Microelectronics Journal 147 (2024) 106197

Contents lists available at ScienceDirect

Microelectronics Journal

journal homepage: www.elsevier.com/locate/mejo

Improving the computational efficiency and flexibility of FPGA-based CNN accelerator through loop optimization

Yuhao Liu [a], Yanhua Ma [a,b,*], Bowei Zhang [a], Lu Liu [a], Jie Wang [a], Shibo Tang [a]

[a] School of Integrated Circuits, Dalian University of Technology, Dalian, 116620, China
[b] Key Laboratory of Intelligent Control and Optimization for Industrial Equipment (Dalian University of Technology), Ministry of Education, Dalian, 116023, China

ARTICLE INFO

Keywords:
FPGA circuit design
Architecture
CNN loop optimization
Accelerated computing

ABSTRACT

The convolution operation consists of three-dimensional multiply-accumulate (MAC) operations within four loops, leading to a large design space to be optimized. However, prior research did not thoroughly investigate the loop optimization operations, which led to the development of accelerators that employed inefficient parallel computing architectures and hence consumed unnecessary resources. This study addresses the limitations of existing FPGA-based Convolutional Neural Network (CNN) accelerators in terms of computational efficiency and flexibility by proposing a novel scalable accelerator architecture. We first define a design space that includes loop optimization operations such as loop tiling, loop interchange, and loop unrolling. Based on this, we explore a more efficient dataflow and accelerator architecture through a quantitative analysis of the trade-off between accelerator performance and hardware costs. Then, this paper demonstrates exploring the optimal loop optimization strategy within the design space to guide the design of accelerator architectures, advancing towards the optimal solutions for accelerator performance. The effectiveness of the suggested acceleration architecture is confirmed by implementing VGG-16, ResNet-50, and ResNet-152 on Xilinx ZCU102 and Xilinx ZCU111 FPGAs. The achieved peak throughputs for the networks are 721.48 GOPS, 546.98 GOPS, and 664.66 GOPS, demonstrating outstanding performance, efficient resource usage, and flexibility.

1. Introduction

The FPGA-based convolutional neural network (CNN) accelerator has garnered significant interest in recent years due to its notable energy efficiency and adaptability. It has been effectively utilized in various domains [1–9]. Deploying large-scale convolutional neural networks (CNNs) on FPGA platforms is limited by the availability of computational resources and the energy-intensive nature of off-chip communication, complicating the deployment process further [10,11]. The three loop optimization operations: loop tiling, loop interchange, and loop unrolling can be utilized for the hardware deployment of convolution loops [12,13]. Loop tiling is to divide the entire kernels and feature maps into multiple blocks to fit the capacity of on-chip memory and process sequentially. The design of tiling data requires consideration of factors such as data scale, bandwidth and computational bottlenecks, data transmission costs, and on-chip cache capacity. However, there is currently no existing work that comprehensively addresses all these aspects. Loop interchange determines the sequence of computation for the loops and the order of data transfer. The loop interchange strategies designed in the current advanced CNN accelerator are generally similar. Specifically, most prior works are used to first calculate the inner loops (Loop-1 and Loop-2), followed by the outer loops (Loop-3 and Loop-4) [14–21]. However, there is currently a lack of research that can provide a thorough explanation of how loop order affects computational resources, performance, and other loop optimization methods. Loop unrolling is to design the parallel computation scheme of the convolution loops in hardware. Since the parallelism design of convolution loops directly impacts the dataflow design, data arrangement, and consumption of various hardware resources, it is the most complex and distinctive aspect of the loop optimization designs.

As depicted in Fig. 2, the convolution operation consists of four levels of loops along the convolution kernels and feature maps, resulting in a large design space of loop optimization to explore the parallelism, data partitioning schemes, and loop computation orders. Many techniques have been employed to search the design space for loop optimization strategies, such as automated architecture generation [22,23] and

* Corresponding author. School of Integrated Circuits, Dalian University of Technology, Economy and Technology Development Zone, 211-B, Xinxi Building, Tuqiang Street, Dalian, 116620, China.
E-mail address: mayanhua@dlut.edu.cn (Y. Ma).

https://doi.org/10.1016/j.mejo.2024.106197
Received 6 January 2024; Received in revised form 7 April 2024; Accepted 7 April 2024
Available online 10 April 2024

cross-layer optimization [20,21]. However, these accelerators focused on specific convolutional layers and lacked design considerations for multi-scale convolutional layers. For example, when the convolutional layers with 1 × 1 kernels are processed on the fixed architecture designed for that of 5 × 5 kernel in parallel, it will result in resource underutilization, leading to reduced efficiency and flexibility. In order to overcome this constraint, certain studies have improved the flexibility of CNN accelerators by emphasizing the reuse of data, resulting in impressive performance gains on VGG and ResNets [22,24]. Nevertheless, in these architectures, the entirety of the parallelism space is assigned to the outer loops, leading to notable extra resource utilization. As an illustration, in Ref. [22], the implementation of a ResNet accelerator with 16-bit precision requires the utilization of 3136 parallel MACs, resulting in the consumption of 231.7 K ALMs (Adaptive Logic Modules in Intel FPGAs) and 2136 Intel FPGA BRAMs (about 42 Mb). However, this configuration is not suitable for deployment on edge devices with limited resources.

To address these challenges, this paper presents a scalable accelerator architecture based on Loop optimization. We initially delineated a comprehensive design space for loop optimization, facilitating an in-depth analysis of the impact of operations such as loop tiling, loop interchange, and loop unrolling on the accelerator's performance and hardware costs. Building on this foundation, as illustrated in Fig. 4, this paper demonstrates how to explore this design space to identify efficient loop optimization strategies, thereby guiding the design of the accelerator hardware architecture. A convolution loop optimization strategy is proposed to improve performance and optimize resource usage. This strategy involves unrolling Loop-2, Loop-3, and Loop-4, with Loop-2 and Loop-4 unrolled using multiple parallel MACs, and Loop-3 unrolled using multiple cycles, as explained in Section 5.2. Ultimately, we developed an innovative accelerator architecture as shown in Fig. 1, and dataflow, as shown in Fig. 6, to execute convolutional calculations using the suggested loop optimization strategies.

The effectiveness of the proposed acceleration scheme and architecture was confirmed by implementing VGG-16 [4], ResNet-50, and ResNet-152 [5] on the Xilinx ZCU102 and Xilinx ZCU111 FPGAs. The networks achieved maximum throughputs of 721.48 GOPS, 546.98 GOPS, and 664.66 GOPS, respectively. These results demonstrate exceptional performance, efficient resource utilization, and broad flexibility.

The rest of the paper is organized as follows. Section 2 introduces the essential design variables utilized to quantitatively describe the loop optimization techniques aimed at accelerating the convolution loops. Section 3 presents the related works. Section 4 details the quantitative analysis and estimation of hardware accelerator objectives, leveraging the design variables for loop optimization. Section 5 presents the optimization process of minimizing the design objectives and exhibits the hardware convolution accelerator implementation. Section 6 demonstrates the experimental results. Section 7 is the conclusion.

2. Optimization and acceleration of the convolution loops

2.1. Convolution loops

A typical convolution loop is illustrated in Fig. 2. In order to map convolution operations with four loops onto a hardware platform, loop optimization techniques [24], including loop tiling, loop interchange, and loop unrolling, are employed to organize the convolution loop computations.

2.2. Design variables

We define design variables to represent specific loop optimization strategies, such as loop unrolling, loop tiling, and loop interchange, as referenced in Refs. [12,13]. The design variables for loop unrolling are (P_{kx}, P_{ky}), P_{if}, (P_{ox}, P_{oy}), and P_{of}, which represent the number of parallel computations along each dimension of features or convolutional kernels, respectively. The design variables for loop tiling are (T_{kx}, T_{ky}), T_{if}, (T_{ox}, T_{oy}), and T_{of}, which represent the amount of data in the four loops in the on-chip memory, respectively. It is noticeable that, there exists a constraint about the above-mentioned design variables, which is $1 \leq P^* \leq T^* \leq N^*$, where N^*, T^*, and P^* are the parameters or variables with the prefix of N, T, and P, respectively. T^* represents the number of iterations in the outer loop and P^* represents the size of the parallelism factor. Table 1 lists all the parameters used for optimization.

2.3. Loop optimization techniques

2.3.1. Loop interchange

Loop interchange determines the computation order of the four loops and the required resources for storing partial sums, denoted by M_{sum}. Parameters P_{kx}, P_{ky}, P_{if}, P_{ix}, P_{iy}, and P_{of} are used to represent the

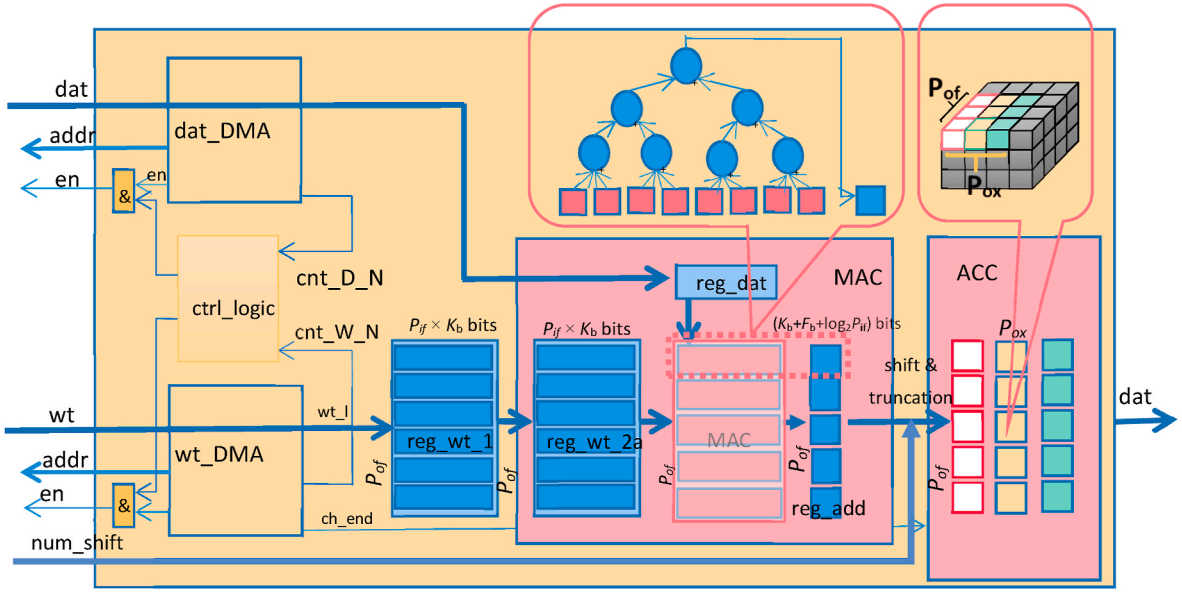

Fig. 1. The schematic diagram of the convolution computation unit.

```
for (no = 0; no < Nof; no ++)                          Loop-4
    for (y = 0; y < Noy; y =+ S)
        for (x = 0; x < Nox; x =+ S)                   } Loop-3
            for (ni = 0; ni < Nif; ni ++)              Loop-2
                for (ky = 0; ky < Nky; ky ++)
                    for (kx = 0; kx < Nkx; kx ++)      } Loop-1
                        pixel(no; x, y) += pixel-1(ni; x + kx, y + ky) × weight-1(ni, no; kx, ky);
    pixelL(no; x, y) = pixel(no; x, y) + bias(no);
```

Fig. 2. Convolution loops. N_{kx}, N_{ky}, N_{if}, N_{ox}, N_{oy}, and N_{of} represent the dimensions of the convolution loop in each respective dimension.

Table 1
Parameters in a convolution layer and hardware device.

Symbol	Meaning
N_{ix}/N_{iy}	Spatial input feature dimensions.
N_{if}/N_{of}	Number of input/output channels.
S_x/S_y	Sliding stride.
N_{ox}/N_{oy}	Spatial output feature dimensions.
$N_{feature}/N_{kernel}$	Amounts of feature and kernel data.
K_b/F_b	Precisions of weight and feature.
M	Number of operations.
BW	Total data transfer volume.
W	Bit width of data transmission.
N_{buffer}	Amount of cache resources.
N_{dsp}	Number of DSPs.
$bits\_BUF\_px\_wt$	Buffer size occupied by pixels and weights.
f_{in}/f_{PE}	Frequencies of reading and processing data.
P^*	Unrolling sizes of convolution loops.
T^*	Tiling sizes of each convolution loops.

unrolling sizes of each loop on the hardware accelerator. When Loop-3 and Loop-4 are executed as inner loops that need to be performed in advance, M_{sum} becomes a function of T^* and P^*, i.e., $M_{sum} = f(T^*, P^*)$, where N^*, T^*, and P^* denote any variable which has a prefix of capital N, T and P, respectively. This implies that the number of registers consumed by the partial sums needs to be consistent with the available on-chip cache resources, which can potentially lead to unpredictable large-scale register consumption and resource mismatch issues. Therefore, performing the computations of Loop-1 and Loop-2 early in the process can effectively manage the number of partial sums, and facilitate the management of data movement when the accelerator processes convolutional layers at different scales.

2.3.2. Loop tiling

Loop tiling represents the data partition scheme for the required data of a convolutional operation in a layer. The input data of a convolution layer can be partitioned into six dimensions, N_{if}, N_{ix}, N_{iy}, N_{of}, N_{ky}, and N_{kx}, to fit into the on-chip buffers according to the tiling scheme to fit the capacity of on-chip memory and processed sequentially. Therefore, the loop tiling parameters are related to the required capacity of the on-chip buffer. The design variables for loop tiling are (T_{kx}, T_{ky}), T_{if}, (T_{ox}, T_{oy}), and T_{of}, which represent the amount of data in the four loops in the on-chip memory, respectively. It is noticeable that there exists a constraint about the above-mentioned design variables, which is $1 \leq P^* \leq T^* \leq N^*$, where N^*, T^*, and P^* are the parameters or variables with the prefixes N, T, and P, respectively. T^* represents the number of iterations in the outer loop and P^* represents the size of the parallelism factor.

2.3.3. Loop unrolling

Loop unrolling determines the parallel computation scheme of the convolution loops in hardware, and directly affects the number of registers and PEs consumed. The design variables for loop unrolling are (P_{kx}, P_{ky}), P_{if}, (P_{ox}, P_{oy}), and P_{of}, which represent the number of parallel computations along each dimension of features or convolutional kernels, respectively.

Loop-1 unrolled. The feature-weight inner products $P_{kx} \times P_{ky}$ are computed per cycle. These inner products require an adder tree with fan-in $P_{kx} \times P_{ky}$ for summation, and an accumulator to add the output of the adder tree to the previous partial sum.

Loop-2 unrolled. Each cycle computes the inner product between P_{if} different channels of features from (x,y) position on the input feature map and the corresponding P_{if} weights. The hardware computation structure for handling the inner product results is similar to unrolling Loop-1, with the fan-in of the adder tree of P_{if}. However, the parallelized multiplication results can be directly accumulated, eliminating the need for a significant amount of resources to store partial sums.

Loop-3 unrolled. Each cycle computes the product between $P_{ix} \times P_{iy}$ features from different (x,y) positions of the same feature map and the same weight. Alternatively, we can say that the weight is reused for $P_{ix} \times P_{iy}$ times. The resulting $P_{ix} \times P_{iy}$ products serve as partial sums without the requirement of adders to sum them up. Instead, it requires additional register resources for caching and accumulation.

Loop-4 unrolled. Each cycle computes the product between a feature from a single (x,y) position in a feature map and multiple weights from different channels. Alternatively, we can say that the feature is reused for P_{of} times. The hardware processing architecture for handling the computation results is similar to unrolling Loop-3.

The unrolling sizes of Loop-3 and Loop-4 determine the number of the sums of the products of P_{if} groups of features and weights computed in parallel in the multiply-accumulate array. Each sum has a data width of $(K_b \times F_b + \log_2(N_{if} \times N_{kx} \times N_{ky}))$ bits. The computed results need to be accumulated in a register array with a size of $P_{ox} \times P_{oy} \times P_{of} \times [K_b \times F_b + \log_2(N_{if} \times N_{kx} \times N_{ky})]$ bits, which is used for storing the partial sums during the computation, where K_b and F_b denote the precisions of weights and features in a layer. An effective optimization strategy must properly control the number of partial sums and process them immediately to minimize data movement. It is observed that data reuse can be achieved by unrolling Loop-3 and Loop-4, but additional resources will be needed to store the partial sums. On the contrary, unrolling Loop-2 allows for savings in partial sum storage resources, but fails to exploit data reuse.

3. Related works

3.1. Unrolling strategies

The loop unrolling strategy directly influences the design of dataflow and computation architecture. In Refs. [14,25], Loop-1 and Loop-3 have been unrolled to realize simultaneous weight and pixel reuse. But Loop-4 was not unrolled, which prevented further pixel reuse. Additionally, since the unrolling size of Loop-1 should be no less than the maximum kernel size in the target convolution layers, it led to severe workload imbalance among the processing elements (PEs) when the kernel size varies, resulting in reduced computational efficiency. Im2col and Row-major [23] have been used to address this issue caused by unrolling Loop-1, but in fact, both of them lead to additional memory or register

resource consumption. Both Loop-3 and Loop-4 were unrolled in Refs. [22,24], which maximized data reuse by allocating all available hardware computation resources to the outer loops. Since the inner loops (Loop-1 and Loop-2) were not unrolled, the parallel computation results accumulated individually on the corresponding partial sums instead of adding up to each other, leading to extremely imbalanced resource consumption. To overcome these limitations, Loop-2 and Loop-4 were unrolled in Refs. [26–29] for combining the inner and outer loops simultaneously. However, since Loop-3 was not simultaneously unrolled, it lacked weight reuse for convolution layers.

3.2. Tiling strategies

Among the adopted loop tiling strategies, it is often the case that if the tiling size does not cover Loop-1 and Loop-2, excellent performance is not achieved. For example, there are cases in $T_{kx} = T_{ky} = T_{if} = 1$ in Ref. [30], and $T_{if} < N_{if}$ in Ref. [31]. In these cases, data used in Loop-1 and Loop-2 was incapable of being fully buffered, which further led to a significant increase in the number of partial sums and data movements. To address this problem [24], tiled Loop-3 and Loop-4. Then the data required for computing an output feature within Loop-1 and Loop-2 was fully buffered to benefit the computation. However, due to the lack of comprehensive quantitative analysis and exploration of suitable loop tiling factors [24], considered the tiling factor, which minimized DMA transfers, as the optimal choice. However, it failed to ensure that the data processing speed of the accelerator should be faster than the data caching speed.

3.3. Interchange strategies

The computation order of loops significantly affects the number of partial sums and the resulting data movement and memory access. However, the quantitative relationship between these factors has not been studied extensively in the previous works. Even though some works have proposed specific loop interchange schemes, they hardly provided a detailed argumentation [24].

Therefore, to optimize convolution loops, we aim to build upon prior works by designing more effective loop optimization strategies, systematically exploring the size of loop tiling, and analyzing the impact of loop order on the design. Although extensive research has been carried out on loop optimization strategies, as far as we know, no previous research has explored a comprehensive design space to construct a convolutional architecture that is applicable for improving the computational efficiency and resource utilization of convolutional layers with different sizes.

4. Optimizing design objectives for CNN accelerator

In this section, we first conduct a quantitative analysis to understand how these loop optimization operations, based on the defined design variables, impact the following design objectives of the CNN accelerator:

1) *Minimizing computing latency* by the loop unrolling strategy.
2) *Minimizing the requirement for storing partial sums* by the loop interchange strategy. An earlier derivation of the final pixel output results in fewer partial sums needing storage.
3) *Minimizing the frequency of external memory accesss* by the loop tiling strategy.

Upon establishing a comprehensive cost model, we are able to search for efficient design variable configurations within this design space to optimize design objectives.

4.1. Computing latency

Computing latency is mainly determined by the loop unrolling strategy. Loop-1 is the only loop that we have not unrolled. First, the unrolling size should be no less than the maximum kernel size, which will always generate idle computing units when the kernel size varies. Second, the data required for one computation may be scattered in memory and thus require multiple transfers. Different layout schemes such as Im2col and Row-major [23] have been used to address this issue, but in fact, both of them lead to additional memory or register resource consumption. Relatively speaking, it is easy to identify and adjust the unrolling sizes of Loop-2 and Loop-4 which help to adapt to the target platform according to the requirements. For example, in ResNets [4], and VGG [5] while the kernel sizes, N_{kx} and N_{ky}, are 1,3,5,7 the feature maps channels N_{if} and N_{of} can take values from (64,128,256,512,1024, 2048).

Unrolling Loop-3 and Loop-4 are the keys to promoting data reuse in convolution loops. The unrolling sizes of the two outer loops determine the number of the sums of the products of P_{if} groups of features and weights computed in parallel in the multiply-accumulate array. Each sum has a data width of $(K_{\mathrm{b}} \times F_{\mathrm{b}} + \log_2 P_{\mathrm{if}})$ bits. The computed results need to be accumulated in a register array with a size of $P_{\mathrm{ox}} \times P_{\mathrm{oy}} \times P_{\mathrm{of}} \times [K_{\mathrm{b}} \times \mathrm{F_b} + \log_2(N_{\mathrm{if}} \times N_{\mathrm{kx}} \times N_{\mathrm{ky}})]$ bits, which is used for storing the partial sums during the computation. An effective acceleration strategy must properly control the number of partial sums and process them immediately to minimize data movement. It is observed that data reuse can be achieved by unrolling Loop-3 and Loop-4, but additional resources will be needed to store the partial sums. On the contrary, unrolling Loop-2 allows for savings in partial sum storage resources, but fails to exploit data reuse.

4.2. Partial sums storage

The requirement for storing partial sums, denoted by M_{sum}, and the data requirements per cycle, denoted by B_{data}, are determined by the loop interchange strategy. This strategy involves the computation order of Loop-1, Loop-2, and Loop-3, Loop-4.

If Loop-1 and Loop-2 are the innermost loops, then the weights and features are updated in each cycle. There is:

$$M_{\mathrm{sum}} = P_{of} \times P_{ox} \times P_{oy} \times \left[K_{\mathrm{b}} + F_{\mathrm{b}} + \log_2\left(N_{if} \times N_{kx} \times N_{ky}\right)\right] \quad (1)$$

If Loop-3 is the innermost loop where the multiplication array primarily updates the features, then the weights can be continuously reused for $T_{\mathrm{ox}} \times T_{\mathrm{oy}}/(P_{\mathrm{ox}} \times P_{\mathrm{oy}})$ cycles. There is :

$$M_{\mathrm{sum}} = P_{of} \times T_{ox} \times T_{oy} \times \left[K_{\mathrm{b}} + F_{\mathrm{b}} + \log_2\left(N_{if} \times N_{kx} \times N_{ky}\right)\right] \quad (2)$$

If Loop-4 is the innermost loop where the multiplication array primarily updates the weights, then the features can be reused for $T_{\mathrm{of}}/P_{\mathrm{of}}$ loop cycles. There is:

$$M_{\mathrm{sum}} = T_{of} \times P_{ox} \times P_{oy} \times \left[K_{\mathrm{b}} + F_{\mathrm{b}} + \log_2\left(N_{if} \times N_{kx} \times N_{ky}\right)\right] \quad (3)$$

It is observed that when Loop-3 and Loop-4 are executed as inner loops that need to be performed in advance, M_{sum} becomes a function of T^* and P^*, i.e., $M_{\mathrm{sum}} = f(T^*, P^*)$. This implies that the number of registers consumed by the partial sums needs to be consistent with the available on-chip cache resources, which can potentially lead to unpredictable large-scale register consumption and resource mismatch issues.

4.3. External memory access

The number of external memory access primarily relies on the loop tiling strategy. The input data of a convolution layer can be partitioned into six dimensions, N_{if}, N_{ix}, N_{iy}, N_{of}, N_{ky}, and N_{kx}, to fit into the on-chip buffers. Fig. 3 a) illustrates the process of tiling Loop-2 (i.e., $T_{\mathrm{if}} < N_{\mathrm{if}}$). Each output pixel represents a partial sum of the convolution operation, which necessitates a large number of off-chip communications to accumulate and obtain the complete output pixels. The same principle

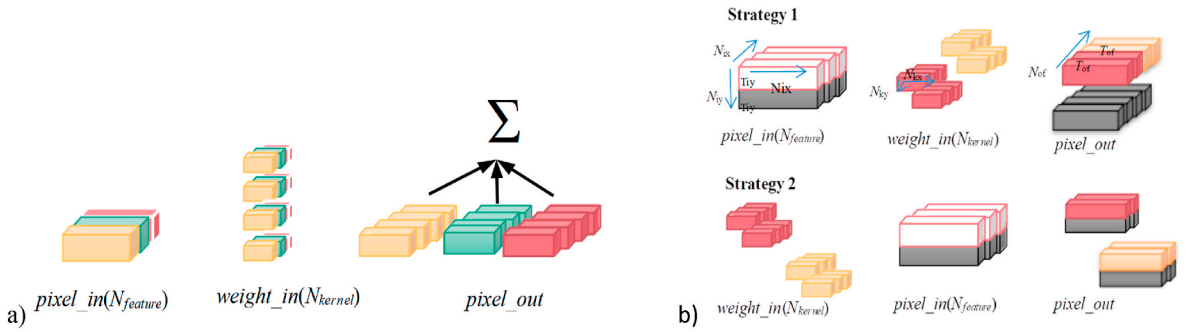

Fig. 3. Comparison of Two Loop tiling schemes. a)Tiling Loop-2 (i.e., $T_{if} < N_{if}$), where the input pixels and weights are divided and ordered along the input channels. b)Tiling Loop-3 and Loop-4, i.e., $T_{iy} < N_{iy}$ (or $T_{ix} < N_{ix}$) and $T_{of} < N_{of}$, involves dividing the $N_{ix} \times N_{iy}$ input feature map and Nof convolutional filters into multiple parts and computing them one by one.

applies to loop tiling for Loop-1 as well. Therefore, it is desirable to ensure that $T_{kx} = N_{kx}$, $T_{ky} = N_{ky}$, and $T_{if} = N_{if}$ to minimize the additional transfer overhead caused by computing partial sums and the extra external memory consumption. As illustrated in Fig. 3 b), with the strategies of tiling Loop-3 and Loop-4, each input pixel block provided for computation meets the minimum size requirements in the length, width, and height dimensions, we can ensure that the resulting output feature maps are complete instead of partial sums when the convolutional kernel slides over each input pixel block.

Therefore, the Loop-3 and Loop-4 should be tiled simultaneously. It can be exerted in two different ways, as shown in Fig. 3 b). As for Strategy 1, the weight groups are sequentially read for each segmented input pixel block. As for Strategy 2, the segmented input pixel blocks are sequentially loaded into on-chip buffers for each weight group. The total amounts of data transferred for the two strategies can be calculated as follows:

$$BW_{Strategy1} = \left[N_{feature} + \frac{N_{feature}}{N_{iy}} \times (N_{ky} - S_y) \times \left(\frac{N_{iy}}{T_{iy}} - 1\right)\right] + \frac{N_{iy}}{T_{iy}} \times N_{ketnel} \quad (4)$$

Strategy 2 requires reading the weights once and the input pixels N_{iy}/T_{iy} times.

$$BW_{Strategy2} = N_{ketnel} + \frac{N_{of}}{T_{of}} \times \left[N_{feature} + \frac{N_{feature}}{N_{iy}} \times (N_{ky} - S_y) \times \left(\frac{N_{iy}}{T_{iy}} - 1\right)\right] \quad (5)$$

Under the constraint of $N_{buffer} > bits\_BUF\_px\_wt$, we can determine the allowable ranges of the tiling factors T_{oy} and T_{of}, where N_{buffer} is the number of buffers available on the platform, and $bits\_BUF\_px\_wt = f(T_{oy}, T_{of})$ represents the buffer size jointly occupied by pixels and weights. As demonstrated in the for loop in Fig. 4, we divide the available input buffer space in our platform, denoted as N_{buffer}, into multiple banks of equal size. These banks are dynamically allocated to store pixels and weights in various ratios, recognizing that different allocation ratios and tiling strategies can significantly impact the required number of DRAM access, where *wt_buff_bank_minimum* and *pixel_buff_bank_minimum* denote the minimum numbers of buffer banks required to be allocated for weights and pixels, respectively, to enable the initiation of standard computational processes. By iterating within these allocation methods, we calculate the resulting data transfer volume pairs, labeled ($BW_{Strategy1}$, $BW_{Strategy2}$), for each ratio under two exemplary tiling strategies. The strategy yielding the minimum transfer volume is then adopted as our preferred approach.

5. Proposed convolution accelerator

In this section, we present the optimization process for our acceleration scheme, guided by the design objectives and analysis discussed in Section 4. Then, a CNN hardware accelerator is designed to implement the proposed loop optimization strategies with an efficient dataflow and PE architecture.

5.1. Proposed loop optimization strategies

Based on the work provided above, we perform loop optimization operations according to the process shown in Fig. 4 to optimize design objectives.

5.1.1. Minimizing computing latency

To minimize computing latency, we employ multiple parallel MAC operations to unroll Loop-2 and Loop-4, while Loop-3 is unrolled across multiple cycles for data reuse and reduction in data transfers. Specifically, the outer loops (Loop-3 and Loop-4) facilitate data reuse, and the inner loop (Loop-2) helps balance logic and BRAM resources. To prevent excessive BRAM resource consumption by the outer loops, Loop-2 and Loop-4 are spatially unrolled, whereas Loop-3 is temporally unrolled, allowing parallel computations in Loop-2 and Loop-4 to complete in a single cycle and those in Loop-3 over multiple cycles. This strategy, coordinated with the data layout design, is further elaborated in Section 5.2. The loop unrolling sizes of the four convolution loops collectively determine the number of required multipliers. Therefore, the product of the unrolling sizes for each loop should not exceed the number of available multipliers in the device (assuming all multiplications are implemented by DSPs), denoted as N_{dsp}, as shown in Equation (6).

$$P_{ix} \times P_{iy} \times P_{if} \times P_{of} < N_{dsp} \quad (6)$$

Taking the implementation of VGG-16 on the ZCU111 FPGA as an example, since only Loop-2 and Loop-4 are spatially unrolled, we adopt design variables $P_{ix} = P_{iy} = P_{kx} = P_{ky} = 1$. Assuming that the number of multipliers that can be realized through the available DSP blocks is 4,272, we target the actual number of parallel multiplications to be around 4000. Consequently, we configure $P_{if} = 64$ and $P_{of} = 64$, to fully utilize the available computing resources and minimize the latency.

5.1.2. Minimizing partial sums storage

To minimize the storage of partial sums, we execute the convolution computation sequentially from Loop-1 through Loop-4. This execution order allows us to perform the computations for Loop-1 and Loop-2 early in the process, enabling efficient management of the number of partial sums. It also aids in optimizing data movement management as the accelerator processes convolutional layers of varying scales. Subject to the constraints $1 \le P^* \le T^* \le N^*$, Equation (1) enables the attainment of the minimum required storage for partial sums.

5.1.3. Minimizing external memory access

To minimize the external memory access, we tile the data on the off-chip memory from dimensions Loop-3 and Loop-4 and sequentially transfer them into the on-chip memory. To ensure that the computational units are fully utilized, the parallel computations for each

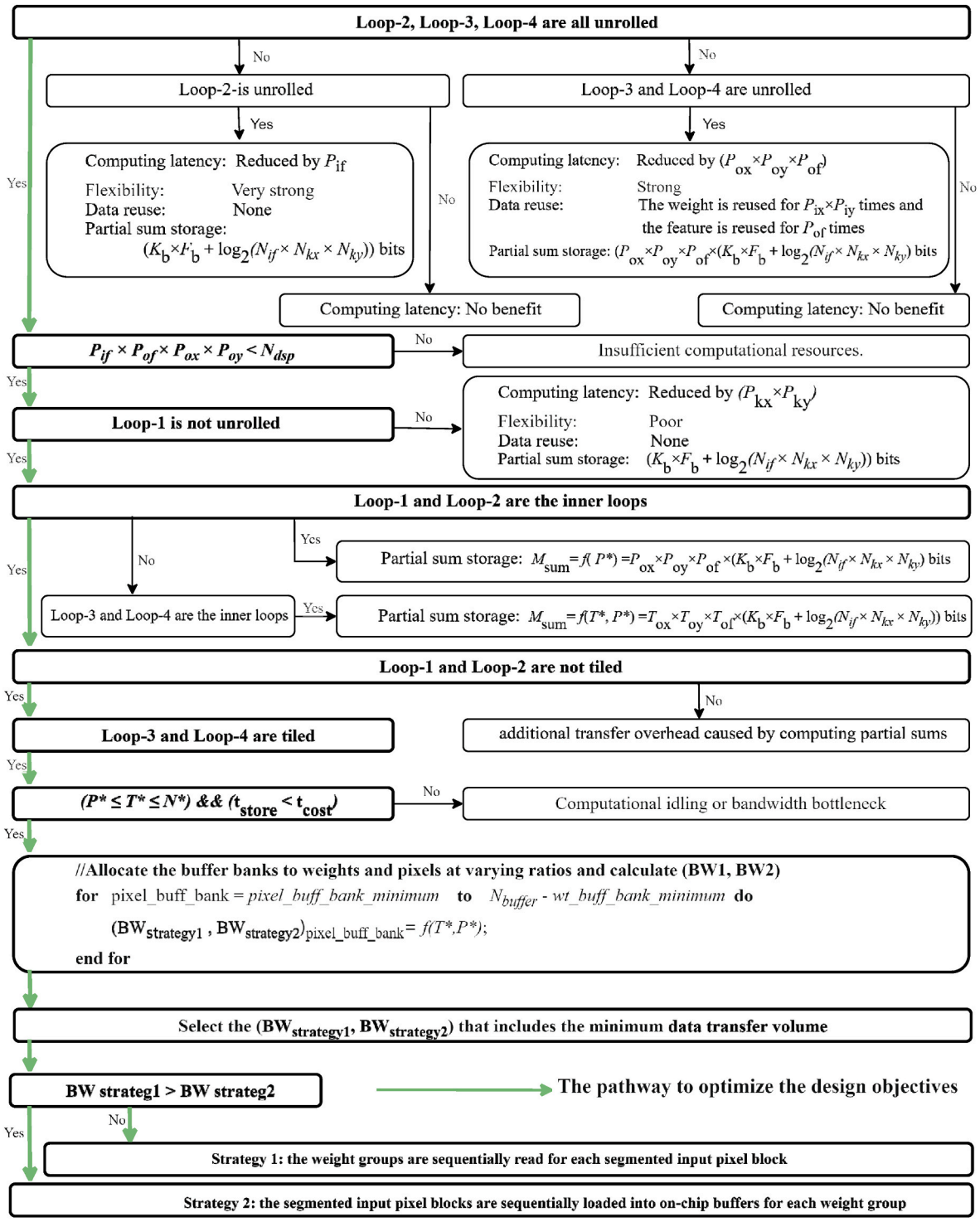

Fig. 4. A flowchart that demonstrates how to optimize design objectives within the loop optimization space. The tradeoff determination steps are highlighted.

dimension should be chosen as a common factor of all possible values for that dimension in the layers being processed. For the convolutional layers L, the design variable T^* should be satisfied that $T_L^* \,\%\, P_L^* = 0$. To prevent transfer bottlenecks, the choice of design variable T_L^* must also satisfy the transfer speed constraint $t_{store\_L} < t_{cost\_L}$, where t_{store_L} denotes the time it takes for feature tiles to be transferred to the on-chip cache in layer L, and t_{cost_L} denotes the time taken for these data to be consumed. This is expressed as

$$t_{store\_L} = \left(\frac{N_{ix}}{T_{ix}} \times \frac{N_{iy}}{T_{iy}} \times \frac{N_{if}}{T_{if}} \times F_b + \frac{N_{kx}}{T_{kx}} \times \frac{N_{ky}}{T_{ky}} \times \frac{N_{kf}}{T_{kf}} \times \frac{N_{of}}{T_{of}} \times K_b\right) \Big/ \left(W \times f_{in}\right) \tag{7}$$

$$t_{\cos t\_L} = \frac{\left[(N_{ix} - N_{kx} + 1) \times \left(T_{iy} - N_{ky} + 1\right) \times N_{kx} \times N_{ky}\right] \times N_{if} \times N_{of}}{\left(P_{kx} \times P_{ky} \times P_{ox} \times P_{oy} \times P_{if} \times P_{of}\right) \times f_{PE}} \tag{8}$$

Then, the formulation for the optimization of DRAM access minimization is presented as follows:

$$\begin{aligned} &\text{minimize} \quad DRAM\_access \\ &\text{subject to} \quad 1 \le P_L* \le T_L* \le N_L*, \\ &\qquad t_{store\_L} < t_{\cos t\_L} \\ &\text{with} \quad \forall L \in [1, \#convs], \end{aligned} \tag{9}$$

where *#CONVs* is the number of convolution layers. By solving Equation (9) as discussed in Section 4.3, an optimal configuration of T^* variables can be identified, leading to the minimization of DRAM access and on-chip buffer size. Nevertheless, given the constraints set on $T_{kx} = N_{kx}$, $T_{ky} = N_{ky}$, $T_{if} = N_{if}$ as detailed in Section 4.3, we can only attain a sub-optimal solution by tuning T_{ox}, T_{oy}, and T_{of}, which entails a larger buffer size requirement.

Taking the VGG-16 implementation on the ZCU102 FPGA as a case study, we aimed to simplify computations to meet real-time requirements. On the ZCU102, the available 16.4 Mbits of BRAM for input caching was divided into 16 banks of equal size, with $T_{ox} = N_{ox}$ to facilitate exploration of T_{oy} and T_{of} configurations. Given the number of convolutional layers (*#CONVs*), (T_{oy}, T_{of}) variable pairs could present up to 15 candidate values, bounded by the constraints $t_{store\_L} < t_{cost\_L}$ and $1 \le P^* \le T^* \le N^*$. Consequently, the total number of (T_{oy}, T_{of}) configurations was determined to be 15 × 12. These could be efficiently calculated in real-time using a Python script executed on a CPU. The optimization process yielded a minimum DRAM access requirement of 0.25 GBytes, with a data width of 16 bits for both pixels and weights.

5.2. Proposed dataflow and data layout

Fig. 5 illustrates the data layout for storing on the on-chip memories. These data are segmented into multiple blocks and are read from off-chip memory to on-chip memory in accordance with the encoding order. Each encoded block represents a group of $1 \times 1 \times P_{if}$ data, which can be obtained through a single bus transfer.

In order to avoid generating a large number of partial sums, priority is given to computing Loop-1 and Loop-2 first, without partitioning Loop-1 and Loop-2. To achieve this, the read order of the feature maps and kernels from on-chip memory to the PEs are customized for data access continuity as shown in Fig. 6. Each encoded block represents a group of $1 \times 1 \times P_{if}$ data. Over consecutive P_{ix} cycles, adjacent P_{ix} blocks are sequentially loaded into *reg_dat* shown in Fig. 7. During this time, P_{of} weight blocks are stored in *reg_wt_2* at the same positions within P_{of} kernels. This way, $P_{if} \times P_{of} \times P_{ix}$ multiplication operations are achieved within P_{ix} cycles. Through this dataflow, Loop-2 and Loop-4 are unrolled with multiple parallel MACs, and Loop-3 is unrolled with multiple cycles.

5.3. Proposed CNN accelerator

As shown in Fig. 7 the proposed CNN architecture in this paper is deployed on a heterogeneous CPU + FPGA platform, utilizing a software-hardware co-design approach. The CPU controls the data interaction between the SDRAM and the accelerator and monitors the hardware computation status and cache consumption in real-time. The accelerator system is implemented on the FPGA side and is responsible for data computation. The main modules implemented in the accelerator include DMA, on-chip buffer, convolution unit, batch normalization unit, element-wise unit, non-linear unit, and pooling unit. After convolution processing, the input data can be chosen to either proceed with further operations or be bypassed for output. The convolution module we designed unrolls input channels and output channels, making it well-suited for fully connected operations. As a result, the fully connected (FC) layer is also implemented using the convolution module.

5.3.1. Convolution unit

The design of the convolution computation unit is divided into four parts based on the functionality, as shown in Fig. 1. Firstly, the *dat_DMA* and *wt_DMA* are used to control the data access from the on-chip cache to the PEs. Prior to the weight data being passed to the PEs, two sets of register banks, *reg_wt_1*, and *reg_wt_2*, with a size of $P_{of} \times P_{if} \times K_b$ bits, are used to cache the weights, implementing a ping-pong working mode. After caching P_{of} sets of weights in *reg_wt_1* (each set containing P_{if} weights), the weights are passed to the *MAC* array for weight reuse calculation through *reg_wt_2*. When *reg_wt_2* caches one of P_{of} sets of weights from *reg_wt_1*, we feed P_{ox} sets of $P_{if} \times F_b$ bits pixel data to the computation unit within the P_{ox} cycles, achieving an unrolling scale of $P_{of} \times P_{ox} \times P_{if}$. Additionally, a control logic module, *ctrl_logic*, is implemented within the computation unit to coordinate the reading of features and weights.

The *MAC* unit performs multiplication operations on the weights of dimension (P_{of},P_{if}) and the pixel data of dimension (P_{if},1) in each cycle. The partial products are accumulated using an adder tree, resulting in a (P_{of},1)-dimensional vector, which represents the partial sum of P_{of} channels at one output pixel point. To implement the MAC unit, $P_{of} \times P_{if}$ multipliers are required, along with an adder tree with a fan-in of $P_{kx} \times P_{ky}$, which performs multiplication and summation operations on $P_{kx} \times P_{ky}$ groups of weights and features. The multiplication-accumulation results are stored in a register bank, *reg_add*, with a size of $P_{of} \times$

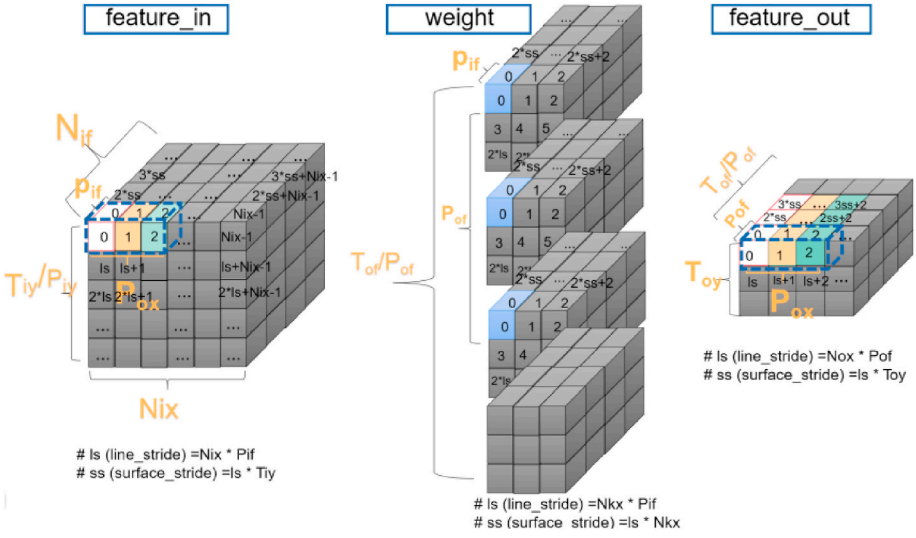

Fig. 5. The data layout on the on-chip and off-chip memories. "*feature_in*" and "*weight*" represent the tiled features and weights stored in the on-chip memory, respectively, which are the results of partitioning the original features and weights along Loop-3 and Loop-4.

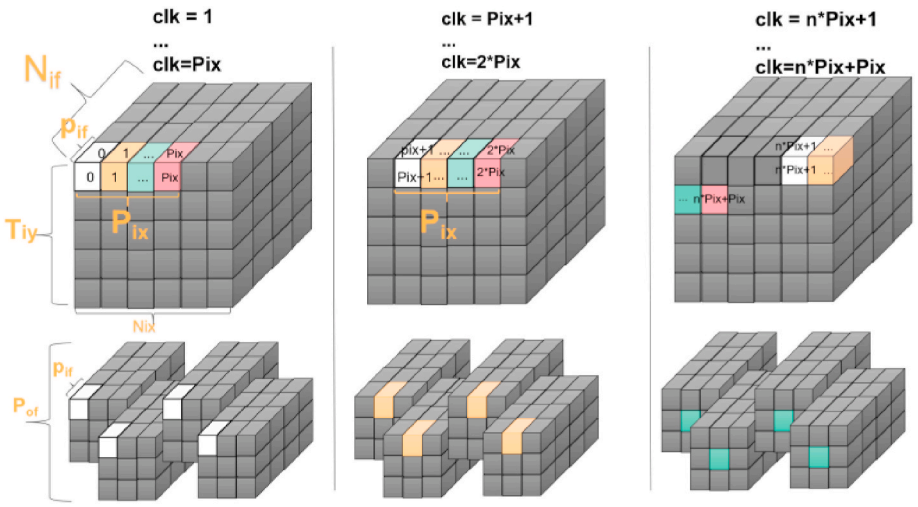

Fig. 6. The customized read order of feature and weight data from on-chip memory to PEs. Each encoded block represents a group of $1 \times 1 \times P_{if}$ data.

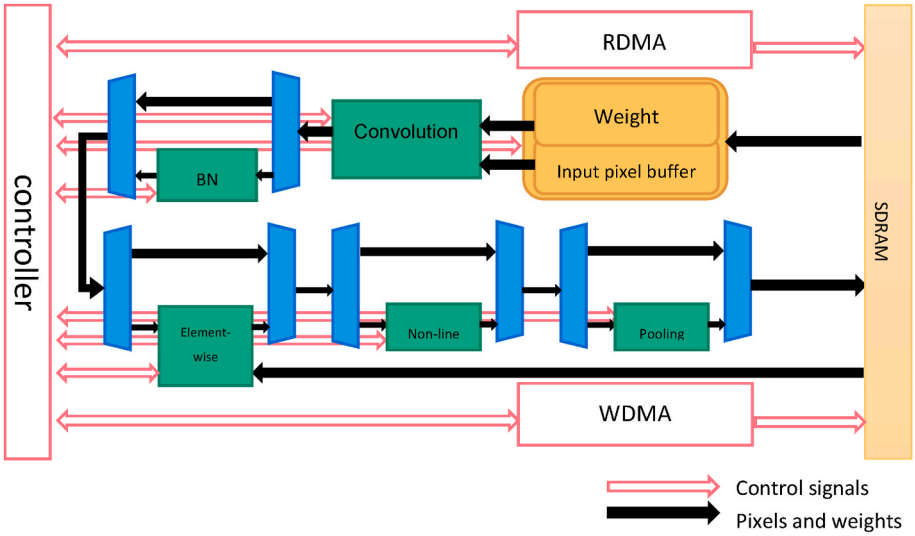

Fig. 7. Top-level CNN acceleration system.

$(\log_2 P_{if} + K_b + F_b)$ bits.

The Accumulator (*ACC*) unit performs the accumulation and buffering of the partial sums computed by the *MAC* unit. The completion of accumulation is determined by the completion signal provided by the *wt_DMA* state machine. Since the computation unit operates in a ping-pong mode, with the unrolling of Loop-3 and Loop-4, the partial sums for Loop-3 are calculated in P_{ox} cycles, while the partial sums for Loop-4 are calculated simultaneously within a single cycle. Therefore, the *ACC* unit requires P_{of} accumulators to accumulate the outputs of the multipliers, as well as a register buffer of size $P_{ox} \times P_{of} \times (K_b + F_b + \log_2(N_{kx} \times N_{ky} \times N_{if}))$ bits to store the partial sums. Once the *ACC* unit completes the accumulation for an output pixel, the output pixel is arithmetically shifted and truncated by *num_shift* bits towards the lower bits. The resulting accumulated value is then output as an F_b bit value.

5.3.2. Element-wise operation unit

The Element-wise layer performs element-wise addition to connect two branches of layers in ResNet CNNs. To achieve this, the element-wise unit stores the computation results from the previous layer while simultaneously reading pixels from another branch in SDRAM. Subsequently, the pixels from the two branches are element-wise added by the adders and finally stored back into the output pixel buffers.

5.3.3. Pooling unit

In the pooling unit, two modules, *Pool_kx* and *Pool_ky*, respectively, handle the horizontal and vertical pooling operations as depicted in Fig. 8. Each time the data bus carries data from *Pof* output channels at a pixel location on the feature map, *Pool_kx* module performs width-wise pooling calculations on the input feature map. Registers within *Pool_kx* temporarily store data from the first data bus transmission and subsequently apply pooling logic (maximum, minimum, or average) to this data in combination with data received in each subsequent data bus transmission. After the pooling unit has processed data from N_{kx} transmissions, the result, with dimensions $N_{ky} \times P_{of}$, is passed to *Pool_ky* for *Nky*-direction pooling.

6. Experiment results

6.1. Experiment setup

The proposed CNN accelerator is demonstrated by accelerating the inference process of both conventional CNNs, e.g. VGG-16 [4], and complex DAG form CNNs, e.g. ResNet-50 and ResNet-152 [5] on the Xilinx ZCU102 and Xilinx ZCU111 platform. The ZCU102 platform features the XCZU9EG SoC and 4 GB DDR4 memory, while the Xilinx

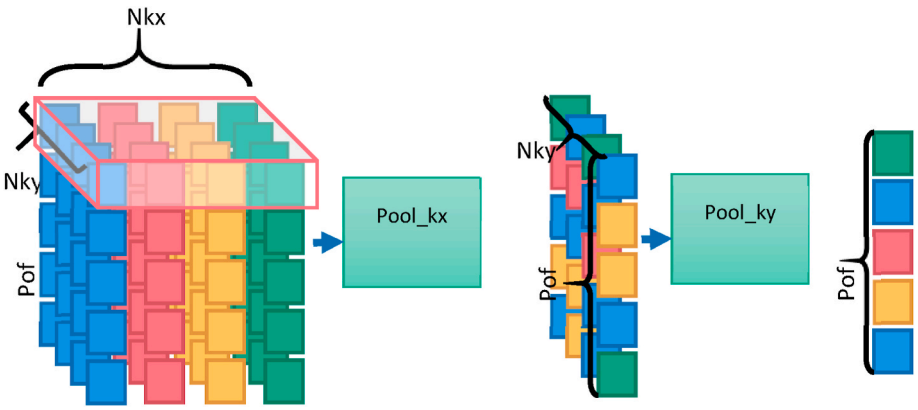

Fig. 8. The FPGA implementation of the pooling operation.

ZCU111 platform features the XCZU28DR SoC and 4 GB DDR4 memory. By configuring the unrolling size on Loop-2 and Loop-4, 1024 and 4096 MACs were implemented on two different boards, respectively, allowing us to observe the impact of different unrolling sizes and variations in hardware resource quantities on the proposed computation architecture. The quantity of resources and the actual utilization rate for the platforms are summarized in Table 2. The 16-bit dynamic fixed-point data arithmetic is employed for both kernel weights and intermediate pixel results and the data precision can be adjusted to trade classification accuracy for hardware utilization and throughput. The synthesis results are obtained from Xilinx Vivado 2022.1.

6.2. Analysis of experimental results

Table 3 presents the performance and the specification of the proposed CNN FPGA accelerator. Fig. 9 presents the breakdown of the end-to-end runtime of CNNs on the ZCU102 and ZCU111 FPGA. The convolutional layer runtime is decomposed to DRAM access of data and MAC computation, i.e. "CONV MAC" and "CONV_DRAM". Our accelerator has demonstrated outstanding performance across various CNNs. When considering only the CONV layers, the average performance of our system has reached as high as 1031.37 GOP/s, which outperforms prior designs [22,31,24]. Our accelerator implemented on the ZCU102 platform achieves computation efficiency over 80 % for VGG-16, ResNet-50, and ResNet-152. It should be noted that our research primarily focuses on how the proposed computation architecture can enhance the efficiency of convolutional operations. We have not specifically designed a computation architecture for the fully connected (FC) layer; instead, we have used the convolution module for its implementation.

To validate the computational efficiency of the proposed CNN accelerator for different sizes of convolutional layers, we implemented the VGG-16 CNN model [4] as our benchmark. The relevant parameters of the network are summarized in Table 4. Due to the lack of diversity in the convolution kernel size of VGG (all 3×3), we added two additional experimental groups as control tests. In these groups, the convolutional kernels in each block are changed to 1×1 and 5×5, just to evaluate the efficiency of the convolution layer with different sizes while ignoring the detection accuracy. The MAC efficiency for convolutional layers is computed as follows:

$$efficiency_{MAC} = \frac{fps_{CONV} \times Num_{multiplacation}}{Clock \times MACs} \tag{10}$$

As shown in Table 5, our FPGA accelerator achieves the CONV2, CONV3, and CONV4 efficiencies over 95 % and approximately 90 % overall efficiency when processing convolutions with kernel sizes of 3×3 (i.e. the original VGG) and 5×5 on both two platforms with different resource quantities. Only when handling some unfrequent CONV blocks e.g. CONV1 and CONV5, the computational efficiency is noticeably reduced. This is because the actual size of the input feature in terms of channels or length (width) gets small, resulting in poor utilization and computational efficiency.

6.3. Comparision with prior works

In order to demonstrate the effectiveness of our proposed method, the performance comparison results with the recent CNN FPGA accelerators [22,31,24] are presented in Table 6 and Fig. 10. Fig. 10 presents the roofline model for running the VGG-16 on our accelerator in comparison with those referenced in Refs. [31,24]. Notably, our accelerator architecture consistently approaches the roofline across various platform scales and when handling convolutional layers of different dimensions, demonstrating exceptional flexibility and scalability. Despite using fewer MAC units, specifically 1024, our accelerator achieved a computational efficiency that is 64.0 % higher and a throughput that is 36.5 % higher compared to Ref. [31]. When compared to Ref. [24], even with two-thirds fewer MAC units, we were able to achieve 4.7 % higher MAC efficiency when computing VGG-16's CONV blocks.

In the initial convolutional layer, conv1_1, with a notably low number of input channels ($N_{if} = 3$), our design exhibits less potential for unrolling along the input channel dimension compared to subsequent layers. This results in our approach demonstrating discernibly reduced performance for this layer in comparison to the designs presented in Refs. [31,24]. Nonetheless, the latency attributed to conv1_1 constitutes a mere 3 % of the overall latency for the VGG-16 model, rendering the performance disparity from this layer negligible. Predominantly optimized for unrolling across the input channel dimension, our design, despite its relatively modest performance in the conv1_1 layer, significantly enhances performance in other layers. Conversely, the strategies employed in Refs. [31,24], which hardly exploit unrolling in the input

Table 2
Resource utilization of Xilinx ZCU111.

Platform	SoC	precision	MACs	Resource			
				FF	LUT	DSP	BRAM
ZCU102	XCZU9EG	16	1024	91245/548160	140832/274080	512/2520	456/912
ZCU111	XCZU28DR	16	4096	158760/850560	314218/425280	2048/4272	627/1080

Table 3
Performance and specification of the proposed CNN FPGA accelerator on ZCU111.

FPGA	Clock (MHz)	MAC unit	Precision	$P_{if} \times P_{of} \times P_{ox}$	CNN	Latency (ms)		Throughput (GOPS)		Efficiency (%)	
						Overall	Conv	Overall	Conv	Overall	Conv
Xilinx ZCU102	150	1024	16 bit	32 × 32 × 32	VGG-16	165.59	110.25	186.92	276.30	59.5 %	89.4 %
					ResNet-50	40.84	38.56	189.65	200.10	75.6 %	80.1 %
					ResNet-152	102.74	96.67	220.18	233.74	79.0 %	84.0 %
Xilinx ZCU111	150	4096	16 bit	64 × 64 × 64	VGG-16	42.90	29.31	**721.48**	1031.38	60.9 %	89.1 %
					ResNet-50	14.67	13.32	**546.98**	579.28	52.6 %	57.9 %
					ResNet-152	34.04	30.87	**664.66**	731.97	59.7 %	65.8 %

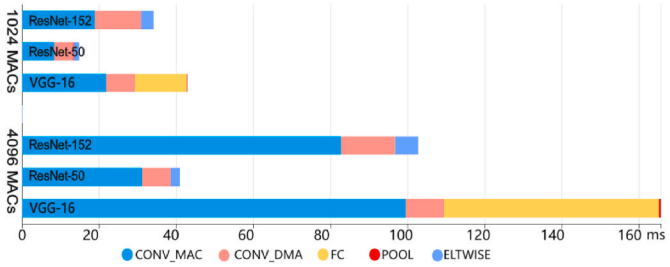

Fig. 9. Latency breakdown of proposed CNN accelerator on ZCU102 and ZCU111.

Table 4
Parameters of VGG-16 CONV blocks.

	Parameter (M)	Operation (G)	Multiplication (G)
CONV1	**0.04**	**3.87**	**1.94**
CONV2	**0.22**	**5.55**	**2.78**
CONV3	**1.47**	**9.25**	**4.63**
CONV4	**5.90**	**9.25**	**4.63**
CONV5	**7.08**	**2.31**	**1.37**
Total	**14.71**	**30.23**	**15.38**

channel dimension, do not experience a pronounced impact in this specific layer. However, this also precludes them from capitalizing on the advantages of Loop-2 unrolling for conserving partial sum storage.

Compared with [31,24], our proposed accelerator shows higher flexibility by handling not only conventional CNN, e.g. VGG-16 [4] but

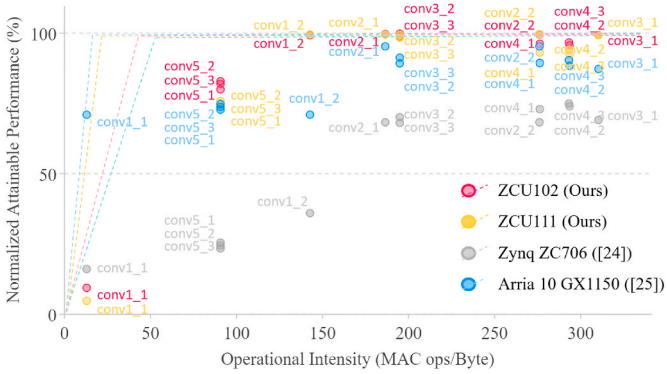

Fig. 10. The roofline model, where the horizontal axis represents the computational intensity of multiplication operations, and the vertical axis showcases normalized performance, comparing our implementation of the VGG-16 with those detailed in Refs. [31,24].

Table 5
Latency(ms) and MAC efficiency(%) of hardware implementation of VGG-16.

Platform	Clock (MHz)	MAC unit	Unrolling	Kernel	Latency(ms) and Efficiency (%) of VGG-16 Conv Group					
					CONV1	CONV2	CONV3	CONV4	CONV5	Overall
ZCU102	150	1024	$P_{if} = 32$, $P_{of} = 32$, $P_{ox} = 32$	1 × 1	6.57 ms, 21.4 %	3.82 ms, 52.5 %	4.86 ms, 68.8 %	4.67 ms, 71.5 %	1.62 ms, 61.9 %	21.54 ms, 51.5 %
				3 × 3	18.38 ms, 68.5 %	18.23 ms, 99.0 %	30.74 ms, 97.7 %	31.74 ms, 94.7 %	11.16 ms, 80.8 %	110.25 ms, 89.4 %
				5 × 5	50.55 ms, 69.3 %	50.68 ms, 98.7 %	87.54 ms, 95.3 %	91.13 ms, 91.4 %	32.19 ms, 79.2 %	315.09 ms, 88.3 %
ZCU111	150	4096	$P_{if} = 64$, $P_{of} = 64$, $P_{ox} = 64$	1 × 1	1.06 ms, 33.1 %	0.63 ms, 79.6 %	0.97 ms, 85.6 %	0.98 ms, 81.9 %	0.39 ms, 64.3 %	4.03 ms, 68.8 %
				3 × 3	6.06 ms, 52.0 %	4.53 ms, 99.4 %	7.57 ms, 99.1 %	8.07 ms, 92.9 %	3.04 ms, 74.6 %	29.3 ms, 89.1 %
				5 × 5	16.78 ms, 52.2 %	12.69 ms, 98.5 %	21.06 ms, 99.0 %	22.46 ms, 92.7 %	8.35 ms, 76.3 %	81.34 ms, 85.5 %

Table 6
Prior works on FPGA CNN accelerators for loop optimization.

	FPGA	Clock (MHz)	CNN	MACs	Latency (ms)	GOPS	Latency(ms) and Computational Efficiency (%)					
							CONV1	CONV2	CONV3	CONV4	CONV5	Overall
[31]	**Zynq ZC706**	150	VGG-16	1152	224.60	136.97	31.2 ms 35.8 %	23.5 ms 68.1 %	39.3 ms 68.9 %	36.2 ms 73.8 %	32.9 ms 24.3 %	163.4 ms 54.5 %
[24]	**Arria 10 GX 1150**	150	VGG-16	3136	47.97	645.25	5.8 ms 70.8 %	6.2 ms 95.1 %	11.0 ms 89.1 %	11.0 ms 89.2 %	4.4 ms 73.6 %	38.0 ms 84.7 %
[22]	**Arria 10 GX 1150**	150	ResNet-50	3136	12.51	619.13	/	/	/	/	/	11.5 ms 82.8 %
		150	ResNet-152	3136	31.85	719.30	/	/	/	/	/	29.8 ms 89.0 %

also highly complex and irregular CNNs, e.g. ResNets [5], through a reconfigurable execution schedule [22]. utilized 1518 DSP blocks on the Arria 10 GX 1150 FPGA, 231.7 K ALMs (logic modules in Intel FPGAs), and 2136 Intel FPGA BRAMs (~42 Mb) to achieve a throughput of 619.13 GOPS for ResNet-50 and 719.30 GOPS for ResNet-152. In comparison, our accelerator achieves throughputs of 546.98 GOPS and 664.66 GOPS for ResNet-50 and ResNet-152 (i.e., 92 % the throughput of [24]) with 2048 DSPs, 314 K LUTs, and 627 Xilinx FPGA BRAMs (~22.6 Mb) on the ZCU111 FPGA, i.e., 66 % of the logic elements (LE), 1.3 × the DSPs, and 54 % of the BRAMs utilized in Ref. [24]. Therefore, as our accelerator approaches the performance level of [24], each MAC unit in our accelerator consumes fewer logic cells on average and exhibits a more optimal resource distribution. This implies that our accelerator architecture can be better adapted to various platforms and networks. In conclusion, our CNN acceleration architecture exhibits outstanding performance across various networks, showcasing strong computational efficiency and generalization capabilities.

7. Conclusion

In this paper, we propose an efficient FPGA accelerator for CNN networks based on loop optimization. We first explored the quantitative relationship between the objectives of the accelerator and the variables involved in its design. Then, a practical guideline for creating an efficient acceleration strategy is developed. Based on this, we proposed an efficient convolution acceleration architecture and dataflow which can not only maximize the utilization of computation resources, but also reduce data movement and energy consumption effectively. Our convolution accelerator is verified on ZCU102 and ZCU111 FPGA, achieving throughputs of 721.48 GOPS 546.98 GOPS, and 664.66 GOPS for VGG-16 [4], ResNet-50 and ResNet-152 [5], respectively. The average MAC efficiency of the multiply-accumulator array reaches 89.4 %, which outperforms the state-of-the-art FPGA implementations. With the optimized CNN acceleration scheme and low communication dataflow, the proposed CNN accelerator uses uniform unrolling factors for all the convolution layers and fully utilizes all the DSP resources to achieve significant performance and efficiency improvements compared to previous FPGA-based CNN accelerators.

Funding

This research was funded by the >National Science and Technology Major Project under Grant 2019-I-0019-0018.

Credit authors statement

Yuhao Liu: Conceptualization, Methodology, Resources, Writing – original draft, Project administration. **Yanhua Ma:** Methodology, Formal analysis, Writing – review & editing. **Bowei Zhang:** Methodology, Resources. **Lu Liu:** Resources, Supervision. **Jie Wang:** Writing – review & editing, Supervision. **Shibo Tang:** Writing – review & editing, Supervision.

Declaration of competing interest

We declare that we have no financial and personal relationships with other people or organizations that can inappropriately influence our work, and there is no professional or other personal interest of any nature or kind in any product, service, and/or company that could be construed as influencing the position presented in, or the review of, the manuscript entitled. Through systematic exploration of loop optimization methods, we propose an efficient dataflow and FPGA-based convolution accelerator architecture.

Data availability

Data will be made available on request.

References

[1] Z. Zhang, C. Li, W. Zhang, et al., An FPGA-based memristor emulator for artificial neural network, Microelectron. J. 131 (2023) 105639.
[2] T.S. Technicolor, S.O.R. Related, T.S. Technicolor, et al., ImageNet Classification with Deep Convolutional Neural Networks, 50, 2023. -12-07].
[3] Y. Wang, Y. Liao, J. Yang, et al., An FPGA-based online reconfigurable CNN edge computing device for object detection, Microelectron. J. 137 (2023) 105805.
[4] K. Simonyan, A. Zisserman, Very deep convolutional networks for large-scale image recognition, Comput. Sci. (2014), https://doi.org/10.48550/arXiv.1409.1556.
[5] M. Shafiq, Z. Gu, Deep residual learning for image recognition: a survey[J], IEEE (2022), https://doi.org/10.1109/cvpr.2016.90.
[6] K He, X Zhang, S Ren, et al., Identity mappings in deep residual networks[C]// Computer Vision–ECCV 2016, 14, Springer International Publishing, 2016, pp. 630–645.
[7] C. Szegedy, S. Ioffe, V. Vanhoucke, et al., Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning, 2016, https://doi.org/10.48550/arXiv.1602.07261, 2023-12-07].
[8] C. Szegedy, W. Liu, Y. Jia, et al., Going deeper with convolutions[C], in: 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), IEEE, 2015, https://doi.org/10.1109/CVPR.2015.7298594.
[9] S. Ren, K. He, R. Girshick, et al., Faster R-CNN: towards Real-Time Object Detection with Region Proposal Networks[C], NIPS, 2016, https://doi.org/10.1109/tpami.2016.2577031.
[10] H. Kwon, L. Lai, M. Pellauer, et al., Heterogeneous dataflow accelerators for multi-DNN workloads.[C]//High-Performance computer architecture, IEEE (2019), https://doi.org/10.1109/HPCA51647.2021.00016.
[11] S. Han, H. Mao, W.J. Dally, Deep compression: compressing deep neural networks with pruning, trained quantization and huffman coding, Fiber 56 (4) (2015) 3–7, https://doi.org/10.48550/arXiv.1510.00149.
[12] C. Zhang, P. Li, G. Sun, et al., Optimizing FPGA-based accelerator design for deep convolutional neural networks[C], in: Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 2015, pp. 161–170.
[13] D.F. Bacon, S.L. Graham, O.J. Sharp, Compiler transformations for high-performance computing, ACM Comput. Surv. 26 (4) (1994) 345–420.
[14] Y.H. Chen, J. Emer, V. Sze, Eyeriss: a spatial architecture for energy-efficient dataflow for convolutional neural networks[J], IEEE (2017), https://doi.org/10.1109/mm.2017.265085944.
[15] P.Y. Hsiao, S.Y. Lin, S.S. Huang, An FPGA based human detection system with embedded platform, Microelectron. Eng. 138 (2015) 42–46, https://doi.org/10.1016/j.mee.2015.01.018.
[16] S. Colleman, T. Verelst, L. Mei, et al., Processor Architecture Optimization for Spatially Dynamic Neural networks[C], in: 2021 IFIP/IEEE 29th International Conference on Very Large Scale Integration (VLSI-SoC), IEEE, 2021, pp. 1–6.
[17] L. Mei, P. Houshmand, V. Jain, et al., ZigZag: A Memory-Centric Rapid DNN Accelerator Design Space Exploration Framework, 2020, https://doi.org/10.48550/arXiv.2007.11360.
[18] S. Colleman, M. Verhelst, High-utilization, high-flexibility depth-first CNN coprocessor for image pixel processing on FPGA, IEEE Trans. Very Large Scale Integr. Syst. (2021), https://doi.org/10.1109/TVLSI.2020.3046125.
[19] J.W. Jang, S. Lee, D. Kim, et al., Sparsity-aware and Re-configurable NPU architecture for samsung flagship mobile SoC[C], in: 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA), ACM, 2021, https://doi.org/10.1109/ISCA52012.2021.00011.
[20] X. Li, H. Huang, T. Chen, et al., A hardware-efficient computing engine for FPGA-based deep convolutional neural network accelerator, Microelectron. J. 128 (2022) 105547.
[21] S. Colleman, M. Shi, M. Verhelst, >COAC: cross-layer optimization of accelerator configurability for efficient CNN processing, IEEE Trans, Very Large Scale Integr 31 (7) (July 2023) 945–958, https://doi.org/10.1109/TVLSI.2023.3268084.
[22] Y. Ma, Y. Cao, S. Vrudhula, et al., An automatic RTL compiler for high-throughput FPGA implementation of diverse deep convolutional neural networks, IEEE (2017), https://doi.org/10.23919/FPL.2017.8056824.
[23] Y. Guan, H. Liang, N. Xu, et al., FP-DNN: an automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates, in: 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), IEEE, 2017, https://doi.org/10.1109/FCCM.2017.25.
[24] Y. Ma, Y. Cao, S. Vrudhula, et al., Optimizing loop operation and dataflow in FPGA acceleration of deep convolutional neural networks[C], in: Acm/sigda International Symposium on Field-Programmable Gate Arrays, ACM, 2017, https://doi.org/10.1145/3020078.3021736.
[25] Y.H. Chen, T. Krishna, J.S. Emer, et al., Eyeriss: an energy-efficient reconfigurable accelerator for deep convolutional neural networks, IEEE (2016), https://doi.org/10.1109/ISSCC.2016.7418007.
[26] Y. Ma, N. Suda, Y. Cao, et al., in: Scalable and Modularized RTL Compilation of Convolutional Neural Networks onto FPGA[C]//Field-Programmable Logic and Applications, Institute of Electrical and Electronics Engineers Inc., 2016, https://doi.org/10.1109/fpl.2016.7577356.

[27] N. Suda, V. Chandra, G. Dasika, et al., Throughput-Optimized OpenCL-Based FPGA Accelerator for Large-Scale Convolutional Neural Networks[C], in: Acm/sigda International Symposium, ACM, 2016, pp. 16–25, https://doi.org/10.1145/2847263.2847276.

[28] H. Li, X. Fan, L. Jiao, et al., A high performance FPGA-based accelerator for large-scale convolutional neural networks[C], in: 2016 26th International Conference on Field Programmable Logic and Applications (FPL), IEEE, 2016, https://doi.org/10.1109/FPL.2016.7577308.

[29] M. Motamedi, P. Gysel, V. Akella, et al., Design space exploration of FPGA-based deep convolutional neural networks[C], in: 2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC), IEEE, 2016, pp. 575–580.

[30] A. Rahman, J. Lee, K. Choi, Efficient FPGA acceleration of convolutional neural networks using logical-3D compute array[C]//2016 Design, in: Automation & Test in Europe Conference & Exhibition (DATE), IEEE, 2016, pp. 1393–1398.

[31] J. Qiu, J. Wang, S. Yao, et al., Going deeper with embedded FPGA platform for convolutional neural network[C], in: Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 2016, pp. 26–35.