (92%) | 552K (92%) | 552K (92%) | 552K (92%) |
| BRAM | 1900 (70%) | 1250 (46%) | 959 (48%) | - | - | 912 (48%) | 912 (48%) | 912 (48%) | 912 (48%) |
| Performance (GOP/s) | 64.5 (eff.) | 128.5 (eff.) | 22.6 (eff.) | 11.2 (eff.) | 71.2 | 309.0 | 223.4 | 291.4 | 257.4 |
| Power (W) | 45.0 | 41.7 | 25 | - | 9.6 | 23.6 | 23.6 | 23.6 | 23.6 |

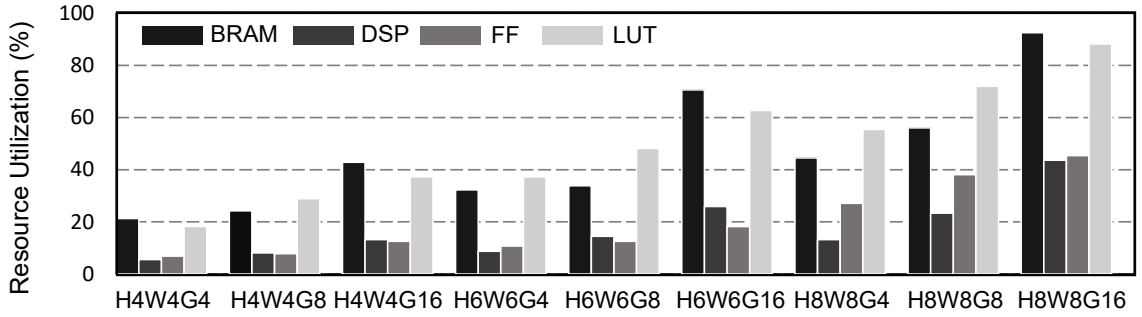


Fig. 10. Resource utilization (HaWbNc means $T_H = a, T_W = b, T_N = c$)

## VI. EXPERIMENT

In this section, we first introduce the experiments setting. In Section VI-B, we show the performance of our accelerator for the state-of-the-art CNNs and compare it with previous dense CNN FPGA accelerators. Then, we measure the resource utilization of various configurations and analyze the utilization breakdown. Next, we examine the hardware efficiency of different configurations with different parallelism.

### A. Experiments Setup

We evaluate our design on Xilinx ZCU102 platform. ZCU102 consists of an UltraScale FPGA, quad ARM Cortex-A53 processors, 500 MB DDR3. Our FPGA implementation is operated at 200MHz frequency on this platform. To measure the runtime power, we plugged in a power meter in the FPGA platform. In this work, we first use Xilinx Vivado HLS (v2017.4) tool chain to transform C code into RTL implementation. Then, we employ Xilinx SDSoC (v2017.4) to compile the source code into bitstream. We apply [5, 6] methods to train the CNN model with sparsity using Caffe model [26]. Specifically, we set the expected sparsity of the network by setting the value that is less than a threshold to zero, followed by retraining the network to regain any lost accuracy. In our experiment, we use the state-of-the-art CNNs including Alexnet, VGG-16, Resnet and GoogLeNet. We achieve 89.2%, 88.3%, 76.5%, 65.8% sparsity of Alexnet, VGG-16, Resnet, GoogLeNet without accuracy loss, respectively.

### B. Performance Analysis

In this section, we show the performance of our accelerator using modern CNNs. We set the accelerator configuration as $< T_H, T_W, T_N >=< 8, 8, 16 >$, which involves 1024 multipliers. In this configuration, the peak available performance can be computed as $2 \times 0.2\ GHz \times 16 \times 8 \times 8 = 409.6\ GOP/s$.

We also compare our design with previous FPGA accelerators in Table II. [8, 9, 11, 12] are dense CNN accelerators and [24] is sparse CNN accelerator. The performance in Table II represents the effective performance. For the dense CNN accelerators, the effective performance is computed by multiplying the performance of dense CNNs with sparsity. According to Table II, our implementation achieves 223.4 GOP/s effective performance on sparse Alexnet which shows 2.4x-19.9x speedup compared with [7, 24][1]. The performance on VGG network is 309.0 GOP/s which is 3.6X-4.8X higher than [8, 9, 27]. For Resnet, our design achieves 291.4 GOP/s which is 12.9X higher than the effective performance of [11]. On GoogLeNet, we achieve 257.4 GOP/s performance. The speedup is because our dataflow can effectively eliminate the useless multiplications, also, the dataflow maintains a high utilization of on-chip resources. Previous implementations cannot efficiently exploit the zeros in the computation, which results in a waste of on-chip resources. On the other hand, previous dense CNN accelerators are highly optimized and DSPs are fully utilized to conduct multiplications. In our implementation, only half DSPs are utilized, and the performance is bounded by the logic resource.

The inefficiency of our implementation mainly comes from three aspects. First, there exist some invalid weights in our weight layout, which leads to imbalanced workload among

[1]This paper [24] only reported the performance and the platform



Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on November 27,2020 at 14:13:41 UTC from IEEE Xplore. Restrictions apply.

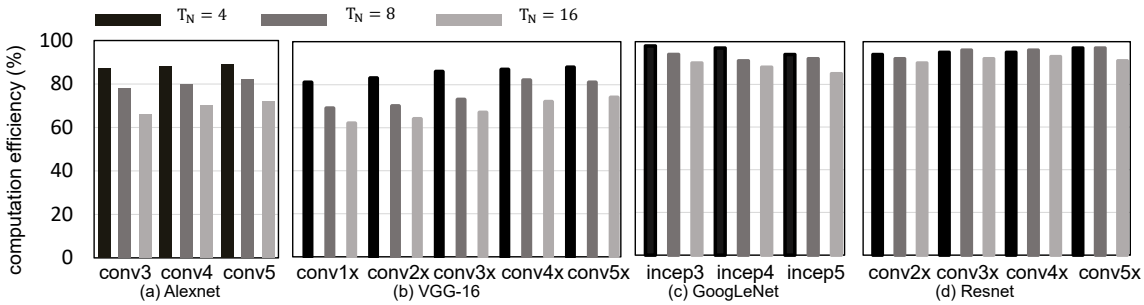


Fig. 11. Computation efficiency

TABLE III
RESOURCE UTILIZATION BREAKDOWN

| | BRAM | DSP | FF | LUT |
|---|---|---|---|---|
| Buffers | 609 | 0 | 0 | 0 |
| TLUT | 0 | 0 | 48894 | 38191 |
| PEs | 0 | 288 | 1088 | 32 |
| CMUX | 0 | 24 | 394 | 43976 |
| Others | 8 | 52 | 18408 | 50177 |
| Total | 617(33%) | 364(%14) | 68784(%12) | 132344(%48) |
| Available | 1824 | 2520 | 548160 | 274780 |

PEs. Section VI-D presents the details of load imbalance problem. Second, the feature map size in the CNN layers cannot divide $T_H$ and $T_W$ evenly. We choose $8 \times 8$ as the output tile size. Take the last convolutional layer of VGG as an example, the feature map size is $14 \times 14$ leading to a 12.5% waste of computation. Third, as mentioned in Section IV-B, we apply pipeline technique in PEs. When the workload is small after pruning, the latency of PE can be bounded by the depth of pipeline. In our implementation, the pipeline depth is 8 cycles. Compared with VGG network, Resnets and GoogLeNet consist of many convolutional layers with $1 \times 1$ kernels, leading to low performance. The speedup of VGG-16 is higher than that of Alexnet, because VGG-16 is a structured and regular network. The kernel size of all layers is $3 \times 3$, however, Alexnet contains many layers in different types.

### C. Resource Utilization Characteristics

Table III shows the resource utilization breakdown with the configuration ($T_H = T_W = 6, T_N = 8$). Block RAMs (BRAM) are mainly used to construct the buffers. The parameters $T_H$ and $T_W$ determine the number of input line buffers. The parameter $T_N$ determines the number of weight buffers. The parameters $T_H$, $T_W$ and $T_N$ determine the number of output line buffers. Each DSP can perform a $16bit \times 16bit$ multiplication operation. The number of DSPs used for the PE array can be calculated as $T_H \times T_W \times T_N$. Both CMUX and TLUT consume the logic resources. CMUX module first uses DSPs to calculate the output channel number according to the index then output which BRAM the results needs to be accumulated to. The size of CMUX depends on how many weights are computed in parallel ($T_N$). TLUT module is constructed by the LUT and flip-flop (FF) on FPGA. In our design, TLUT has a constant number of input wires since we unify the kernel size to $3 \times 3$. Besides, TLUT module needs registers to store all possible sub-input tiles which are implemented using FF. The LUT utilization of TLUT module is a constant and the FF utilization depends on the tile size $T_H \times T_W$.

Figure 10 shows the resource utilization of different configurations obtained from Xilinx Vivado tool (v2017.4). In Figure 10, the LUT utilization increases as the parallelism factor $T_N$ increases because of CMUX. The utilization of BRAMs is mainly determined by the parallelization degree of feature maps ($T_H, T_W$). When $T_N$ is small, BRAM utilization is similar. This indicates that the consumption of BRAM is determined by the input and output data size, instead of the partition factors. We also observe that the FF utilization is almost linear to the tile size.

### D. Computation efficiency

In our implementation, the PE array processes $P_N$ weights with different remainder at the same time. However, the remainder distribution is irregular which can result in load imbalance problem as discussed in Section IV-D. Figure 8 shows the efficiency across different layers with different parallelism factors. We find that the format efficiency decreases when the modular factor $T_N$ becomes larger. Because a large $T_N$ will bridge the gap between the maximum number of valid value and the average number of valid value among different remainders. We find that the efficiency increases as the network goes deeper. This because the number of channels increases as the network goes deeper, which makes the total number of nonzeros larger. A large number of nonzeros can compensate the gap between the maximal and minimal number of remainders. Also, we observe that the computation efficiency of GoogLeNet and Resnet is much higher. This is because the sparsity of these two networks is relatively small which leads to a large number of nonzeros. Besides, the computation efficiency of some layers in GoogLeNet is low. Because the channel number cannot be divided evenly by $T_N$. For example, the output channel number of inception_4b layer in GoogLeNet is 24 which is not a multiple of $T_N = 16$. In conclusion, our dataflow can maintain high computation efficiency for different configurations and networks.

### E. Scalability and comparison with GPU

We also test our design on ZC706 platform to demonstrate the scalability. Our implementation is operated at 166MHz fre-



Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on November 27,2020 at 14:13:41 UTC from IEEE Xplore. Restrictions apply.

TABLE IV
COMPARISON WITH GPU PLATFORM USING RESNET

| Device | TitanX[1] | TitanX[2] | ZC706 | ZCU102 |
|---|---|---|---|---|
| Technology | 28 nm | 28 nm | 28 nm | 16 nm |
| Frequency (MHz) | 1075 | 1075 | 166 | 200 |
| Precision | 32bits float | 32bits float | 16bits fixed | 16bits fixed |
| conv average (GOP/s) | 212(eff.) | 119 | 134 | 291 |
| Power (W) | 130 | 134 | 9.4 | 23.6 |
| Energy efficiency (GOP/s/W) | 1.63 | 0.88 | 12.66 | 12.33 |

[1] The sparse network is considered as the dense network and accelerated using CuDNN.
[2] The sparse network is accelerated using CuSparse.

quency on this platform. ZC706 has 900 DSPs, 1090 BRAMs and 305K logic cells. We set the configuration parameter as ($T_H = T_W = 8, T_N = 8$) and achieve 134.2 GOP/s on Resnet which means our design can be scaled to different platforms. Besides, we conduct a comparison between GPU and FPGA platforms. We measure the performance of dense Resnet using the latest CuDNN on NVIDIA TitanX platform. To make a fair comparison, we also apply CuSparse library to accelerate sparse Resnet. The sparse version shows a lower performance because of the memory uncoalesing problem. In conclusion, our design shows 1.37X speedup and 7.56X energy-efficiency compared with TitanX platform.

## VII. RELATED WORK

**Architecture for dense CNNs on FPGAs**. Prior efforts to accelerate CNNs have shown substantial successes on FPGAs. Ma [9] et al. make an in-depth analysis of loop optimization techniques in spatial convolution, which includes loop tiling, loop unrolling and loop interchange. Zhang [8] et al. focus on reducing the on-chip memory bandwidth requirement. [13] proposed a novel Concatenate-and-Pad (CaP) technique, which improves OaA significantly by reducing the wasted computation on the padded pixels. Wei et al. [15] implemented CNN on an FPGA using a systolic array architecture, which can achieve high clock frequency under high resource utilization. Zhang et al. [14] proposed AccDNN tool which included high-quality RTL network layer IPs, a fine-grained layer-based pipeline architecture and an automatic design space exploration tool. Besides, there are some works that implement fast algorithms to further accelerate CNNs [25, 28–33].

**Architecture for sparse CNNs on ASICs**. Recently, some works explore the dataflow and architecture to accelerate sparse CNNs on ASICs. Han et al. [34] proposed EIE CNN accelerator which exploits sparsity both in input feature maps and filters but only focused on the fully-connected layer. The fully-connected layer is computed using matrix multiplication, in EIE design, the matrix is stored in CSC format and multiple columns are computed in parallel. Parashar et al. proposed SCNN accelerator with a dataflow named PT-IS-CP (planar-tiled input-stationary Cartesian-product) [22]. Zhang et al. [23] presented Cambricon-X accelerator which applies step indexing techniques. In Cambricon-X design, the nonzeros in the same row is divided into multiple segments with the same size in subsequent addresses. And the row that contains nonzeros less than the size will be aligned to the size. In recent years, some ASIC accelerators apply hardware-software design that prune the weight with structured pattern [19–21].

## VIII. CONLUSION

In this work, we propose an FPGA accelerator for sparse CNNs. We first propose a weight-oriented dataflow that exploits element-matrix multiplication. Based on this dataflow, we design an FPGA architecture mainly composed of a tile look-up table and a channel multiplexer. Besides, we propose a weight layout where the weights calculated in parallel are stored continuously. To cooperate with the weight layout, a channel multiplexer is inserted to locate the address which can ensure no data access conflict. Experiments demonstrate that our accelerator can achieve 223.4-309.0 GOP/s for the modern CNNs on Xilinx ZCU102, which provides a 2.4x-12.9x speedup over previous dense CNN FPGA accelerators.

## IX. ACKNOWLEDGEMENT

This work is supported by Alibaba Group through Alibaba Innovative Research (AIR) Program. This work is also supported by Beijing Natural Science Foundation (No. L172004), Municipal Science and Technology Program under Grant Z181100008918015.

## REFERENCES

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *ICCV*, 2015.
[2] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *CVPR*, 2014.
[3] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," in *arXiv preprint arXiv:1409.1556*, 2014.
[4] J. Redmon *et al.*, "You only look once: Unified, real-time object detection," in *CVPR*, 2016.
[5] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *NIPS*, 2015.
[6] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," in *ICLR*, 2015.
[7] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *FPGA*, 2015.
[8] J. Zhang and J. Li, "Improving the performance of OpenCL-based fpga accelerator for convolutional neural network," in *FPGA*, 2017.
[9] Y. Ma, Y. Cao, S. Vrudhula, and J. Seo, "Optimizing loop operation and dataflow in FPGA acceleration of deep convolutional neural networks," in *FPGA*, 2017.
[10] J. Qiu *et al.*, "Going deeper with embedded FPGA platform for convolutional neural network," in *FPGA*, 2016.
[11] Y. Guan *et al.*, "FP-DNN: An Automated Framework for Mapping Deep Neural Networks onto FPGAs with RTL-HLS Hybrid Templates," in *FCCM*, 2017.
[12] N. Suda *et al.*, "Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks," in *FPGA*, 2016.
[13] H. Zeng, R. Chen, C. Zhang, and V. Prasanna, "A framework for generating high throughput CNN implementations on FPGAs," in *FPGA*, 2018.
[14] X. Zhang *et al.*, "AccDNN: An IP-Based DNN Generator for FPGAs," in *FCCM*, 2018.
[15] X. Wei *et al.*, "Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs," in *DAC*, 2017.
[16] Q. Xiao *et al.*, "Exploring heterogeneous algorithms for accelerating deep convolutional neural networks on FPGAs," in *DAC*, 2017.



Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on November 27,2020 at 14:13:41 UTC from IEEE Xplore. Restrictions apply.

[17] S. Han *et al.*, "ESE: Efficient speech recognition engine with sparse lstm on FPGA," in *FPGA*, 2017.
[18] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *NIPS*, 2012.
[19] C. Ding *et al.*, "CIRCNN: accelerating and compressing deep neural networks using block-circulant weight matrices," in *MICRO*, 2017.
[20] D. Chunhua *et al.*, "PERMDNN: Efficient Compressed DNN Architecture with Permuted Diagonal Matrices," in *MICRO*, 2018.
[21] Z. Xuda *et al.*, "Cambricon-S: Addressing Irregularity in Sparse Neural Networks through a Cooperative Software-Hardware Approach," in *MICRO*, 2018.
[22] A. Parashar *et al.*, "SCNN: An accelerator for compressed-sparse convolutional neural networks," in *ISCA*, 2017.
[23] S. Zhang *et al.*, "Cambricon-X: An accelerator for sparse neural networks," in *MICRO*, 2016.
[24] S. Li *et al.*, "An FPGA Design Framework for CNN Sparsification and Acceleration," in *FCCM*, 2017.
[25] J. H. Ko *et al.*, "Design of an Energy-Efficient Accelerator for Training of Convolutional Neural Networks using Frequency-Domain Computation," in *DAC*, 2017.
[26] Y. Jia *et al.*, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.
[27] C. Zhang *et al.*, "Caffeine: towards uniformed representation and acceleration for deep convolutional neural networks," in *ICCAD*, 2016.
[28] U. Aydonat, S. O'Connell, D. Capalija, A. C. Ling, and G. R. Chiu, "An OpenCL deep learning accelerator on arria 10," in *FPGA*, 2017.
[29] C. Zhang and V. Prasanna, "Frequency Domain Acceleration of Convolutional Neural Networks on CPU-FPGA Shared Memory System," in *FPGA*, 2017.
[30] A. Podili, C. Zhang, and V. Prasanna, "Fast and Efficient implementation of Convolutional Neural Networks on FPGA," in *ASAP*, 2017.
[31] L. Lu, Y. Liang, Q. Xiao, and S. Yan, "Evaluating fast algorithms for convolutional neural networks on FPGAs," in *FCCM*, 2017.
[32] L. Lu and Y. Liang, "SpWA: an efficient sparse winograd convolutional neural networks accelerator on FPGAs," in *DAC*, 2018.
[33] Y. Liang, L. Lu, Q. Xiao, and S. Yan, "Evaluating Fast Algorithms for Convolutional Neural Networks on FPGAs," *TCAD*, 2019.
[34] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: efficient inference engine on compressed deep neural network," in *ISCA*, 2016.



Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on November 27,2020 at 14:13:41 UTC from IEEE Xplore. Restrictions apply.