IPSJ Transactions on System LSI Design Methodology Vol.12 22–37 (Feb. 2019)

[DOI: 10.2197/ipsjtsldm.12.22]

Regular Paper

Parallelism-flexible Convolution Core for Sparse Convolutional Neural Networks on FPGA

Salita Sombatsiri[1,2,a)] Seiya Shibata[1,b)] Yuki Kobayashi[1,c)] Hiroaki Inoue[1,d)]
Takashi Takenaka[1,e)] Takeo Hosomi[1,f)] Jaehoon Yu[2,g)] Yoshinori Takeuchi[3,h)]

Received: July 1, 2018, Revised: August 31, 2018,
Accepted: October 22, 2018

Abstract: The performance of recent CNN accelerators falls behind their peak performance because they fail to maximize parallel computation in every convolutional layer from the parallelism that varies throughout the CNN. Furthermore, the exploitation of multiple parallelisms may reduce calculation-skip ability. This paper proposes a convolution core for sparse CNN that leverages multiple types of parallelism and weight sparsity efficiently to achieve high performance. It alternates dataflow and scheduling of parallel computation according to the available parallelism of each convolutional layer by exploiting both intra- and inter-output parallelism to maximize multiplier utilization. In addition, it eliminates redundant multiply-accumulate (MACC) operations due to weight sparsity. The proposed convolution core enables both abilities with ease of dataflow control by using a parallelism controller for scheduling parallel MACCs on the processing elements (PEs) and a weight broadcaster for broadcasting non-zero weights to the PEs according to the scheduling. The proposed convolution core was evaluated on 13 convolutional layers in a sparse VGG-16 benchmark. It outperforms the baseline architecture for dense CNN that exploits intra-output parallelism by 4x speedup. It achieves 3x effective GMACS over prior arts of CNN accelerator in total performance.

Keywords: CNN, multi-parallelism, flexible parallelism, sparsity

1. Introduction

In modern artificial intelligence (AI) platforms, data processing at the edge and on embedded systems requires high-performance computing devices. Convolutional Neural Network (CNN), which is one of the most vigorous AI algorithms, evolves day-by-day for a vast number of applications especially in image and video analytic domain, such as surveillance systems and autonomous driving, because of their remarkable classification performance shown in several image recognition studies [1], [2], [3] on ImageNet benchmark [4]. The processing of these applications usually takes place at the edge (near sensor, such as camera) or on embedded systems in order to achieve a real-time response. Unfortunately, CNN comes with the cost of an excessive computation that becomes critical for real-time and low-power inference processing on both edge and embedded systems. Most processing time of CNN is consumed by convolutional layers. In order to accelerate its computation, CNN requires high-performance and low-power accelerator to deliver its superior ability, and programmability of the device in response to its rapid growth. For that reason, FPGA is one of the promising platforms for real-time CNN acceleration at the edge and on embedded systems.

High-performance CNN accelerators bring about real-time ability with the exploitation of four major techniques. First, data-reuse maximization focuses on reusing input feature maps (IFMs), kernels and output feature maps (OFMs). It is employed by several low-power architectures [5], [6], [7], [8] because it reduces high-latency and energy-consuming external memory access. Second, data precision minimization aims to reduce data bitwidth while the recognition accuracy is maintained [9], [10]. Third, calculation-skip maximization reduces the calculation by omitting zero-operand multiply-accumulate (MACC), which is the result from weight pruning process [11], [12]. This allows several architectures to achieve performance improvement by the degree of sparsity [8], [13]. Fourth, parallel calculation maximization leverages various types of parallelism in CNN. Recent publications exploit specific types of parallelism and schedule the computation accordingly to maximally utilize the multipliers in processing elements (PEs) [6], [7], [14]. Typically, the accelerators exploit more than one techniques.

There exist two main problems that prevent CNN accelerators from achieving superior performance. First, most CNN accelerators fail to maximize parallel calculation of all convolutional layers due to the fact that the dominant type of parallelism varies by the size and number of IFMs and OFMs, while the dataflow and computation scheduling (mapping parallel operations to the multipliers) remain fixed throughout all layers. Specifically,

[1] NEC Corporation, Kawasaki, Kanagawa 211–8666, Japan
[2] Osaka University, Suita, Osaka 565–0871, Japan
[3] Kindai University, Higashi-Osaka, Osaka 577–8502, Japan
[a)] s-sombatsiri@bp.jp.nec.com
[b)] s-shibata@ax.jp.nec.com
[c)] y-kobayashi@hq.jp.nec.com
[d)] h-inoue@ce.jp.nec.com
[e)] takenaka@aj.jp.nec.com
[f)] hosomi@ah.jp.nec.com
[g)] yu.jaehoon@ist.osaka-u.ac.jp
[h)] takeuchi@ele.kindai.ac.jp

the accelerators have difficulty in adjusting their dataflow and scheduling according to layer specification, which results in low multiplier utilization and low performance in some layers. For example, even though the architecture proposed in Ref. [7] exploits various types of parallelism, it cannot achieve high multiplier utilization in the first layer because of the fixed dataflow and scheduling. To resolve this problem, flexible scheduling, aka flexible parallelism, is required to improve multiplier utilization with other types of parallelism according to the layer specification. The second problem addresses the difficulty in effectively employing calculation-skip maximization and parallel calculation maximization techniques, specifically flexible parallelism, at the same time. For example, parallelizing multiplications comprising one output may occupy more multipliers, but it cannot fully leverage zero-skipping without complicating data control because the scheduling is regulated by the pre-defined dataflow. As a consequence, the dataflow to exploit flexible parallelism may reduce calculation-skip ability.

We propose a parallelism-flexible convolution core for sparse CNN [*1] for accelerating convolutional layers. This paper includes the following contributions:

(1) we introduce a flexible parallelism concept to maximize multiplier utilization;
(2) we propose a parallelism-flexible convolution core for sparse CNN that efficiently exploits weight sparsity by skipping zero-operand computation;
(3) we extend the determination of parallelism in effect and degree of parallelism, P, to maximize multiplier utilization in all convolutional layers of the sparse CNN;
(4) to show the effectiveness of our method, we implemented and evaluated the parallelism-flexible convolution core for sparse CNN on Intel's Arria10 and Stratix10 FPGAs.

Parallelism in effect and degree of parallelism refer to types of parallelism that the proposed convolution core exploits in computing a certain layer and degree of inter-output parallelism (the number of OFMs to be computed simultaneously), respectively.

The rest of this paper is organized as follows. Section 2 describes CNN and the prior arts of CNN accelerators. Section 3 introduces flexible parallelism, and explains architecture organization and operations of the proposed parallelism-flexible convolution core in leveraging both multiple types of parallelism and sparsity. In Section 4, we present the effectiveness of the proposed convolution core and make a comparison with prior FPGA-based CNN accelerators. We conclude this paper in Section 5.

2. Related Studies

First, this section explains the terminology, algorithm, and parallelism of CNN. Then, it describes prior arts of CNN accelerators by four acceleration techniques.

2.1 Preliminary of CNN

2.1.1 Terminology of CNN

A CNN consists of four kinds of layers: (1) convolutional layer, which functions as a feature extractor; (2) pooling layer, which subsamples the extracted features; (3) normalization layer, which normalizes feature correlations; (4) fully-connected layer, which produces non-linear activations for regression or classification problems. This paper focuses on accelerating the multi-channel two-dimensional convolution of the convolutional layers, which is computation-intensive and time-consuming.

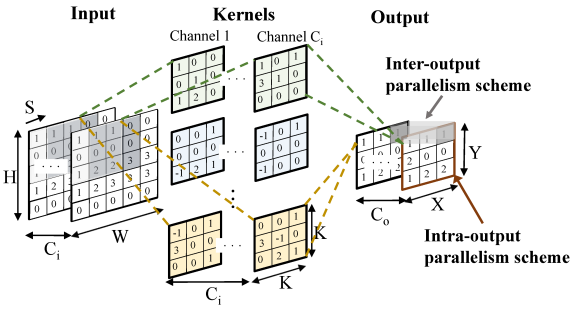


Fig. 1 The computation of a convolutional layer and its parallelism.

Figure 1 illustrates the terminology of a convolutional layer, where H and W are height and width of an IFM, C_i is the number of input channels, S is stride (the number of pixels to shift the kernel in convolution), K is kernel size (a kernel includes $C_i \times K \times K$ weights), X and Y are height and width of an OFM, and C_o is the number of output channels, which is equal to the number of kernels. Each activation of the OFMs is computed by a deep nested loop according to the following equations:

$$A_o^v(x,y) = f(F_o^v(x,y)) \tag{1}$$

$$F_o^v(x,y) = b^v + \sum_{t=1}^{C_i}\sum_{m=1}^{K}\sum_{n=1}^{K} k_v^t(m,n) \times F_i^t(x \times S + m, y \times S + n) \tag{2}$$

where $A_o^v(x,y)$ is the activation at position (x,y) of the OFM v, f is an activation function, $F_o^v(x,y)$ is the result of convolution at position (x,y) between the IFMs and kernel v, b^v is a bias of kernel v, $k_v^t(m,n)$ is the weight at position (m,n) in channel t of kernel v, and $F_i^t(x \times S + m, y \times S + n)$ is the activation at position $(x \times S + m, y \times S + n)$ of IFM t.

2.1.2 Parallelism in CNN

There are four types of parallelism incorporated with convolutional layers: inter-layer, inter-output, intra-output, and operation-level parallelism. Inter-layer parallelism is the parallelism that executes the convolution of multiple layers in a pipeline manner. The latter three types comprise an intra-layer parallelism, in which the MACCs within the same layer are computed in parallel. The inter- and intra-output parallelism are the parallelism between multiple OFMs (the C_o axis in Fig. 1) and output activations within an OFM (the X-Y plane in Fig. 1), respectively. The operation-level parallelism is the most fine-grained type that parallelizes the multiplications of the same output activation. It occupies most multipliers when accelerating a typical dense CNN, in which all weights are non-zero.

The dominant type of parallelism of each layer varies throughout the CNN by layer specification, such as size and number of OFMs. When the size of OFMs is large, the intra-output parallelism is efficient in terms of multiplier utilization. However, multiplier utilization decreases as the OFMs become smaller. In this case, inter-output parallelism can complement the small amount

[*1] A part of this work is proposed in Ref. [15].

of intra-output parallelism, hence, increase multiplier utilization.

2.2 Prior Arts of CNN Accelerator

This section explains the prior-art accelerators in terms of four techniques that they exploit to achieve real-time performance.

2.2.1 Data-reuse Maximization

Recent CNN accelerators exploit the weight sharing property and data locality within a convolutional layer to maximize data-reuse. They reuse IFMs, kernels, and OFMs in on-chip memory to reduce high-latency and energy-consuming external memory access through dataflow pattern and data tiling. Hence, data reuse improves performance and reduces power consumption.

Efficient dataflow promotes data reuse in four major patterns. First, weight-stationary dataflow pattern maximizes weight reuse in the PEs, and shifting IFMs and OFMs to the neighboring PEs [16], [17], [18], [19]. Second, the output-stationary dataflow pattern maximizes output data reuse by accumulating the OFMs locally in the PEs, while circulating the weights and/or IFMs during the computation [6], [20], [21], [22]. Third, global reuse dataflow pattern reuses both weights and IFMs from the global on-chip memory [7], [23], [24], [25]. Fourth, row-stationary dataflow maximally reuses weights, IFMs, and OFMs locally in a row unit [5].

Data tiling partitions and processes IFMs in small tiles [7], [23], [26] to reuse IFMs with all kernels. The SCNN [8] maps data tiles onto its PEs in order to reuse both IFMs and OFMs locally without inter-layer external memory access.

The proposed convolution core exploits data tiling and output-stationary dataflow pattern to distribute kernels and reuse IFMs and OFMs locally. Both techniques enable calculation-skip without complex dataflow control to access IFMs or sparse weights, while the execution time is reduced by the degree of sparsity.

2.2.2 Data Precision Minimization

Data precision minimization is achieved through a quantization method that reduces the number of required bits for CNN computation without the loss of accuracy. Several techniques quantize arithmetic precision of kernels, IFMs, and OFMs from floating point to a few bits of fixed-point precision [9], [10], [12], [14], [27]. This optimization lowers both computational resource per one MACC and storage requirement of the customized hardware. The proposed convolution core computes CNN with 16- and 32-bit fixed-point precision for multiplication and accumulation of MACC, respectively.

2.2.3 Calculation-skip Maximization

Calculation-skip maximization omits zero-operand MACCs from the sparsity in IFMs and weights of the kernels. It reduces the number of MACCs involved with non-zero weights and can accelerate CNN inference computation by the degree of sparsity. Sparsity in IFMs comes from activation functions such as Rectified Linear Units. Weight pruning process introduces sparsity in weights by zeroing out weight values with the trade-off between the number of remaining weights and recognition accuracy. Many state-of-the-art studies have shown that more than 80% of weight sparsity is possible without jeopardizing the accuracy [11], [12].

Unlike dense CNN, accessing weights and IFMs of sparse CNN has irregular patterns that may incur complex control. Recent accelerators exploit weight sparsity or activation sparsity or both. The ones that exploit weight sparsity usually use kernels in sparse format to extract non-zero weights and skip MACCs having zero-valued weights efficiently with the output-stationary dataflow pattern [26], [28]. The architectures that leverage both weight and IFM sparsity usually include a zero-detection mechanism to dynamically skip zero-operand multiplication [8], [13]. As a result, these architectures achieved performance improvement over the dense CNN accelerator. The proposed convolution core assumes a sparse CNN model and leverages weight sparsity from a compressed CNN model in a straightforward way.

2.2.4 Parallel Calculation Maximization

To maximize parallel calculation, CNN accelerators schedule MACCs from various types of parallelism onto their vast amount of multipliers. The reconfigurable processor array maps intra-output parallelism onto its PEs [6]. Many high-performance architectures schedule multiple types of parallelism, i.e., intra-output, inter-output, and operation-level parallelism, onto multipliers by rows, columns, or groups of PEs [7], [14], [29]. However, they cannot achieve high performance in terms of giga operations per second (GOPS) in every layer because the scheduling is fixed, while the dominant parallelism in each layer usually varies across the CNN with different layer specifications, such as the size and number of OFMs.

To further increase parallel calculation in every layer, the architecture should schedule MACCs onto the multipliers flexibly according to the dominant parallelism of each layer. The FlexFlow architecture [30] adjusts its scheduling of multiple types of parallelism to improve multiplier utilization layer by layer. Even though it achieves near peak performance, it neither supports the compressed CNN model nor exploits sparsity efficiently because it exploits operation-level parallelism. It is difficult to skip zero-operand MACCs while exploiting multiple types of parallelism, especially operation-level parallelism, without either complex dataflow control mechanism or wasting a vast number of multiplier cycles. That is because irregular sparsity pattern trickers non-deterministic weight, IFM, and OFM access.

3. Parallelism-flexible Convolution Core for Sparse CNN Accelerator

First, this section explains the format of the compressed CNN model that packs a sparse CNN. Then, the proposed CNN accelerator is described. Finally, it proposes the concept of flexible parallelism and the parallelism-flexible convolution core, which effectively leverages weight sparsity and flexibly alternates the scheduling of multiple types of parallelism layer by layer.

3.1 Compressed CNN Model

To reduce the required bandwidth in reading CNN model to CNN accelerator and simply exploit weight sparsity, kernels of a sparse CNN model from quantization and weight pruning is compressed into a channel-major modified compressed sparse column format [13] layer by layer. Each channel of kernels in each convolutional layer is compressed as a non-zero weight vector, **w**, which includes non-zero weight elements, and a leading-zero vector, **z**, which includes the number of zero-valued weights pre-

ceding the non-zero weight at the same vector index as **w**.

For example, a convolutional layer that contains three kernels, each of which includes two channels of 3 × 3-weights, is compressed as shown in **Fig. 2**. The notation w_i represents the i^{th} non-zero weight. The kernels are compressed channel by channel. The non-zero weight vector of channel 1, $\mathbf{w_{c1}}$, includes non-zero weights in order as shown by the bold arrow (written in channel 1 of kernel 1) from kernel 1 to kernel 3. The corresponding leading-zero vector, $\mathbf{z_{c1}}$, includes the number of leading zeros of w_1, w_2, w_3, and so on, respectively. The number of leading zeros is counted continuously regardless of different kernels. For that reason, the number of leading zeros of w_8 is 3 since there is one 0 after w_7 in kernel 2 and two 0 before w_8 in kernel 3. The weights of channel 2 are compressed similarly.

3.2 Overview of CNN Accelerator

Figure 3 (a) illustrates an overview of the proposed CNN accelerator, which includes five key components. The memory controller reads and writes data from/to external memory, such as DRAM, through a DDR memory interface with a 512-bit data bus. It forwards incoming data, including compressed CNN model, layer specification, parallelism in effect and degree of parallelism (denoted as Parallelism eff. in the figure; see Section 3.3.3 for detail), and IFMs, to the CNN controller. Section 4.2.1 discusses the bandwidth in more details. To perform convolution in a layer-wise manner, the CNN controller controls the execution of the accelerator from the layer specification and parallelism in effect, forwards the compressed CNN model to the convolution core, and manages the incoming IFMs using line buffer. The convolution core performs convolution and stores the intermediate results in the partial sum buffer in Fig. 3 (b). The pooling and f unit include multiple arithmetic logic units (ALUs), which subsample and apply activation function to the OFMs, and activation buffer, which stores the output activations. Finally, the activations are either moved to external memory or reused as IFMs of the next layer.

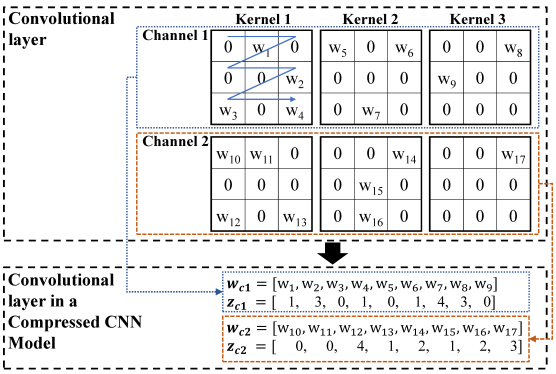


Fig. 2 An example of compressing a convolutional layer to a compressed CNN model.

Since convolutional layers consume most CNN computation time, this paper focuses on the convolution core that accelerates the convolutional layers. It efficiently leverages both multiple types of parallelism and weight sparsity of the compressed CNN model according to the size of OFMs. The proposed convolution core is applicable to various sizes of IFMs, sizes of OFMs, numbers of input channels, numbers of output channels, kernel sizes, and strides. Other CNN processing, i.e., activation function, pooling layers, and fully-connected layers, are lightweight, and hence they can be computed on either general purpose processors or specialized hardware such as EIE [13].

3.3 Convolution Core

To overcome the problems in exploiting multiple types of parallelism and its integration with calculation-skip, the proposed convolution core flexibly adjusts its dataflow and scheduling to multiple types of parallelism, i.e., intra- and inter-output parallelism, with various degrees of parallelism layer by layer, and eliminates the operations related to zero-valued weights through output-stationary dataflow pattern. Compared to the conventional accelerators, the proposed convolution core uses the parallelism controller and the weight broadcaster to enable such abilities. The weight broadcaster and parallelism controller compensate the decreased multiplier occupancy due to reduced intra-output parallelism by broadcasting different kernels and assigning repeated OFM coordinates to the PE grid, respectively, to increase inter-output parallelism according to the degree of parallelism, P. At the same time, the weight broadcaster distributes only non-zero weights and their indices, which are calculated from the leading-zero vector, to the PE grid. The irregular access to IFMs due to weight sparsity is made simple with local indexing to the addresses in local input buffer near PEs. Hence, the output-stationary dataflow pattern that is regulated by our parallelism controller and the weight broadcaster efficiently integrates both flexible parallelism and calculation-skip.

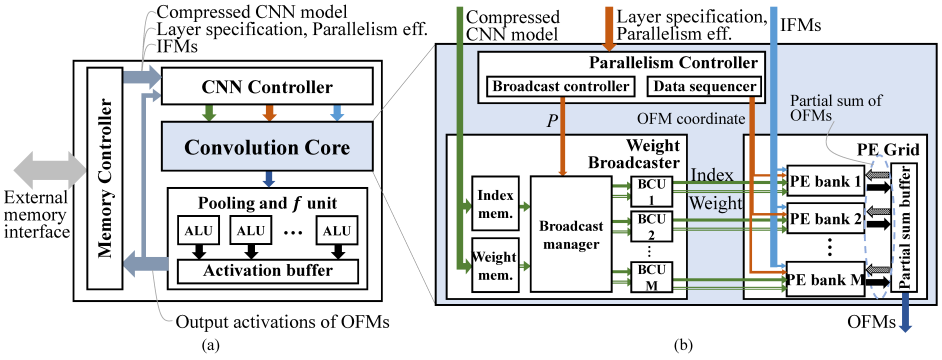


Fig. 3 Architecture of the proposed CNN accelerator. (a) An overview architecture; (b) Architecture of the proposed parallelism-flexible convolution core for sparse CNN.

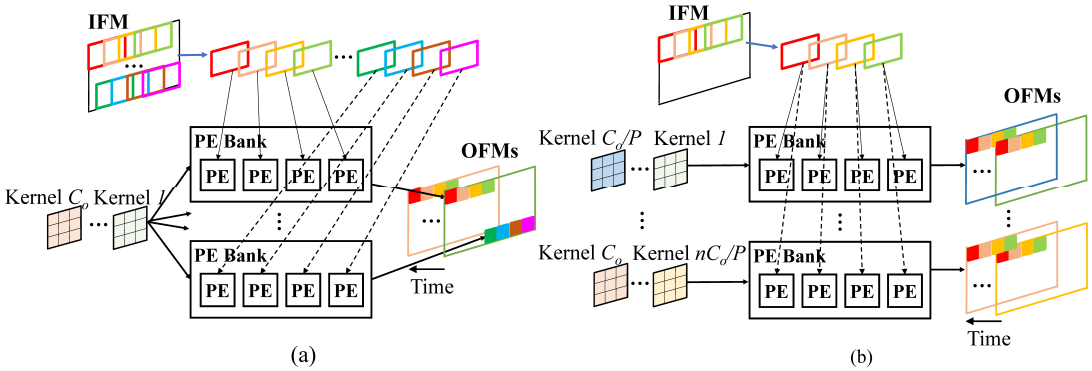


Fig. 4 The flexible parallelism concept. (a) Exploitation of intra-output parallelism; (b) Exploitation of intra- and inter-output parallelism.

3.3.1 Flexible Parallelism Concept

To maximize parallel calculation (multiplier utilization) in every convolutional layer throughout the CNN, flexible parallelism changes dataflow and scheduling of the convolution layer by layer. **Figure 4** illustrates an example of the MACC scheduling. A PE bank means a group of PEs that convolute the IFM pixels with the same kernel. The scheduling when the architecture exploits only intra-output parallelism as parallelism in effect with $P = 1$ is shown in Fig. 4 (a). All PEs convolute IFM's sliding windows with the same kernel to compute distinct OFM pixels simultaneously and convolute with all kernels consecutively to compute all OFMs. The scheduling when the architecture exploits both intra- and inter-output parallelism simultaneously, aka multi-parallelism, as parallelism in effect with $P > 1$ is shown in Fig. 4 (b). PEs within a PE bank compute distinct OFM pixels with the same kernel at the same time to realize intra-output parallelism, while different PE banks compute distinct OFMs with P different kernels to realize inter-output parallelism and each PE bank computes OFMs with $\frac{C_o}{P}$ kernels sequentially. Depending on P, several PE banks convolute distinct OFM pixels with the same kernel to increase intra-output parallelism when P is small, and convolute IFM with more distinct kernels to increase inter-output parallelism when P is large.

The parallelism in effect and degree of parallelism are determined in advance in order to maximize multiplier utilization (see Section 3.3.4). In addition, if the size of OFMs or P is large, IFMs and OFMs are partitioned into tiles (data tiling) so that multipliers and buffer can accommodate parallel MACCs and data, respectively. For example, assuming that there are 50 PEs. If an OFM consists of 100 output activations, the OFM is partitioned into two tiles to be able to map on 50 PEs in case of $P = 1$. As P grows larger, the OFM is further partitioned into four tiles in case of $P = 2$ and so on. The PEs process one tile at a time.

3.3.2 Operations of the Convolution Core

The OFMs of each layer are computed as shown in Algorithm 1. First, IFMs and OFMs are divided into T equal data tiles. Then, the algorithm loops through all C_i IFMs of each tile in the second loop in order to maximally reuse each IFM. To implement multi-parallelism, the proposed convolution core flexibly unrolls the third and fourth loop layer by layer according to P. Unrolling the third loop parallelizes the convolution of P different kernels to realize inter-output parallelism. Hence, P different OFMs are computed on PEs in P different PE banks simultaneously. Each PE is assigned to compute $\lceil \frac{C_o}{P} \rceil$ kernels. The fourth loop iterates all outputs at different OFM coordinates, $F_o(x, y)$ of tile T_s. Unrolling this loop and mapping each output on different PEs realize intra-output parallelism. Next, in line 5, the algorithm iterates over each kernel, K_v^t, in the set of kernels assigned to sequentially compute within one PE in the third loop, which is denoted as K_u. Finally, the most inner loop sequentially accumulates the result at coordinate (x, y) of OFM v, $F_o^v(x, y)$, with the multiplication results of the non-zero weight elements, $k_v^t(m, n)$, and the corresponding IFM, $F_i^t(x \times S + m, y \times S + n)$.

Algorithm 1 Processing of a convolutional layer on the proposed convolution core

```
1:  for s from 1 to T do                                   ▷ Loop all Tiles
2:      for t from 1 to C_i do                             ▷ Loop all IFMs
3:          for u from 1 to P do                           ▷ Loop all degree of parallelism
4:              for F_o(x, y) ∈ T_s do                     ▷ Loop all outputs in tile
5:                  for K_v^t ∈ K_u do                     ▷ Loop ⌈C_o/P⌉ kernels
6:                      for k_v^t(m, n) ∈ K_v^t and k_v^t(m, n) > 0 do
7:                                                         ▷ Loop all non-zero weights in kernel
8:                          F_o^v(x, y) += k_v^t(m, n) × F_i^t(x × S + m, y × S + n)
9:                      end for
10:                 end for
11:             end for
12:         end for
13:     end for
14: end for
```

3.3.3 Architecture Organization

The architecture of the proposed convolution core is illustrated in Fig. 3 (b). It receives compressed CNN model, Parallelism Eff., and IFMs as input. The layer specification includes the size and number of kernels, IFMs, and OFMs. Parallelism Eff. includes parallelism in effect and degree of parallelism.

Parallelism Controller

The parallelism controller is responsible for alternating the dataflow on the convolution core. It is composed of a broadcast controller and a data sequencer. Both work according to the parallelism in effect and the degree of parallelism, P.

The broadcast controller forwards P to the weight broadcaster to control the dataflow of kernels. It forwards 1 as P if the parallelism in effect is intra-output parallelism and P, where $P > 1$, if the parallelism in effect is multi-parallelism with the degree of

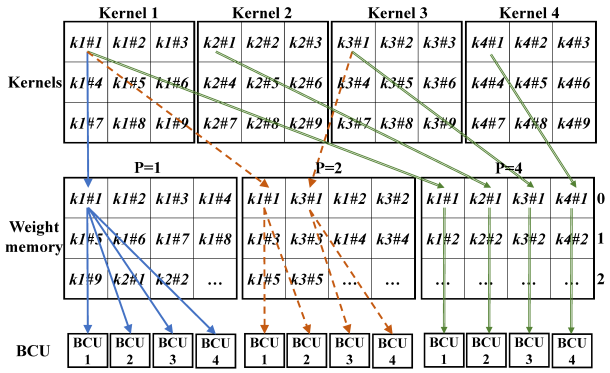


Fig. 5 Example of weight arrangement of four kernels in weight memory, so that BCUs can broadcast weights from different kernels at the same time.

parallelism P.

The data sequencer alternates the dataflow of IFMs through the assignment of OFM coordinates to be computed by each PE. For intra-output parallelism ($P = 1$), the data sequencer assigns different coordinates to all PEs. For multi-parallelism, the data sequencer assigns different OFM coordinates to PEs in $\lfloor \frac{M}{P} \rfloor$ PE banks, where M is the number of PE banks, and duplicates the same coordinates for P times. For example, assuming that $P = 2$, the OFM coordinates assigned to PE bank #1 to PE bank #$(\frac{M}{2})$ are different, but are the same as the ones assigned to PE bank #$(\frac{M}{2} + 1)$ to PE bank #M. If data tiling is necessary, the data sequencer repeats the coordinate assignment process for all tiles after the convolution of the previous tile has completed.

Weight Broadcaster

The weight broadcaster is composed of a weight memory, an index memory, a broadcast manager, and multiple broadcast units (BCUs). First, the compressed CNN model of each layer is loaded into the weight memory and index memory (Weight mem. and Index mem. in Fig. 3 (b), respectively) channel by channel. Next, upon the completion of storing IFM into the local input buffer, the broadcast manager reads **w** and **z** of a channel from the memories and distributes them to BCUs according to P. Finally, each BCU decompresses the compressed CNN model from **w** and **z** and broadcasts them consecutively to a PE bank.

For ease of distributing the compressed CNN model to BCUs in order to exploit multi-parallelism, **w** and **z** are re-ordered in advance according to P in such a way that weights and indices from P different kernels can be read at the same time. When $P = 1$, all BCUs broadcast the same weight value, so the weights and indices in one channel of all kernels are ordered contiguously. On the other hand, when $P > 1$, the weights and indices from different kernels that must be broadcasted at the same time are ordered in the same memory word. **Figure 5** illustrates an example of weight arrangement in the weight memory and weight distribution to BCUs when assuming that there are four BCUs, one memory word stores four weights, and P equals to 1, 2, and 4. The weights are re-ordered and distributed as follows:

- When $P = 1$: the first memory word contains $k1\#1$, $k1\#2$, $k1\#3$, and $k1\#4$. First, the weight $k1\#1$ is distributed to all BCUs, then, followed by $k1\#2$, and so on.
- When $P = 2$: the first memory word contains $k1\#1$, $k3\#1$, $k1\#2$, and $k3\#2$. First, the weight $k1\#1$ is distributed to BCU#1 and BCU#2 and the weight $k3\#1$ is distributed to BCU#3 and BCU#4 at the same time, then followed by $k1\#2$ and $k3\#2$, and so on.
- When $P = 4$: the first memory word contains $k1\#1$, $k2\#1$, $k3\#1$, and $k4\#1$. The weight $k1\#1$, $k2\#2$, $k3\#3$, and $k4\#4$ are distributed to BCU#1 through BCU#4, respectively.

Consequently, multiple kernels can be convoluted simultaneously when $P > 1$. Hence, the weight broadcaster can alter the dataflow of kernels to enable multi-parallelism.

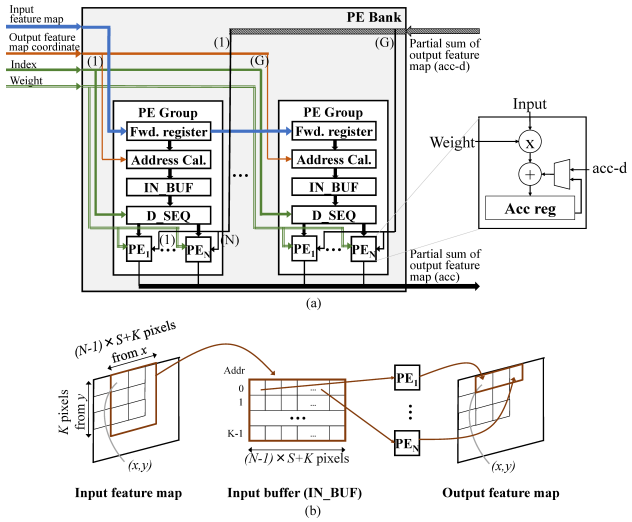


Fig. 6 Architecture of a PE bank. (a) an overview architecture of a PE bank; (b) data layout of the local input buffer (IN_BUF).

To leverage sparsity, each BCU decompresses the compressed CNN model by extracting only non-zero weights from **w** and accumulates their indices from **z** in order. Then, non-zero weights and indices are broadcasted to a PE bank consecutively so that the PEs continuously perform MACCs related to non-zero weights while the ones related to zero-valued weights are skipped.

Processing Element Grid

A PE grid consists of multiple PE banks that perform MACCs, and a partial sum buffer that stores accumulation results of the previous input channels. One PE bank is connected to one BCU, so the number of PE banks and the number of BCUs are equal. Each PE bank receives IFMs from CNN controller, OFM coordinates from data sequencer, and pairs of weight and index from the corresponding BCU.

As shown in **Fig. 6** (a), a PE bank includes G groups of PEs, aka PE groups, that compute different OFM pixels. Every PE group within a PE bank consumes the same pair of weight and index but unique OFM coordinates.

A PE group consists of a forward register, an address calculator unit, a local input buffer, a local data sequencer and PEs, which are denoted as Fwd. register, Address Cal., IN_BUF, D_SEQ and PE_i in Fig. 6, respectively. The forward register receives IFMs and forwards them to the neighbor PE group in order to reduce physical wire delay. The address calculator determines the address of the required IFM pixels from the OFM coordinate assigned to the PE group. The target IFM pixels of one channel are stored in the local input buffer in order to reuse them for the computation of all kernels. Assuming that there are N PEs in one PE group, N consecutive OFM pixels of the same row starting from

the assigned OFM coordinate are computed within a PE group. The local input buffer is register array that stores K rows of N overlapping IFM windows, where K is kernel size and S is the stride. Specifically, it stores input pixels x to $x+(N-1)\times S+K-1$ of row y to $y+K-1$ in total of $K\times((N-1)\times S+K)$ IFM pixels as shown in Fig. 6 (b) when the assigned OFM coordinate is (x, y). The local data sequencer selects data from the local input buffer and passes them to PEs. Each PE is composed of a multiplier, an adder, and an accumulation register. It multiplies the selected data with the broadcasted non-zero weight and accumulates the result with the partial sum result from either the partial sum buffer if the weight is the first one of a kernel or the local accumulation register otherwise. The accumulation result is stored in the accumulation register, denoted as Acc reg in Fig. 6 (a), and it is written to the partial sum buffer after the PE finishes the accumulation of all weights within one channel of each kernel. These operations are pipelined in order to compute MACC in every clock cycle and achieve high frequency.

Figure 7 illustrates data layout in the partial sum buffer. When $P = 1$, all output activations of one OFM in one tile are stored in the same address of the partial sum buffer and the C_o addresses are required. On the other hand, when $P > 1$, all output activations of P OFMs in one tile are stored in the same address of the partial sum buffer and $\frac{C_o}{P}$ addresses are used. Note that a tile is smaller when P grows larger. Partial sum buffer is divided into M banks to store the results from each PE bank. That is because PE banks may perform convolution on different kernels, so they may

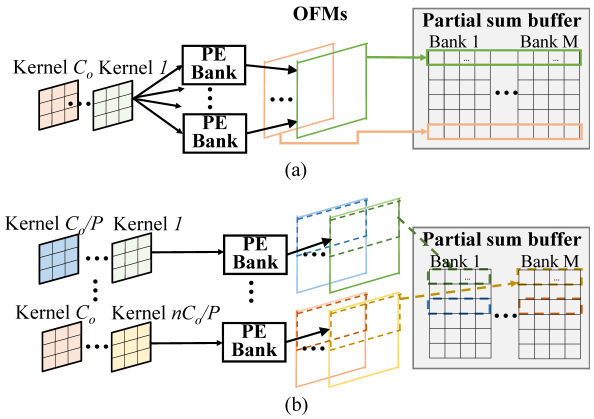


Fig. 7 The data layout in partial sum buffer assuming the number of output activations in an OFM equals to the total number of PEs (a) when $P = 1$, all output activations of one OFM are stored in the same address and C_o addresses are required; (b) when $P > 1$, all output activations of P OFMs in one tile are stored in the same address and $\frac{C_o}{P}$ addresses are required.

access M different addresses at the same time, while PEs within a PE bank do the convolution with the same kernel and they store results to the same address.

To handle irregularity in accessing IFM pixels caused by weight sparsity, the irregular data access is made local in the PE groups. The local data sequencer selects IFM pixels in the local input buffer using the index of each non-zero weight for addressing. Since the local input buffer is register array, the irregular access is simple and fast. The PE computes only the MACCs related with the non-zero weights. The accumulation result of the previous channels is read from the partial sum buffer when the first non-zero weight of a kernel is received. The accumulation result up to the current channel is written to the partial sum buffer after the MACC of the last non-zero weight of a kernel.

Double Buffering To hide the latency of data transfer from external memory, weight memory (Weight mem.), index memory (Index mem.), local input buffer (IN_BUF), and partial sum buffer are implemented with double buffer. **Figure 8** illustrates data load, compute, store timing of the proposed convolution core. First, the compressed CNN model and IFMs of input channel 1 of the first tile are pre-fetched from external memory to weight mem.#0 and IN_BUF#0, respectively. Then, the PE grid performs MACCs on the pre-fetched data and stores the results in partial sum buffer#0. At the same time, the compressed CNN model and IFM pixels of input channel 2 of the first tile are loaded into weight mem.#1 and IN_BUF#1, respectively. While the PE grid is computing the last channel of the tile, the first input channel of IFM of the next tile is loaded into the next available IN_BUF and the first input channel of compressed CNN model is re-loaded to the next available Weight mem. The results of the second tile will be stored in partial sum buffer#1 so that the results of the first tile in partial sum buffer#0 are transferred to the external memory or fed back to the CNN controller as IFMs of the next layer.

3.3.4 Determination of Parallelism in Effect and Degree of Parallelism

The parallelism in effect and degree of parallelism, P, of a layer are determined in advance based on layer specification, the number of BCUs, and the number of PEs in layer-wise as the value that maximizes PE utilization, U. In this context, PE utilization is the percentage between the total number of MACCs of a sparse layer and the total available PE cycles, and is defined as follows:

$$U = \frac{X \times Y \times C_o \times K \times K \times C_i \times R \times 100}{N \times G \times M \times E}, \tag{3}$$

where X, Y, C_o, K, C_i are as in Fig. 1, R is ratio of the number of non-zero weights and the number of all weights, N, G, M refer to

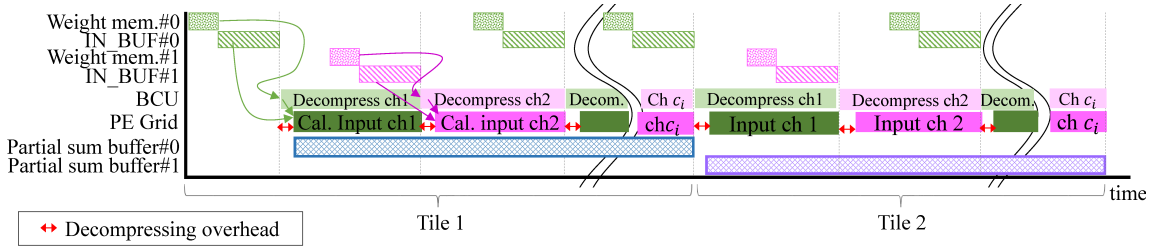


Fig. 8 Timing of data loading, computing, and stroing data of the convolution core using double buffering.

architecture's parameters in Section 3.3.3, and E is the estimated number of cycles in computing a convolutional layer as follows:

$$E = \lceil \frac{C_o \times K \times K \times C_i \times R \times T}{P} \rceil + H \times T \times C_i, \quad (4)$$

$$T = \lceil \lceil \frac{X}{N} \rceil \times \frac{Y}{G \times \lfloor \frac{M}{P} \rfloor} \rceil, \quad (5)$$

The first term of E is the theoretical time for computing the layer with P, and the second term is the total overhead for decompressing and broadcasting the compressed CNN model.

The overhead, H, incurs once for one loop of all IFMs (see Algorithm 1) as illustrated in Fig. 8, and it is constant regardless of layer specification. The existence of the overhead term means that larger P incurs more decompressing overhead even though inter-output parallelism improves multiplier utilization theoretically.

4. Evaluation

To demonstrate the merits of the proposed parallelism-flexible convolution core for sparse CNN, we evaluate performance, resource usage on FPGA and power consumption. This section explains the experimental methodology, and presents the results and the comparison with prior arts of CNN accelerator on FPGA.

4.1 Experimental Methodology

4.1.1 Workload

In the experiment, we measure the performance in computing the convolutional layers of VGG-16 [2]. It is chosen because of three reasons. First, VGG-16 possesses different dominant parallelism within the same network because it includes convolutional layers with various size of IFMs, size of OFMs, number of input channels and number of output channels. For example, the dominant parallelism of the shallow layers, such as conv1_1 or conv1_2, is intra-output parallelism because their size of OFMs is as large as 224×224 pixels. On the contrary, the inter-output parallelism is dominant in the deep layers like conv5_1, conv5_2, and conv5_3 because the number of kernels is larger than the size of OFMs. Second, VGG-16 serves as the backbone of many CNNs, such as SSD [31]. Third, VGG-16 is sparsified by several techniques and its state-of-the-art sparsity is published in Ref. [12].

We generated a sparse VGG model by removing small-valued weights according to the sparsity reported in Ref. [12]. The model was compressed into the compressed CNN model using 16-bit for weight and 4-bit for index. The arithmetic precision is 16-bit for multiplication and 32-bit fixed-point for accumulation.

4.1.2 Architecture Configuration

The proposed convolution core is implemented on Intel's Arria10 GX1150 and Stratix10 GX2800 FPGA with parameters as shown in **Table 1**. A forward register is inserted every one other PE groups in order to save registers. The multipliers of two PEs are mapped onto one DSP since Intel's DSP contains two 18x19-bit multipliers. Specifically, a PE is implemented with one 18x19-bit multiplier, one 32-bit adder, and one 32-bit register for accumulating the results.

In the evaluation, the proposed convolution core execute the convolution layer by layer. The compressed CNN model, IFMs, and OFMs of each layer were transferred between the FPGA and external memory.

Table 1 Parameters of the implemented convolution core.

Parameter	Value
M	16
G	4
N	16
Total PE (multiplier)	1,024

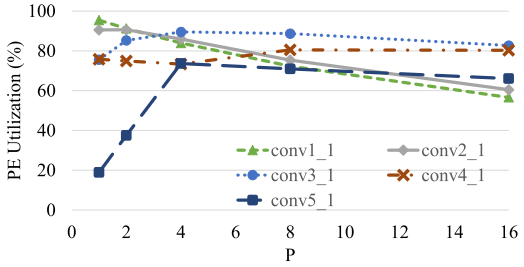


Fig. 9 The estimated PE Utilization when $P = 1, 2, 4, 8$ for conv1_1, conv2_1, conv3_1, conv4_1, and conv5_1 of VGG-16.

Table 2 The parallelism in effect and degree of parallelism for convolutional layers of VGG-16.

Layer	Parallelism in effect	Degree of parallelism (P)
conv1_1 ~ conv1_2	Intra-output parallelism	1
conv2_1 ~ conv2_2	Multi-parallelism	2
conv3_1 ~ conv3_3	Multi-parallelism	4
conv4_1 ~ conv4_3	Multi-parallelism	8
conv5_1 ~ conv5_3	Multi-parallelism	4

The proposed convolution core performs convolution according to the determination of parallelism in effect and degree of parallelism, P, as described in Section 3.3.4. This experiment considers P as 1, 2, 4, 8 or 16. **Figure 9** shows the relationship between P and the estimated PE utilization of VGG-16's conv1_1, conv2_1, conv3_1, conv4_1, and conv5_1 layers. The other layers exhibit similar relationship as the layer computing the same size of OFMs. The overhead of decompressing the compressed CNN model, H, is 16 cycles in our implementation that is designed to achieve high frequency. As a result, increasing P incurs more overhead cycles, hence PE utilization may degrade. For conv1_1, intra-output parallelism occupies all PEs since the size of OFMs is large. The utilization is less than 100% due to the decompressing overhead. For conv2_1, conv3_1, and conv4_1, even though the size of OFMs is large, there exist a small amount of idle PEs when employing only intra-output parallelism. The size of conv5_1's OFMs is very small compared to the number of PEs that only intra-output parallelism is not enough to utilize a large number of PEs. Therefore, employing multi-parallelism by increasing P for conv2_1 through conv5_3 improves multiplier utilization. According to the estimated PE utilization based on Eq. (3), **Table 2** summarizes parallelism in effect and degree of parallelism, P, for exploiting flexible parallelism in the experiments.

4.1.3 Evaluation Method

Performance The execution cycles and giga MACCs per second (GMACS) were measured using RTL simulation.

Resource usage The resource usage was reported from the HDL synthesis results using Quartus Prime software.

Power consumption The resource usage was reported from

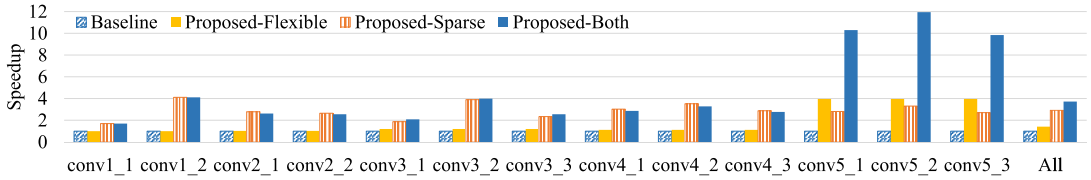


Fig. 10 Speedup of the proposed parallelism-flexible convolution core by layer of VGG-16 compared to the baseline architecture.

the power analysis tool of Quartus Prime software.

4.2 Evaluation Results on VGG-16

The results, i.e., speedup, multiplier utilization, and effective GMACS, show that the proposed convolution core has achieved high performance with a small amount of FPGA resource for controlling flexible parallelism and weight sparsity. The power consumption shows that the proposed convolution core is efficient for edge processing and embedded systems.

4.2.1 Performance

To illustrate the effectiveness of flexible parallelism and weight sparsity, we compared the performance of the proposed convolution core that exploits only flexible parallelism, only weight sparsity, and both techniques with our baseline architecture. The baseline architecture is the architecture that exploits only intra-output parallelism and does not skip zero-operand MACCs.

Speedup The speedup of the proposed convolution core over the baseline architecture in computing VGG-16's convolutional layers is shown in **Fig. 10**. The results of the baseline architecture are denoted by Baseline, and the results of the proposed convolution core that employs only flexible parallelism, only weight sparsity, and both techniques are denoted by Proposed-Flexible, Proposed-Sparse, and Proposed-Both, respectively.

By exploiting flexible parallelism, the performance of Proposed-Flexible achieves 1.42x speedup over the Baseline in the total of all layers. For layer group conv1_x, the Proposed-Flexible does not gain speedup because the intra-output parallelism already occupies all PEs. On the other hand, the intra-output parallelism in layer conv2_1 through conv5_3 leaves some PEs idle. By occupying them with inter-output parallelism, the Proposed-Flexible gains speedup over the Baseline. In layer group conv2_x, conv3_x and conv4_x, there are only a 5%, 23% and 23% of idle PEs in computing as a dense CNN, respectively, so the Proposed-Flexible gains 1.13x speedup in average. Layer group conv5_x takes advantage of the flexible parallelism the most because there are as much as 81% of idle PEs when only intra-output parallelism is exploited. It gains 3.96x speedup compared to the Baseline. Such speedup is achieved because the proposed convolution core can flexibly alternate the dataflow to various degrees of parallelism of multi-parallelism that is the most beneficial for each convolutional layer.

The performance of Proposed-Sparse achieves 2.96x speedup in the total of all layers over the Baseline. By leveraging weight sparsity, it can reduce the execution cycles by the degree of sparsity and gain speedup in every layer. Hence, skipping zero-operand MACC is highly effective in acceleration.

The Proposed-Both achieves 3.73x speedup in the total of all layers since it leverages flexible parallelism and weight sparsity with simple dataflow control. The speedup of layer group conv1_x comes from weight sparsity only, while the speedup of other layers comes from both techniques. In layer group conv5_x, the speedup mainly comes from flexible parallelism. The maximum speedup of 11.95x is achieved in layer conv5_2. However, it is noticeable that the Proposed-Both gains less speedup than the Proposed-Sparse in some layers, such as layer group conv4_x. Furthermore, considering 1.42x and 2.96x speedup in the total of all layers from both techniques, we expected a higher total speedup at 4.17x. Such speedup was not achieved because of two reasons: (1) the imbalance workload of sparse CNN leaves some PEs idle in order to wait for the others to finish their workload of the same channel when inter-output parallelism is leveraged; (2) the decompressing overhead exists and becomes larger when the exploitation of higher P requires data tiling. We discuss these insufficiencies in Section 4.5.

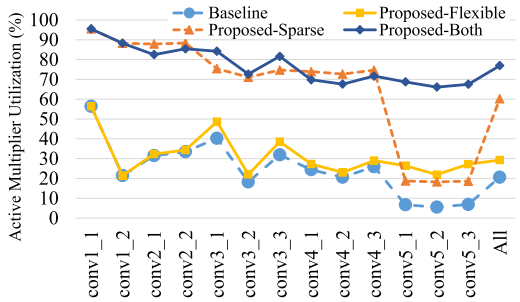


Fig. 11 Active multiplier utilization of the proposed parallelism-flexible convolution core by layer of VGG-16 compared to the baseline architecture.

Active Multiplier utilization To confirm that the proposed convolution core can improve PE utilization in computing sparse CNN, active multiplier utilization, which is defined as the percentage of the number of MACCs incorporated with non-zero weights and the total available multiplier cycles, was examined. Since a PE includes one multiplier and one adder in the proposed convolution core, multiplier utilization and PE utilization is the same in this context.

Figure 11 shows active multiplier utilization, which is calculated as in Eq. (3) with E as the number of execution cycles from the simulation. The active multiplier utilization results of the Baseline and the Proposed-Flexible are quite low in all layers because they compute a sparse CNN in the same way as a dense CNN. In other words, they compute zero-operand MACCs.

Compared to the Baseline, active multiplier utilization of the Proposed-Flexible improves in layer conv2_1 through conv5_3, where multi-parallelism is applied. In layer group conv5_x, the utilization increases by approximately 4x as expected when exploiting multi-parallelism with P equals to 4 and the utilization of the Baseline is under $\frac{100}{P}$%. The improvement is less than the

degree of P in layer conv2_1 to conv4_3 because 77% to 95% of multipliers are occupied considering that they execute as dense CNN, which leaves only a small room for improvement.

The active multiplier utilization rises dramatically when exploiting weight sparsity because all MACCs that take place when using the Proposed-Sparse and the Proposed-Both are meaningful. The active multiplier utilization of the Proposed-Both reaches almost 70% in every layer and as high as 77% in total. It is higher than the Proposed-Sparse except for the layers that achieve lower speedup because of multi-parallelism. However, the multipliers of Proposed-Both are not fully utilized from two causes: (1) architectural fragmentation refers to the fact that PEs are idle because the parameters in Table 1 limit scheduling. There exist two types of fragmentation. First, fine-grained fragmentation refers to the case that some PEs within a PE group are idle when the number of the dimension X of OFM is indivisible by N because the local input buffer limits that all PEs in the same PE group must process the OFM pixels from the same row. Second, medium-grained fragmentation refers to the situation that some PE groups are idle when the number of OFM pixels in one tile is indivisible by $N \times G$ because the BCUs and PE banks are connected one-to-one, so inter-output parallelism cannot be scheduled within the same PE bank to occupy the idle PE groups. We explain further in Section 4.5; (2) imbalance workload as mentioned above.

Required external memory bandwidth The logical bandwidth of the proposed convolution core is 100 Gbps (512 bits data bus operating at 200 MHz), which is achievable in most FPGA boards [32], [33]. The required bandwidth of each VGG-16's convolutional layer is calculated as follows:

$$bandwidth = \frac{total\ bits\ from\ external\ memory}{ideal\ computation\ time}, \tag{6}$$

where $total\ bits\ from\ external\ memory$ includes total bits of compressed CNN model, Bit_{model}, and total bits of IFMs, Bit_{IFM}. The Bit_{model} is calculated as follows:

$$Bit_{model} = \#weight_{nz} \times T \times (Bit_w + Bit_z), \tag{7}$$

where $\#weight_{nz}$ is the number of non-zero weights in a layer, T is number of tiles as in Eq. (5) (because we reload the compressed CNN model for every tile), Bit_w is the number of bits per one weight, which is 16 bits, and Bit_z is the number of bits per one leading-zero value, which is 4 bits. The Bit_{IFM} is calculated as follows:

$$Bit_{IFM} = W \times H \times C_i \times Bit_{data}, \tag{8}$$

where W, H, C_i are as in Fig. 1, and Bit_{data} is the number of data bits, which is 16 bits. The *ideal computation time* is the time for computing MACCs of the sparse CNN with 1,024 PEs at 200 MHz. The result of the calculation shows that the required bandwidth is 29.2 Gbps, which is low compared to the available bandwidth. Therefore, data transfer time can be hidden by double buffering, are hence does not affect the performance of the proposed convolution core.

4.2.2 Resource Usage and Power Consumption

Table 3 shows the Arria10 GX1150 FPGA's resource usage of the implementation of the proposed convolution core with 1,024 PEs that is optimized for VGG-like convolutional layers (kernel size is 3 and stride is 1). The resource usage for the parallelism controller and the weight broadcaster (Parallelism Cntl and Broadcaster in Table 3) is 5, 2, 3, and 3% of LUTs, registers, DSPs, and M20K block RAMs (BRAMs), respectively. It shows that the proposed convolution core can leverage both flexible parallelism and weight sparsity of sparse CNN simply by adjusting the dataflow with a very small resource usage. The maximum frequency of the implementation is 270 MHz.

Table 3 Resource Usage of the implementation of the proposed convolution core with 1,024 PEs optimized for VGG-like convolutional layers on Intel's Arria10 GX1150.

Module	LUTs	Registers	DSPs	M20K
Parallelism Cntl	10,719 (2%)	16,892 (1%)	49 (3%)	0 (0%)
Broadcaster	27,725 (3%)	17,156 (1%)	1 (0%)	104 (3%)
PE Grid	202,309 (24%)	344,416 (20%)	576 (38%)	1,664 (61%)
Core (Total)	240,753 (29%)	378,543 (22%)	626 (41%)	1,768 (64%)

The result shows that BRAMs, which are used up to 65%, are the bottleneck. They are used for storing the compressed CNN model of each layer and the partial sum of OFMs. A large BRAM usage comes from two reasons. First, the design requires a wide bitwidth memory. The weight and index memories for storing the compressed CNN model consume 104 blocks of BRAMs (52 blocks each for each memory). They require wide bitwidth to support the maximum degree of inter-output parallelism according to the number of BCUs. The number of BRAMs can be reduced when the number of BCUs decreases. Likewise, the partial sum buffer, which consumes 1,664 blocks of BRAMs (26 blocks for each PE group), requires wide bitwidth because the bitwidth of the partial sum is as high as 32 bits. Second, we prepared the BRAMs for partial sum buffer for 1,024 PEs in the worst case scenario that P equals to 1 in the 512-kernel layers. In that case, it requires $32 \times 1{,}024$ bits with 512×2 addresses in total to accommodate the data with double buffer. However, flexible parallelism technique employed by the proposed convolution core may reduce the required BRAMs when computing VGG-16. This issue is discussed in Section 4.5.

The power consumption of the proposed convolution core is 25 Watts. It is considered efficient for edge processing platform or embedded systems considering its deliverable performance.

4.3 Comparison with Prior FPGA-based CNN Accelerator

The design quality of the proposed convolution core was compared with prior FPGA-based CNN accelerators. First, we make a fair comparison in terms of performance, i.e. active multiplier utilization and effective GMACS, to demonstrate that the proposed convolution core can increase multiplier utilization, hence, improve the GMACS performance. Then, we make a comparison with the prior arts at their best performance to show the efficiency of the proposed convolution core. Since the definition of PE is different among FPGA-based CNN Accelerators, we refer to the resource for convolution as multiplier in this section. Note that prior accelerators report giga operations per second (GOPS), where one GMACS is equivalent to two GOPS.

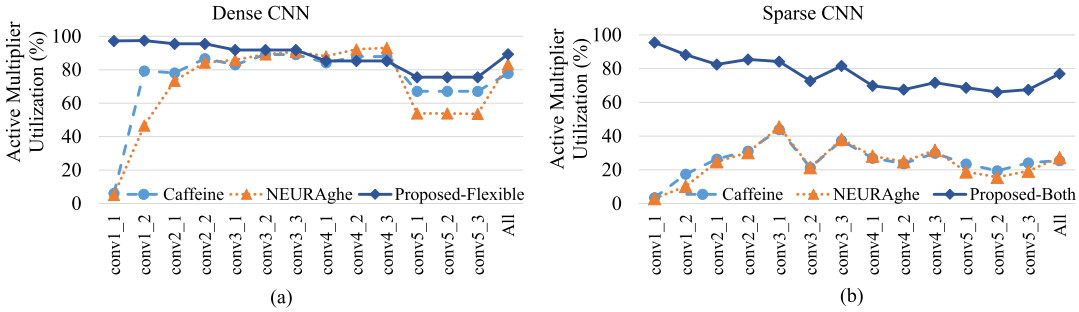


Fig. 12 Active multiplier utilization of Caffeine, NEURAghe, and the proposed parallelism-flexible convolution core by layer of VGG-16 (a) in computing dense CNN; (b) in computing sparse CNN.

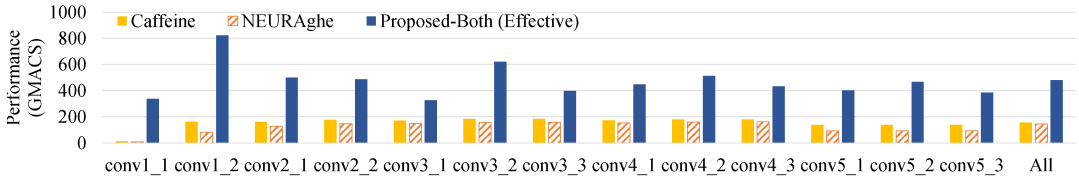


Fig. 13 Performance in GMACS of Caffeine, NEURAghe, and the proposed parallelism-flexible convolution core by layer of VGG-16.

4.3.1 Performance

To make a fair comparison, we selected prior accelerators based on types of parallelism that they exploit and that they provide their results on VGG-16. Their descriptions are as follows:

- **Caffeine** [7] implements inter-output and operation-level parallelism of multiple IFMs with the factor of 32×32 for unrolling the parallelism of OFMs and IFMs in total of 1,024 multipliers. Its operation frequency is 200 MHz on Xilinx's Ultrascale KU060.
- **NEURAghe** [29] implements intra-output and operation-level parallelism. It includes 16 SoP modules, each of which contains 54 multipliers, in total of 864 multipliers. It was not scaled to 1,024 multipliers due to architecture constraints. We scaled their reported performance that operated at 140 MHz to the performance at 200 MHz as follows:

$$GMACS_{200\,\mathrm{MHz}} = \frac{GMACS_{140\,\mathrm{MHz}} * 200}{140}, \tag{9}$$

where $GMACS_{200\,\mathrm{MHz}}$ and $GMACS_{140\,\mathrm{MHz}}$ are GMACS at 200 MHz and 140 MHz, respectively.

Active Multiplier Utilization To show that the proposed convolution core can efficiently utilize multipliers, we show the active multiplier utilization in two aspects: (1) in computing a dense CNN; (2) in computing a sparse CNN. In this context, the active multiplier utilization is not equivalent to PE utilization since the definition of PE varies between the chosen accelerators. For Caffeine and NEURAghe, the active multiplier utilization, $U_{Est.}$, is calculated from the performance in GMACS as follow:

$$U_{Est.} = \frac{GMACS_{200\,\mathrm{MHz}} \times 100}{\#MUL \times f}, \tag{10}$$

where $\#MUL$ and f are the number of multipliers and operating frequency, respectively.

First, **Fig. 12** (a) shows active multiplier utilization of the Caffeine, NEURAghe, and Proposed-Flexible in computing a dense CNN to demonstrate the utilization improvement from flexible parallelism. Note that the utilization of Proposed-Flexible here is different from the previous section because the one in the previous section is the utilization in computing a sparse CNN as a dense CNN, so the number of meaningful MACCs is less than a dense CNN. The figure shows that the Proposed-Flexible utilizes the multipliers better than both Caffeine and NEURAghe in almost all layers and in the total of all layers. That is because the Proposed-Flexible alternates the parallelism in effect and P to use the one that theoretically results in the highest active multiplier utilization. However, the utilization of layer group conv4_x is slightly lower than the Caffeine and NEURAghe due to the effect of decompressing overhead and fine-grained PE fragmentation.

Second, Fig. 12 (b) shows active multiplier utilization in computing a sparse CNN. Our superior active multiplier utilization shows that most multiplier cycles are spent on meaningful MACCs, unlike the architectures that exploit operation-level parallelism and waste time on zero-operand MACCs. Furthermore, the results also imply that flexible parallelism works well with weight sparsity since the utilization of all layers is relatively high.

GMACS In **Fig. 13**, the performance in GMACS is illustrated. The Proposed-Both (Effective) refers to the equivalent effective GMACS that the proposed convolution core can achieve from leveraging weight sparsity. Since the proposed convolution core skips all zero-operand MACCs, it can achieve a superior GMACS compared to other accelerators. The effective GMACS of the proposed convolution core in computing all 13 convolutional layers of VGG-16 is 480.7 GMACS.

4.3.2 Accelerator Comparison

To understand the usability of the proposed convolution core, we make a comparison with prior FPGA-based accelerators according to their reported implementation. In addition to the above-mentioned accelerators, we also compared the proposed convolution core to the accelerators in Ref. [14] and [26] (512-opt-pr variant). While Caffeine [7], NEURAghe [29] and the work in Ref. [14] compute a dense CNN, the work in Ref. [26] and the proposed convolution core can skip MACCs related to zero-valued weights.

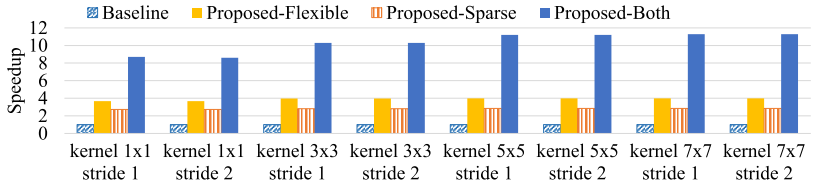


Fig. 14 Speedup of the proposed parallelism-flexible convolution core by kernel size and stride compared to the baseline architecture.

Table 4 Comparison with prior FPGA work.

	[7]	[29]	[14]	[26]	ours
Device	Zynq KU060	Zynq XC7Z045	Zynq XC7Z045	Arria10 SX660	Arria10 GX1150
Frequency	200 MHz	140 MHz	150 MHz	120 MHz	200 MHz
#Multipliers (#DSPs)	1,024 (1,058)	864 (864)	1,152 (780)	- -	1,024 (626)
Power (Watt)	26	10	9.63	-	25
Effective GOPS	310	170	188	53	960
Resource Efficiency	0.31	0.20	0.16	-	0.94
Power Efficiency	12.4*	17.0*	19.50*	-	38.4**

Resource Efficiency is GOPS/Multiplier and Power Efficiency is GOPS/Watt
*The power consumption is measured for the entire system of the CNN accelerator
**The power consumption is measured when there is only the convolution or core on FPGA

Table 4 presents the comparison of the proposed convolution core with prior FPGA-based CNN accelerators. All FPGAs are implemented for 16-bit fixed-point arithmetic precision. #Multipliers refers to the number of logical multipliers for MACCs on the design, which is calculated based on the parameters described in each paper. #DSPs refers to the number of DSPs utilized on each accelerator as reported. Note that a Xilinx's DSP and an Intel's DSP can accommodate one and two 16-bit fixed-point MACCs, respectively. The GOPS is evaluated from 13 convolutional layers of VGG-16.

Compared to other CNN accelerators, the proposed convolution core outperforms them in terms of effective GOPS performance, effective resource efficiency and effective power efficiency. It achieves 3x, 5x, 5x and 18x better performance than the Caffeine, NEURAghe, the work in Ref. [14] and the work in Ref. [26], which leverages sparsity, respectively. In the case of Ref. [26], the low performance despite the fact that it can leverage sparsity is partially due to a relatively low frequency, which might be the results from high-level synthesis. It seems that our high effective GOPS is the result of high frequency. However, when we scaled the NEURAghe as they claim to a larger FPGA, which may bring the frequency up to 200 MHz and double its performance, the proposed convolution core still outperforms in terms of effective GOPS. Similarly, we achieved the highest effective resource efficiency. Our high effective power efficiency implies that the architecture is capable of processing one image with a lower power budget. This means that the proposed convolution core is efficient for being a platform at the edge or on embedded systems.

4.4 Applicability to Modern State-of-the-art CNNs

Based on our survey, the convolutional layer specification of modern state-of-the-art CNNs varies by kernel size and stride in addition to the size of IFMs, the size of OFMs, the number of input channels and the number of output channels. Except for AlexNet [1] which contains kernel size of 11 and stride 4 in the first layer, most modern state-of-the-art CNNs, such as YOLOv2 [34], FCN [35] and ResNet [3], contain convolutional layers with kernel size between 1 to 7 and stride of 1 and 2.

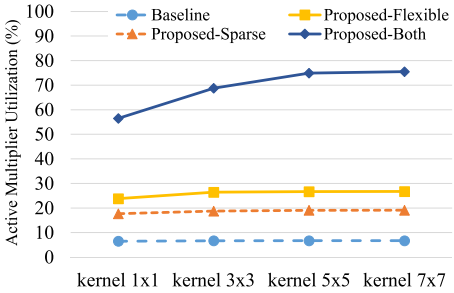


Fig. 15 Active multiplier utilization of the proposed parallelism-flexible convolution core by kernel size compared to the baseline architecture when stride is 1. The active multiplier utilization is the same when stride is 2.

The concept of the proposed convolution core is effective for not only various sizes of IFMs, sizes of OFMs, numbers of input channels and numbers of output channels as shown in the experiment on VGG-16, but it also gains speedup and multiplier utilization despite various kernel sizes and strides. To show that the proposed convolution core can handle a wide range of modern CNNs, the implementation of the proposed convolution core was extended to various kernel sizes and strides by (1) adding logic for model decompression for various kernel size in the broadcaster; (2) enlarging local input buffer of each PE group to accommodate data for kernel size up to 7 and stride up to 2; (3) adding logic for selecting IFM pixels from local input buffer according to the index of sparse weights. The kernel size up to 7 and stride up to 2 were chosen because larger kernel size and stride are rare (no such parameter in YOLOv2, FCN or ResNet) although they can be handled by extending the implementation in a similar manner. In the evaluation of the extended implementation, a sparse model of convolutional layers was generated by zeroing out small values from a randomly generated kernels. The layer specification, i.e. X, Y, C_i, C_o and R, are fixed according to the conv5_1 of VGG-16 because its speedup is achieved from both sparsity and flexible parallelism. Likewise, P is chosen as 4 since it is independent of kernel size and stride.

The speedup and active multiplier utilization are shown in **Fig. 14** and **Fig. 15**, respectively. Despite different strides, the speedup and active multiplier utilization are the same because the total number of MACCs is equal for the same size of OFMs and kernel size. For different kernel sizes, the Proposed-Flexible, Proposed-Sparse, and Proposed-Both achieve similar speedup and active multiplier utilization since performance improvement

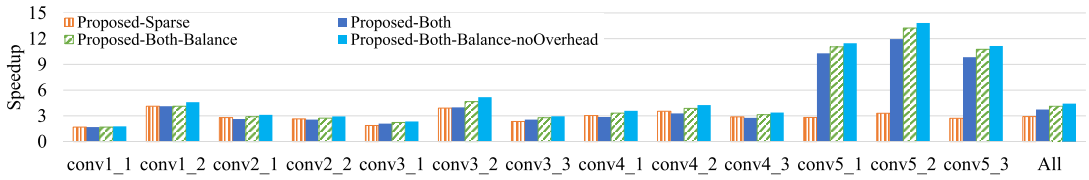


Fig. 16 Speedup of the proposed parallelism-flexible convolution core by layer of VGG-16 in ideal execution scenario.

Table 5 Resource Usage of the extended implementation of the proposed convolution core with 1,024 PEs on Intel's Stratix10 GX2800.

Module	LUTs	Registers	DSPs	M20K
Parallelism Cntl	12,750 (1%)	20,143 (1%)	49 (1%)	0 (0%)
Broadcaster	30,804 (2%)	18,934 (1%)	1 (0%)	104 (1%)
PE Grid	328,430 (18%)	736,013 (20%)	576 (10%)	1,664 (15%)
Core (Total)	374,208 (21%)	774,782 (21%)	626 (11%)	1,768 (16%)

comes from sparsity and P, which are not affected by kernel size. Nevertheless, as the kernel size grows, more speedup and active multiplier utilization are achieved because they suffer less from imbalance workload. The performance improvement in both VGG-16 benchmark and this experiment is achieved by the concept of the parallelism-flexible convolution core, hence, they are not affected by the extension of the implementation.

The synthesis result in **Table 5** shows the required resources. The resources in the PE Grid increase due to the larger local input buffer and IFM pixel selection from the local input buffer. The increased resources in broadcaster and parallelism controller come from accumulating the index of sparse weight during model decompression and assigning OFM coordinates for a larger size of OFMs, respectively. The extended implementation of the proposed convolution core is synthesized for Intel's Stratix10 GX2800 FPGA. We have tried to evaluate the extended implementation on Arria10, however, the required resources exceed the capacity of Arria10 GX1150 FPGA, though the numbers in Table 5 seem likely to accommodate in the capacity of Arria10. We believe this comes from the architectural difference between Arria10 and Stratix10; for instance, Stratix10 has special registers on routing network called HyperFlex. In other words, the proposed architecture is able to exploit the latest feature of state-of-the-art devices. For kernel size of 7 and stride of 2, the size of input buffer increases by 4.8 times compared to when kernel size of 3 and stride of 1. Consequently, the LUTs for selecting IFM pixels from input buffer also increase despite the simple and fast access.

As the state-of-the-art CNNs, such as YOLOv2 and ResNet, have as much as 1k or 2k output channels in a convolutional layer, a large number of output channels can be handled by either increasing the size of output buffer or using $P > 1$. In the extended implementation, we keep the output buffer size as 512 addresses and using $P > 1$ because such large number of output channels usually occurs in deep layers, where the size of OFMs is small that degree of parallelism P is more than 1.

The results have shown that the proposed convolution core is useful for various layer specifications. It is applicable to accelerating the convolutional layers for various state-of-the-art CNNs such as YOLOv2, FCN, ResNet.

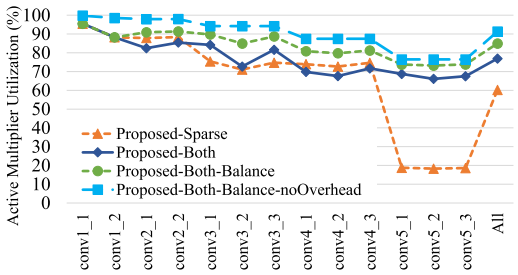


Fig. 17 Active multiplier utilization of the proposed parallelism-flexible convolution core by layer of VGG-16 in ideal execution scenario.

4.5 Discussion

This section analyzes the insufficiencies and bottleneck of our work. Then, it discusses possible solutions and improvement.

4.5.1 Performance

There exist three insufficiencies in the proposed convolution core that prevent it from bringing about its peak performance. First, idle PE cycles arise from the imbalance workload. Second, decompressing the compressed CNN model incurs decompressing overhead. Third, the architectural fragmentation constraints the scheduling of parallelism.

The first insufficiency is that the imbalance workload of sparse kernels increases idle PE cycles when the proposed convolution core exploits inter-output parallelism. That is because the proposed convolution core unrolls the degree-of-parallelism loop in line 3 in Algorithm 1 to implement inter-output parallelism. If the total number of non-zero weights in all kernels (loop in line 5 and 6) that belong to each iteration of line 3 is not equal, PEs are idle in order to wait for PEs in other iterations to finish their workload. To investigate the effect of the imbalance workload, we generated an artificial sparse VGG-like CNN that the workload in every kernel is equal and measure the performance, which is shown as Proposed-Both-Balance in **Fig. 16**. The result shows that the overall performance is improved by 9%. In addition, the Proposed-Both-Balance outperforms the Proposed-Sparse in every layer, which implies that the flexible parallelism can improve the performance of every layer. **Figure 17** illustrates active multiplier utilization, which shows that the Proposed-Both-Balance utilizes PEs better because no PE waits for the others. This problem can be solved in either hardware or software. In hardware, the kernels should be divided into P partitions with an arbitrary number of kernels per partition in such a way that the workload is balanced. However, this may cause complication in storing the results to the partial sum buffer because the partitioning may vary in every input channel. In software, the CNN sparsification process should constraint the number of non-zero weights of each

kernel so that it results in a balanced workload.

Second, the existence of decompressing overhead degrades both performance and active multiplier utilization because the PEs are idle during those cycles. As shown in Fig. 8, the overhead occurs once every input channel as a pipeline latency. This means that more data tiles due to a large P incur more overhead, which degrades the benefit of flexible parallelism. Figure 16 and Fig. 17 show that the speedup and utilization of the ideal execution (Proposed-Both-Balance-noOverhead) improve and the effect of overhead is illustrated with the difference of Proposed-Both-Balance and Proposed-Both-Balance-noOverhead. A 16-cycle decompressing overhead comes from the pipeline for decompressing the compressed CNN model that aims to achieve high frequency. As a consequence, decreasing this overhead may degrade the operating frequency, which results in longer execution time despite the reduced execution cycles.

Third, the proposed convolution core suffers from the architectural fragmentation that prevents PE occupancy during convolution cycles. As mentioned above, there are two types of fragmentation: fine-grained and medium-grained. The example of fine-grained fragmentation is layer group conv4_x, where 28 pixels in one row of OFM leave four idle PEs out of 32 PEs in two PE groups, each of which contains 16 PEs. It adds up to at least 12.5% of all PEs. They cannot be occupied due to the local input buffer limitation. Medium-grained fragmentation occurs in layer group conv5_x, where 14×14 output pixels of one OFM occupy only 196 PEs out of 256 PEs in four PE banks. The effect is as large as 24% of all PEs, which is the main reason for no more than 76% of active multiplier utilization. The architecture is unable to schedule neither inter- nor intra-output parallelism due to the limitation in one-to-one connection to the BCU and dimension of OFM. The effect of this problem can be mitigated by choosing the parameter that is suitable for certain CNN.

4.5.2 Resource Usage

In the implementation optimized for VGG-16, the bottleneck is BRAMs. We designed the partial sum buffer so that it supports the worst case. However, the number of words can be reduced by the factor of P when executing the proposed convolution core with $P > 1$ in the layers that contain a large number of kernels. In other words, the required number of words can be reduced to the maximum of $\frac{C_o}{P}$ across the CNN. Hence, the reduction of BRAM usage allows the FPGA to accommodate more PEs.

5. Conclusion and Future Work

To achieve high performance, the proposed parallelism-flexible convolution core for sparse CNN accelerator exploits multiple types of parallelism flexibly layer by layer to maximize multiplier utilization and skips redundant MACCs due to weight sparsity. The integration of both techniques with parallelism controller and weight broadcaster that is not complicated in terms of dataflow control and resource usage improves performance significantly by 4x speedup over the baseline architecture and 3x in effective GMACS over prior arts of CNN accelerator. To maximally take advantage of the proposed convolution core, the constrained sparsification process remains as our future work.

References

[1] Krizhevsky, A., Sutskever, I. and Hinton, G.E.: ImageNet Classification with Deep Convolutional Neural Networks, *Proc. 25th International Conference on Neural Information Processing Systems - Volume 1*, *NIPS'12*, pp.1097–1105 (2012).

[2] Simonyan, K. and Zisserman, A.: Very Deep Convolutional Networks for Large-Scale Image Recognition, *CoRR*, Vol.abs/1409.1556 (2014).

[3] He, K., Zhang, X., Ren, S. and Sun, J.: Deep Residual Learning for Image Recognition, arXiv preprint arXiv:1512.03385 (2015).

[4] Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K. and Fei-Fei, L.: ImageNet: A Large-Scale Hierarchical Image Database, *CVPR09* (2009).

[5] Chen, Y.H., Krishna, T., Emer, J.S. and Sze, V.: Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks, *IEEE J. Solid-State Circuits*, Vol.52, No.1, pp.127–138 (2017).

[6] Ando, K., Orimo, K., Ueyoshi, K., Ikebe, M., Asai, T. and Motomura, M.: Reconfigurable Processor Array Architecture for Deep Convolutional Neural Networks, *SASIMI2016* (2016).

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

[8] Parashar, A., Rhu, M., Mukkara, A., Puglielli, A., Venkatesan, R., Khailany, B., Emer, J., Keckler, S.W. and Dally, W.J.: SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks, *ISCA '17*, pp.27–40 (2017).

[9] Wu, J., Leng, C., Wang, Y., Hu, Q. and Cheng, J.: Quantized Convolutional Neural Networks for Mobile Devices, *CoRR*, Vol.abs/1512.06473 (2015).

[10] Lin, D.D., Talathi, S.S. and Annapureddy, V.S.: Fixed Point Quantization of Deep Convolutional Networks, *CoRR*, Vol.abs/1511.06393 (2015), available from ⟨http://arxiv.org/abs/1511.06393⟩.

[11] Liu, B., Wang, M., Foroosh, H., Tappen, M. and Penksy, M.: Sparse Convolutional Neural Networks, *2015 IEEE Conference on Computer Vision and Pattern Recognition* (*CVPR*), pp.806–814 (2015).

[12] Han, S., Mao, H. and Dally, W.J.: Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding, *ICLR'16* (2016).

[13] Han, S., Liu, X., Mao, H., Pu, J., Pedram, A., Horowitz, M.A. and Dally, W.J.: EIE: Efficient Inference Engine on Compressed Deep Neural Network, *ISCA '16*, pp.243–254 (2016).

[14] Qiu, J., Wang, J., Yao, S., Guo, K., Li, B., Zhou, E., Yu, J., Tang, T., Xu, N., Song, S., Wang, Y. and Yang, H.: Going Deeper with Embedded FPGA Platform for Convolutional Neural Network, *Proc. 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, *FPGA '16*, pp.26–35, ACM (online), DOI: 10.1145/2847263.2847265 (2016).

[15] Sombatsiri, S., Shibata, S., Kobayashi, Y., Inoue, H., Takenaka, T. and Hosomi, T.: Parallelism-flexible Convolution Core for Sparse Convolutional Neural Networks, *SASIMI2018* (2018).

[16] Sankaradas, M., Jakkula, V., Cadambi, S., Chakradhar, S., Durdanovic, I., Cosatto, E. and Graf, H.P.: A Massively Parallel Coprocessor for Convolutional Neural Networks, *2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors*, pp.53–60 (2009).

[17] Chakradhar, S., Sankaradas, M., Jakkula, V. and Cadambi, S.: A Dynamically Configurable Coprocessor for Convolutional Neural Networks, *SIGARCH Comput. Archit. News*, Vol.38, No.3, pp.247–257 (2010).

[18] Gokhale, V., Jin, J., Dundar, A., Martini, B. and Culurciello, E.: A 240 G-ops/s Mobile Coprocessor for Deep Neural Networks, *2014 IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pp.696–701 (online), DOI: 10.1109/CVPRW.2014.106 (2014).

[19] Cavigelli, L. and Benini, L.: A 803 GOp/s/W Convolutional Network Accelerator, *IEEE Trans. Circuits and Systems for Video Technology* (2016).

[20] Du, Z., Fasthuber, R., Chen, T., Ienne, P., Li, L., Luo, T., Feng, X., Chen, Y. and Temam, O.: ShiDianNao: Shifting vision processing closer to the sensor, *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture* (*ISCA*), pp.92–104 (2015).

[21] Gupta, S., Agrawal, A., Gopalakrishnan, K. and Narayanan, P.: Deep Learning with Limited Numerical Precision, *CoRR*, Vol.abs/1502.02551 (2015).

[22] Peemen, M., Setio, A.A.A., Mesman, B. and Corporaal, H.: Memory-centric accelerator design for Convolutional Neural Networks, *2013 IEEE 31st International Conference on Computer Design* (*ICCD*), pp.13–19 (2013).

[23] Zhang, C., Li, P., Sun, G., Guan, Y., Xiao, B. and Cong, J.: Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks, *Proc. 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, *FPGA '15*, pp.161–170, ACM (2015).

[24] Chen, T., Du, Z., Sun, N., Wang, J., Wu, C., Chen, Y. and Temam, O.: DianNao: A Small-footprint High-throughput Accelerator for Ubiquitous Machine-learning, *SIGPLAN Not.*, Vol.49, No.4, pp.269–284 (2014).
[25] Chen, Y., Luo, T., Liu, S., Zhang, S., He, L., Wang, J., Li, L., Chen, T., Xu, Z., Sun, N. and Temam, O.: DaDianNao: A Machine-Learning Supercomputer, *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp.609–622 (2014).
[26] Kim, J.H., Grady, B., Lian, R., Brothers, J. and Anderson, J.H.: FPGA-based CNN inference accelerator synthesized from multi-threaded C software, *2017 30th IEEE International System-on-Chip Conference* (*SOCC*), pp.268–273 (2017).
[27] Gysel, P., Motamedi, M. and Ghiasi, S.: Hardware-oriented Approximation of Convolutional Neural Networks (2016).
[28] Zhang, S., Du, Z., Zhang, L., Lan, H., Liu, S., Li, L., Guo, Q., Chen, T. and Chen, Y.: Cambricon-X: An accelerator for sparse neural networks, *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture* (*MICRO*), pp.1–12 (2016).
[29] Meloni, P., Capotondi, A., Deriu, G., Brian, M., Conti, F., Rossi, D., Raffo, L. and Benini, L.: NEURAghe: Exploiting CPU-FPGA Synergies for Efficient and Flexible CNN Inference Acceleration on Zynq SoCs, *CoRR*, Vol.abs/1712.00994 (2017), available from ⟨http://arxiv.org/abs/1712.00994⟩.
[30] Lu, W., Yan, G., Li, J., Gong, S., Han, Y. and Li, X.: FlexFlow: A Flexible Dataflow Accelerator Architecture for Convolutional Neural Networks, *HPCA'17*, pp.553–564 (2017).
[31] Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C.-Y. and Berg, A.C.: SSD: Single Shot MultiBox Detector, *Computer Vision – ECCV 2016*, pp.21–37 (2016).
[32] Intel Programmable Acceleration Card with Intel Arria 10 GX FPGA, available from ⟨www.altera.com/products/boards_and_kits/dev-kits/altera/acceleration-card-arria-10-gx.html⟩.
[33] Nallatech 510T Compute Acceleration Card, available from ⟨www.nallatech.com/store/fpga-accelerated-computing/pcie-accelerator-cards/nallatech-510t-fpga-computing-acceleration-card⟩.
[34] Redmon, J. and Farhadi, A.: YOLO9000: Better, Faster, Stronger, *CoRR*, Vol.abs/1612.08242 (2016), available from ⟨http://arxiv.org/abs/1612.08242⟩.
[35] Shelhamer, E., Long, J. and Darrell, T.: Fully Convolutional Networks for Semantic Segmentation, *IEEE Trans. Pattern Analysis and Machine Intelligence*, Vol.39, No.4, pp.640–651 (2017).

Salita Sombatsiri received her B.E. from Chulalongkorn University in 2010 and Master of Information Science and Technology from Osaka University in 2013. She is currently a Ph.D. candidate at Osaka University and working for NEC Corporation, Japan. Her research interests include system-level design, hardware design, deep learning, and biomedical information processing.

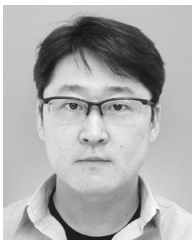
Seiya Shibata received his B.E. degree in information engineering, and his M.S. and Ph.D. degrees in Information Science from Nagoya University, Nagoya, Japan, in 2007, 2009 and 2012, respectively. Currently, he is working for NEC Corporation, Japan. His research interests include system-level design, hardware design, deep learning and video codecs.

Yuki Kobayashi received his B.E. degree from Osaka University, in 2003, then M.S. and Ph.D. degrees in Information Science and Technology from Osaka University, Japan in 2004 and 2007, respectively. From 2005 to 2008, he was a research fellow of the Japan Society for the Promotion of Science. He joined NEC Corporation in 2008. In 2010, he was transferred to Renesas Electronics, Japan, where he was working on a vision processor. In 2015, he moved to NEC Corporation and currently he is a principal researcher in Data Science Research Laboratories. His research interest includes FPGA, design automation, HW/SW co-design, and processor architecture. He is a member of IEEE and IEICE.

Hiroaki Inoue received his B.S., M.E. and Ph.D. degrees from Keio University in 1997, 1999 and 2009, respectively. He joined NEC Corporation in 1999, and is now a senior manager of Data Science Research Laboratories. His current research interests include heterogeneous computing platforms. From 2007 to 2008, he was a visiting scholar of Stanford University. He is a senior member of IEEE and IEICE.

Takashi Takenaka received his M.E. and Ph.D. degrees from Osaka University in 1997 and 2000 respectively. He joined NEC Corporation in 2000 and is currently a senior principle researcher of NEC Corporation. He was a visiting scholar of the University of California, Irvine from 2009 to 2010. His current research interests include system-level design methodology, high-level synthesis, formal verification, and stream processing. He has also served on the organizing and program committees of several premier conferences including ASP-DAC and DAC. He is also a member of IEEE, IEICE and IPSJ.

Takeo Hosomi received his B.S., M.E. degrees from Kyoto University in 1992 and 1994 respectively. He joined NEC Corporation in 1994, and is now a senior expert of Data Science Research Laboratories. His current research interests include heterogeneous computing platforms.



Jaehoon Yu received his B.E. degree in Electrical and Electronic Engineering and his M.S. degree in Communications and Computer Engineering from Kyoto University, Kyoto, Japan, in 2005 and 2007, respectively, and received his Ph.D. degree in Information Systems Engineering from Osaka University, Osaka, Japan, in 2013. He is currently an assistant professor in the Department of Information Systems Engineering, Osaka University. His research interests include computer vision, machine learning, and system level design. He is a member of IEEE and IPSJ.

Yoshinori Takeuchi received his B.E., M.E. and Dr. Eng. degrees from Tokyo Institute of Technology in 1987, 1989 and 1992, respectively. From 1992 through 1996, he was a research associate of Department of Engineering, Tokyo University of Agriculture and Technology. From 1996 through 2017, he was an assistant professor and associate professor with the Osaka University. He was a visiting scholar in University of California, Irvine from 2006 to 2007. From 2018, he is a Professor of Department of Electric and Electronic Engineering at Kindai University. His research interests include System Level Design, VLSI design and VLSI CAD. He is a member of IEICE ACM, and SP, CAS and SSC Society of IEEE.

(Recommended by Associate Editor: *Eiichi Hosoya*)