IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 27 July 2022, accepted 9 August 2022, date of publication 17 August 2022, date of current version 6 September 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3199441

RESEARCH ARTICLE

Design and Implementation of Convolutional Neural Networks Accelerator Based on Multidie

QINGZENG SONG[1], JIABING ZHANG[1], LIANKUN SUN[1], AND GUANGHAO JIN[2], (Member, IEEE)

[1]School of Computer Science and Technology, Tiangong University, Tianjin 300387, China
[2]School of Telecommunication Engineering, Beijing Polytechnic, Beijing 100176, China

Corresponding author: Guanghao Jin (jingh_research@163.com)

This work was supported in part by the National Natural Science Foundation of China under Grant 61802279, Grant 6180021345, Grant 61702281, and Grant 61702366; in part by the Natural Science Foundation of Tianjin under Grant 18JCQNJC70300, Grant 19JCTPJC49200, Grant 19PTZWHZ00020, and Grant 19JCYBJC15800; in part by the Fundamental Research Funds for the Tianjin Universities under Grant 2019KJ019; in part by The Tianjin Science and Technology Program under Grant 19PTZWHZ00020; in part by the State Key Laboratory of ASIC and System under Grant 2021KF014 and Grant 2021KF015; in part by the fund of Beijing Polytechnic under Grant 2022 × 017-KXZ; and in part by the Tianjin Educational Commission Scientific Research Program Project under Grant 2020KJ112 and Grant 2018KJ215.

ABSTRACT To achieve real-time object detection tasks with high throughput and low latency, this paper proposes a multi-die hardware accelerator architecture. It implements three accelerators on the VU9P chip, each of which is bound to an independent super logic region (SLR). To reduce off-chip memory access and power consumption, this design uses three on-chip buffers to store the weights and intermediate result data on one hand; on the other hand, it minimizes data access and movement and maximizes data reuse. This design uses an 8-bit quantization strategy for both weights and feature maps to achieve twice the throughput and computational efficiency of a single digital signal processor (DSP). In addition, many operators are designed in the accelerator, and all of them are fully parameterized, so it is easy to extend the network, and the control of the accelerator can be realized by configuring the instruction group. By accelerating the YOLOv4-tiny algorithm, the accelerator architecture can achieve a frame rate of 148.14 frames per second (FPS) and a peak throughput of 2.76 tera operations per second (TOPS) at 200 MHz with an energy efficiency ratio of 93.15 GOPS/W. The code can be found at https://github.com/19801201/Verilog_CNN_Accelerator.

INDEX TERMS Hardware accelerator, multi-die, object detection, YOLOv4-tiny.

I. INTRODUCTION

As one of the most representative networks in deep learning, convolutional neural networks (CNNs) achieved good performance in object detection [1], [2], image classification [3], [4], image segmentation [5], [6], and other fields. To capture more features, the structure of the model becomes larger, which leads to the high computational complexity of CNN. As a result, its computational process requires more computing power and storage resources. Furthermore, the complex structure causes it not suitable for the general-purpose calculation and serial calculation of central processing units (CPUs). On the contrary, graphics processing units (GPUs), application-specific integrated circuits (ASICs), and field-programmable gate arrays (FPGAs) can meet the requirements well in terms of computing power and computational parallelism. However, the high cost and power consumption of GPUs, or the lack of programmability of ASICs cause these devices to be unsuitable for accelerating convolutional neural networks at the edge. In contrast, FPGAs have a good balance between flexibility, performance, power consumption, and cost, and are ideal devices for inference acceleration of convolutional neural networks.

Over the past 50 years, chip process technology has been following Moore's Law development, but in recent years, chip process technology is getting closer to the physical limit, society has entered the post-Moore's Law era, and the major semiconductor manufacturers have to find new technologies

The associate editor coordinating the review of this manuscript and approving it for publication was Mario Donato Marino.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

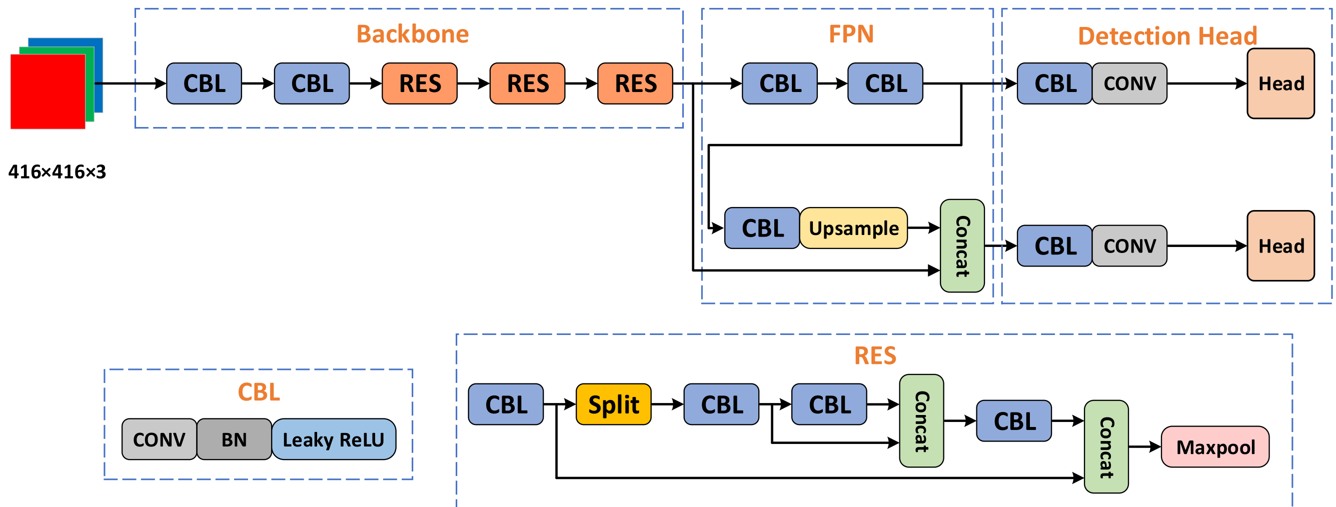

FIGURE 1. Network structure of YOLOv4-tiny.

to break through the limits of Moore's Law. In response, AMD Xilinx has introduced stacked silicon interconnect (SSI) [7] technology, which packages multiple small chips into a single chip with greater bandwidth and capacity, and has been applied to several families of products. This design is a multi-die-based convolutional neural network accelerator design using Virtex UltraScale+ VU9P chip based on SSI technology, which is very important for multitasking and batch processing in convolutional neural networks.

The main contributions of this work are summarized as the following:

(1) A multi-die-based convolutional neural network accelerator design framework is proposed, in which three accelerators are integrated into one VU9P chip and each accelerator is constrained to each individual SLR so that they have their independent resources and perform computations independently without interfering with each other.

(2) The intermediate results of each layer of calculation are no longer written back to the DDR but to an on-chip buffer consisting of URAM. This design reduces frequent access to the DDR, increases computation speed, and reduces power consumption.

(3) This accelerator maximizes the reuse of input feature maps and weights. The amount of data read from DDR and on-chip buffers for weights and input feature maps is minimized for each layer of computation, reducing the pressure on bandwidth and AXI bus.

The rest of the paper is organized as follows. Section II introduces convolutional neural networks and optimization strategies. Section III introduces the overall architecture design of the system. Section IV describes the architectural design of each module in more detail. Section V introduces the instruction group and host computer scheduling. Section VI discusses the experimental setup and the analysis of the experimental results. Finally, the conclusions are drawn in Sec.VII.

II. BACKGROUND AND RELATED WORK

A. CONVOLUTIONAL NEURAL NETWORKS

The current target detection and recognition algorithms based on convolutional neural networks are mainly divided into two categories: single-stage target detection algorithms and two-stage target detection algorithms. The two-stage target detection algorithm mainly generates candidate boxes on the input image first, and then classifies and regresses these candidate boxes, represented by R-CNN [8] and Faster R-CNN [9]. Although their accuracy is high, their detection speed is very slow. In contrast, the single-stage target detection algorithm represented by you only look once (YOLO) [10] series abandons the two-step frame drawing approach of the R-CNN series and uses only one CNN network to achieve target detection, so its detection speed is particularly fast and can be well applied to scenarios requiring high real-time performance, such as automatic driving.

Since Joseph Redmon proposed YOLO [10], the YOLO series has gone through many versions. YOLOv4-tiny is a simplified version of YOLOv4 [11], which is a lightweight model with only about six million parameters, one-tenth of YOLOv4. Due to its simple network structure, small computational and parametric quantities, and fast detection speed, it is well suited for inference acceleration on resource-constrained edge computing platforms.

The network structure of YOLOv4-tiny when the network input is 416 × 416 × 3 in size is shown in Fig.1, which mainly consists of three parts: backbone, feature pyramid network (FPN), and detection head. YOLOv4-tiny uses CSPdarknet53_tiny as the backbone feature extraction network, which reduces the number of layers and modifies the activation function to Leaky ReLU to be faster compared to CSPdarknet53. FPN mainly performs feature fusion on the effective feature layer generated by the backbone network. The detection head is mainly to extract the final two feature layers. YOLOv4-tiny network has a total of 21 convolutional

layers, 3 split layers, 3 max-pooling layers, 7 concat layers and 1 upsample layer.

B. OPERATOR FUSION

In the network structure of the YOLO family, the convolutional block is usually composed of three parts: convolutional layer, batch normalization (BN) [12] layer, and activation function layer. The use of batch normalization allows the use of larger learning rates to reduce the training time of neural networks, it allows deep networks to use saturating activation functions such as sigmoid, tanh, etc. to alleviate the problem of gradient disappearance and gradient explosion, and it has some regularization effect to reduce overfitting.

However, in the network inference stage, since the batch normalization layer often follows the convolutional layer, the operations of the batch normalization layer are often fused into the convolutional layer to reduce the computation and accelerate the inference.

C. 8-BIT QUANTIZATION

The parameters obtained after network model training are often in FP32 format, and when inferring convolutional neural networks on FPGAs, they are not suitable for floating-point calculations due to the circuit characteristics of FPGAs and their resource limitations, so quantization of floating-point models is required. Mittal [13] compared different quantization schemes and quantization bit widths and found that the accuracy loss of the network was stable within 1% when the quantization bit width was 8 bits and above, while the accuracy loss was significant when quantization of lower bit widths such as 6 bits and 4 bits was performed. Considering the above factors, the 8-bit quantization scheme is adopted in this design. It is shown that, compared with 32-bit floating-point numbers, when using 8-bit fixed-point numbers for network inference, can greatly reduce the number of parameters, the bandwidth requirement, and the hardware computation pressure, thus increasing the inference speed, while guaranteeing almost no loss of accuracy.

This design performs asymmetric quantization of feature map data into UINT8 format and symmetric quantization of weight data into INT8 format. The API function of PyTorch is used to implement both feature data and weight data. In addition, after the convolution of two 8-bit data, the convolution calculation result is no longer 8 bits, and the convolution calculation result is to be used as the input feature map of the next layer, so the 8-bit quantization of the convolution calculation result is also performed. According to the quantization scheme of [14], we can obtain the mapping relation from the integer q to the real number r, as shown in Equation (1):

$$r = s(q - z) \tag{1}$$

where r is the real value, q is the quantized value, and s and z are both quantization parameters. The convolution calculation formula is shown in Equation (2):

$$r_3 = r_1 r_2 + bias \tag{2}$$

Among them, r_3 is the result of the convolution calculation, r_1 is the input feature map, and r_2 is the input weight. Putting Equation (1) into Equation (2), we get Equation (3):

$$s_3(q_3 - z_3) = s_1(q_1 - z_1) \times s_2(q_2 - z_2) + bias \tag{3}$$

Since the weight data is symmetrically quantized, z_2 is 0, and Equation (3) is transformed to obtain Equation (4):

$$q_3 = \frac{s_1 s_2}{s_3}(q_1 q_2 - q_2 z_1 + \frac{bias}{s_1 s_2}) + z_3 \tag{4}$$

In Equation (4), $q_1 q_2$ is the convolution result of 8-bit weights and input feature maps, and we define the parameters $M = \frac{s_1 s_2}{s_3}$, $B = \frac{bias}{s_1 s_2} - q_2 z_1$, $Z = z_3$, then Equation (4) is expressed as Equation (5):

$$q_3 = M(q_1 q_2 + B) + Z \tag{5}$$

The TVM compiler can resolve the parameters M, B, and Z. In the hardware implementation, the quantization value of the convolution result can be obtained by first adding the parameter B to the convolution result, then multiplying the parameter M, and finally adding the parameter Z.

III. SYSTEM ARCHITECTURE DESIGN

The VU9P chip is AMD Xilinx's third-generation 3D IC [15] product, which uses SSI technology to package three dies into one chip with greater bandwidth and capacity. Each die is also called an SLR, and the resources of each SLR are independent of each other and can be regarded as a single small-scale FPGA chip. For the VU9P chip, each SLR can be designed individually, or multiple SLRs can be combined as one large chip. In this design, three convolutional neural network accelerators are instantiated for each SLR individually, and the three accelerators are constrained in three separate SLRs by creating Pblocks in Floorplanning mode in the Vivado development tool. The three accelerators only use the resources of their own SLRs without interfering with each other, except for sharing the PCIe of the main SLR.

A. MULTIDIE DESIGN

The multi-die design framework is shown in Fig.2. The Host PC is mainly responsible for the pre-processing of input feature maps, sending and receiving data and task scheduling of the FPGA board, decoding of output feature maps, and non-maximum suppression (NMS).

First, we train the YOLOv4-tiny network on the GPU server using the PyTorch framework. Then we select the most effective parameters from the list of parameters obtained from the training and provide the network structure and the selected parameters to the TVM compiler. The TVM compiler can extract the weights required for each layer of the convolution operation and the parameters M, B, and Z required for quantization based on the network structure and this parameter. At the same time, the TVM compiler reorders this data according to the calculation method of the hardware accelerator and converts this data into the binary files required by the accelerator.

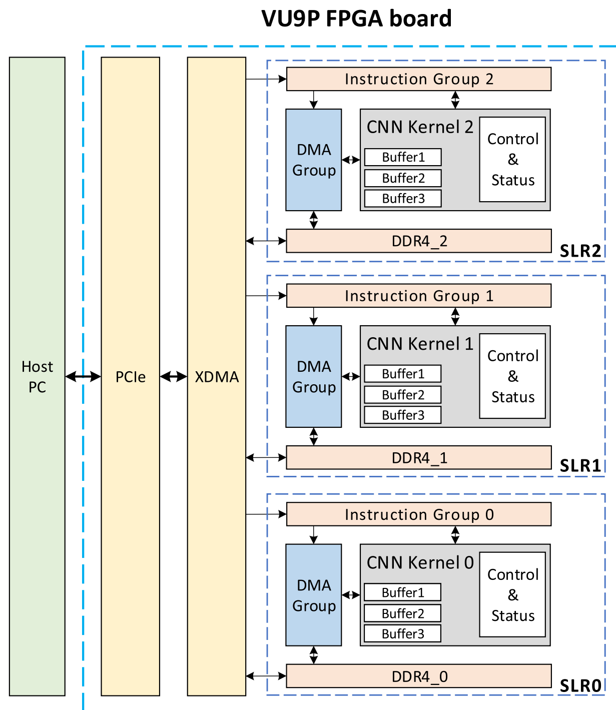

FIGURE 2. Multi-die-based CNN accelerator architecture.

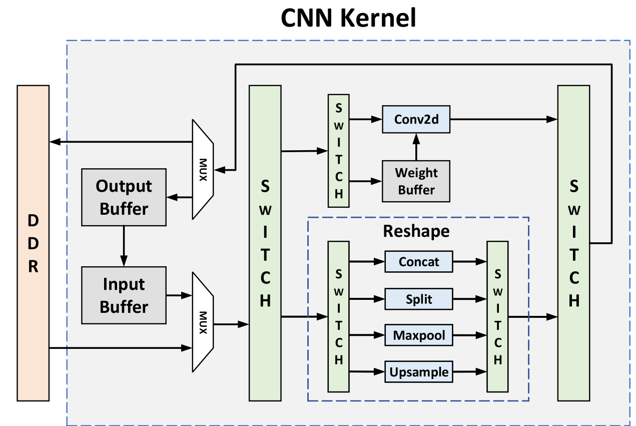

FIGURE 3. CNN kernel architecture design.

The host computer sends the data such as weights, feature maps, and instructions required for the accelerator calculation from the PC to the FPGA board through the PCIe interface. As can be seen in Fig.2, the VU9P board deploys three convolutional neural network accelerators, each belonging to a separate SLR with its own independent DDR and Instruction Group. The DDRs and instruction groups of different SLRs are distinguished by their addresses. The host computer writes data and instructions to the corresponding address for whichever accelerator it wants to start, or it can write data and instructions at the same time to start multiple accelerators executing in parallel.

Taking a single accelerator as an example, the host computer of the Host PC firstly pre-processes the input feature map and generates the instructions required for the calculation, then connects to PCIe interface through XDMA, writes the weights, quantization parameters, and feature maps to the DDR through AXI interface, and writes the instructions to the instruction group through AXI-Lite interface. The DMA Group is controlled by the instruction to read the weights and input feature maps from the DDR and send them to the CNN Kernel for computation. The results of each layer are stored in the on-chip buffer of the CNN Kernel, and the DMA Group writes the final results back to the DDR after all layers have been computed.

The CNN kernel consists of several modules, which on the one hand select which module to execute through instruction control, and on the other hand feed the current state to the instruction group in real-time. The instruction group decides whether to execute the next instruction or to continue waiting based on the status of the CNN kernel. After all the instructions are executed, the detection heads of the last two output feature layers will be sent to the Host PC through PCIe for post-processing operations such as decoding and non-maximum suppression.

B. CNN KERNEL ARCHITECTURE DESIGN

The CNN Kernel is mainly developed based on operators, and the convolutional neural network accelerator is implemented by implementing various operators. Fig.3 shows the architecture design of CNN Kernel, which is mainly composed of on-chip buffers and various operator modules. The on-chip buffers include input buffer, output buffer, and weight buffer [16], which are represented by gray boxes in Fig.3. The input buffer and output buffer for storing intermediate results are implemented by URAM, while the weight buffer cannot be implemented by URAM due to the inconsistent bit width of input and output data, so we use BRAM for this purpose. As can be seen in Fig.3, the data required for the calculation of each operator module can be read either from the DDR or from the input buffer. Similarly, the results generated by the computation can be written either to the DDR or to the output buffer. The selection of these paths is controlled through instructions. To minimize the power consumption caused by off-chip DDR accesses, we write only the computation results of the first two layers in the YOLOv4-tiny network and the final prediction results back to the DDR and write the computation results of all other layers to the output buffer. However, since the weights are so large that they can only be stored in the DDR, the weights are read from the DDR and written to the weight buffer during each convolution calculation.

When starting the calculation, the input buffer and the weight buffer send the data to the operator module for calculation, and the output result of each layer is saved in the output buffer. After this calculation is completed, the data in the output buffer is immediately sent to the input buffer to wait for the next calculation to be read. After all calculations are completed, the results are written to the DDR via the DMA Group. The blue boxes in Fig.3 are the operator modules implemented by the accelerator, including Conv2d which can implement 3×3 convolution and 1×1 convolution, Concat,

Split, Maxpool, and Upsample. These operators are based on a parametric design and can be adapted to the most common dimensions. This means that any convolutional neural network can be inferentially accelerated by this convolutional neural network accelerator, as long as its internal operators are within the range of these operators and their dimensions.

In this architecture, the data are transmitted in the form of AXI-Stream. When the calculation is performed, the data stream is output from the input buffer and the weight buffer. First, the data stream goes through a SWITCH (path selector) to choose which module to enter according to the control signal. If the data stream enters the Conv2d module, the calculation is started directly. If the data stream enters the Reshape module, it will go through another SWITCH, and the control signal is used to select one of the four operator modules of Concat, Split, Maxpool, and Upsample for calculation. After the calculation is finished, the result is written into the output buffer through SWITCH.

IV. SYSTEM DETAILED DESIGN

A. Conv2d MODULE DESIGN

The Conv2d module is the most important operation for feature extraction in convolutional neural networks. This design uses mainly 3×3 convolution and 1×1 convolution. Compared with larger convolution operations such as 5×5 and 7×7, the 3×3 convolution operation can achieve better results while reducing the number of parameters.

The Conv2d module is mainly used to perform the convolution operation of input feature maps and weights, and the 8-bit quantization operation of the convolution results. The required parameters include the parameters M, B, and Z described in Section II-C in addition to the weights. As shown in Fig.2, before the convolution calculation, the input feature maps, weights, and all parameters required for the calculation have been stored in the DDR. When each layer is calculated, the DMA Group reads the weights and the parameters required for the calculation of this layer from the DDR and puts them into the weight buffer of the CNN Kernel. The input feature maps are only loaded by the DDR to the CNN Kernel for computation in the first and second layers, while the other layers are loaded from the output buffer to the input buffer and then to the CNN Kernel for computation. When the calculation is performed, the input buffer inputs the data to the internal Conv2d operator module, and Conv2d internally reads the corresponding weights from the weight buffer and performs the convolution calculation with the input feature maps. For the convolution result, the parameters M, B, and Z are also read from the weight buffer for 8-bit quantization calculation and Leaky ReLU calculation, and the result is written to the output buffer after the calculation is complete.

When performing the convolution calculation, this design adopts the principle of channel priority, and the calculation of the next pixel point is started only after all channels of a pixel point of the output feature map have been calculated. In the convolution operation of Fig.4, the input feature map

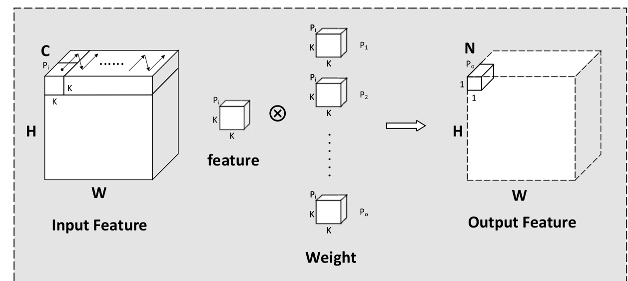

FIGURE 4. Conv2d convolution operation.

TABLE 1. Hardware parameters defined in this accelerator architecture.

Parameter	Description
H	High of input feature maps
W	Width of input feature maps
C	Channels of input feature maps
N	Channels of output feature maps
P_i	Parallelism of input channels
P_o	Parallelism of output channels
P_p	Parallelism of pixels
$Freq$	Frequency of accelerator operation

dimension is $H \times W \times C$, the convolution kernel dimension is $K \times K \times C \times N$ and the output feature map dimension is $H \times W \times N$. The dimension of each sliding cube and weight block is $K \times K \times P_i$. Table 1 lists all the hardware parameters in this accelerator architecture.

The Conv2d module can be configured for regular 3×3 convolution and PW convolution via instruction. For the conventional 3×3 convolution, this design selects P_i, P_o and P_p as 16, 8 and 9, respectively, after considering the DSP resources of the VU9P board. For the PW convolution, since it is a 1×1 convolution, to maximize the resources occupied by the nine pixels of the 3×3 convolution, we split the input channels of the PW convolution into eight parts and send them sequentially to the resources occupied by the first eight pixel points of the 3×3 convolution for computation, and the other pixel is temporarily not used. In this case, the PW convolution has P_i, P_o and P_p of 128, 8, and 1, respectively. In addition, since some PW convolutions have only 64 input channels, this design also supports the case where P_i is 64.

The process of convolution is as follows: the input sliding cube moves from the beginning to the end of the channel dimension, each sliding cube is convolved with P_o weight blocks, and the temporary result of P_i input and P_o output is stored in a temporary buffer. Next, the sliding cube slides along the channel direction, and each sliding cube is convolved with different P_o weight blocks, and the obtained P_i input and P_o output results are accumulated with the temporary buffer and stored in the temporary buffer. Until the last sliding cube of this input channel is calculated, the complete P_o channel calculation of the first pixel point of the output feature map is calculated and output to the quantization module for quantization processing. The next sliding cube returns

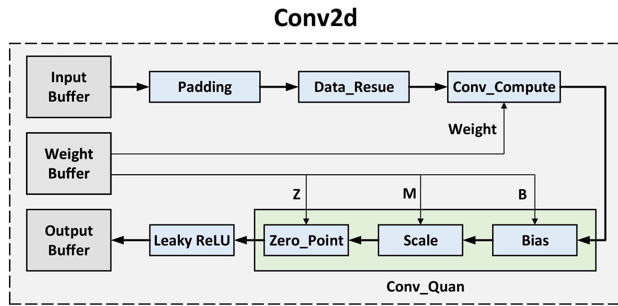

FIGURE 5. Conv2d calculation process.

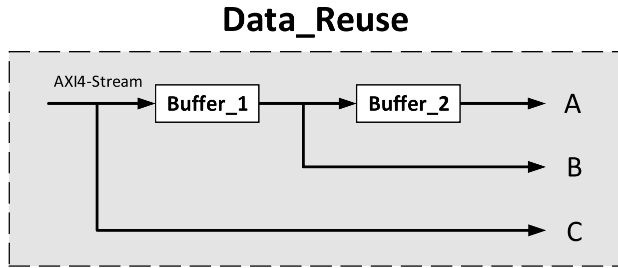

FIGURE 6. Data_Reuse architecture design.

to the starting pixel point of that channel and repeats the above steps until the convolution operation is done with all the convolution kernels, then all the N channels of the first pixel point of the output feature map are computed. Then the input sliding cube continues the sliding computation until the last pixel point is computed.

Throughout the computation, this design maximizes the reuse of input feature maps and weights. The input feature maps are read repeatedly N/P_o times, and the weights are reused $H \times W$ times. Such a scheme is designed so that the parameters read from the DDR and the input feature maps read from the input buffer at each layer of the computation are the smallest amount of data, thus reducing the power consumption caused by frequent accesses.

In the Conv2d module, the convolution operation is only the core part, and some other operations are needed to cooperate with the convolution operation to complete the calculation of the whole module. As shown in Fig.5, after the input feature map enters the Conv2d module from the input buffer, it first passes through the Padding module. PyTorch supports four padding modes: zeros, reflect, replicate, and circular. Since the purpose of the accelerator is target detection, the default zeros mode is sufficient. This design mainly performs padding operation on 3×3 convolution, padding a circle of zeros around the channel dimension of the input feature map can ensure that the convolution with a stride of one can maintain the boundary information and keep the output dimension consistent with the input. The function of the Padding module is to pad a circle of zeros around the input feature maps.

Then the data enters the Data_Reuse module for data reuse. In the convolution operation, pixels at the edges of the image and pixels in the middle of the image are used multiple times. The Data_Reuse module is to copy the input feature map into multiple copies to meet the needs of the convolution operation. The architecture design of the Data_Reuse module is shown in Fig.6. The Data_Reuse module has one input port and three output ports, and each output port corresponds to a data path. The data stream must pass through Buffer_1 and Buffer_2 successively to reach port A, only after passing through Buffer_1 to reach port B, and reaching port C without passing through the buffer. Each buffer can store one whole row of data for the input feature map. When Buffer_2 is full of the first row of data and Buffer_1 is full of the second row of data, the read enable of the three ports is turned on at the same time. This allows the same number of columns of the three adjacent rows to be fetched in the same clock cycle. The three adjacent rows of data are delayed by one clock cycle and two clock cycles, respectively. The nine pixel points required for the 3×3 convolution calculation can be obtained by taking the values at the same time when delaying the second clock cycle. The data from these nine pixel points are then sent to the Conv_Compute module for computation in the same clock cycle.

The calculation of the Conv_Compute module is the convolution calculation process shown in Fig.4. The result of the calculation is 32-bit fixed-point data. The 32-bit fixed-point convolution result needs to be quantized to 8-bit output to be used by the next layer. The Conv_Quan module is to do this quantization process, and its internal Bias, Scale, and Zero_Point sub-modules correspond to the operations of adding parameter B, multiplying parameter M, and adding parameter Z in Equation (5), respectively. The quantized result becomes 8 bits and then enters the Leaky ReLU module. The coefficient of Leaky ReLU is chosen to be 0.125, so that the floating-point multiplication operation can be replaced by only a 3-bit right-shift operation in FPGA design, saving computational cost. After the Leaky ReLU operation is completed, the data completes the calculation of the whole Conv2d module, and then the data is written to the output buffer and waits for the next calculation.

B. CONCAT MODULE DESIGN

The Concat module mainly performs the stitching of the two input feature maps in a certain dimension. Since the torch.cat() function provided by PyTorch is floating-point, it needs to be quantized to 8-bit for calculation output. In this design, the two input feature maps are quantized first before the stitching operation is performed in the channel dimension. The quantization formula is shown in Equation (6):

$$s_3(q_3 - z_3) = s_1(q_1 - z_1) \tag{6}$$

The meaning of each parameter is the same as in Section II-C. Transform Equation (6) to get Equation (7):

$$q_3 = \frac{s_1}{s_3}(q_1 - z_1 + \frac{s_3}{s_1}z_3) \tag{7}$$

By defining the parameters $scale = \frac{s_1}{s_3}$, $bias = \frac{s_3}{s_1}z_3 - z_1$, the quantization result is obtained by simply adding $bias$ to

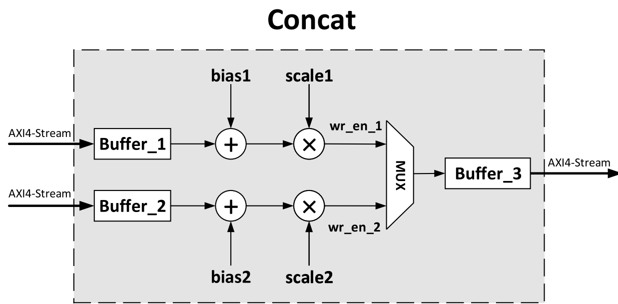

FIGURE 7. Concat architecture design.

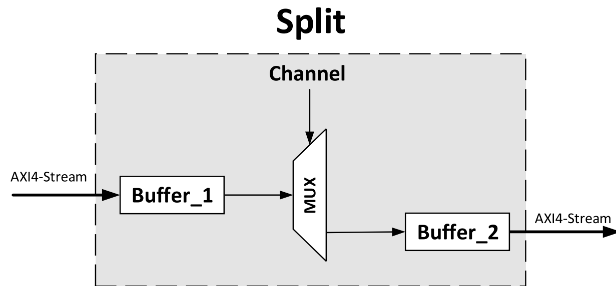

FIGURE 8. Split architecture design.

the input feature map data and then multiplying by *scale*. The parameters *bias* and *scale* are also parsed by the TVM compiler and passed into the Concat module through the instruction registers when performing Concat calculations. The detailed module design is shown in Fig.7.

The two groups of data in AXI4-Stream format enter the Concat module at the same time and enter Buffer_1 and Buffer_2 respectively. First, all channel data of one pixel point are read from Buffer_1, then *bias*1 is added to these data, and then the result is multiplied by *scale*1, that is, the quantization operation is completed, and the quantization result is written to Buffer_3 via the multiplexer. Buffer_3 converts the data into AXI4-Stream format for output. After all the channel data of one pixel in Buffer_1 is written into Buffer_3, start reading data from Buffer_2, and then perform the same operation. Similarly, after all channel data of a pixel of Buffer_2 is written into Buffer_3, Buffer_1 is read again, and so on, until all the data is processed, and the Concat module is finished.

C. SPLIT MODULE DESIGN

The Split module is mainly used for channel splitting of the input feature map, taking the first half of the channel data or the second half of the channel data as the backbone part for output, whether the first half or the second half of the channel is taken is controlled by instructions. In this design, the second half of the channel data is taken as the output, and the first half of the channel data is discarded directly. The module design is shown in Fig.8.

The data in AXI4-Stream format enters the Split module first into the temporary buffer Buffer_1, and then the data output from Buffer_1 goes through the multiplexer to select the channel. No processing is done for the data of the first half channels of the input feature map, and only the data of the second half channels are written to Buffer_2, which converts the data into AXI4-Stream format for output, that is, the output result of Split is obtained.

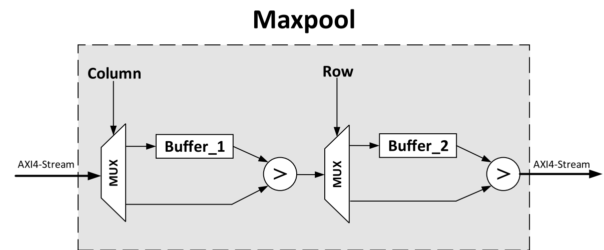

FIGURE 9. Maxpool architecture design.

D. MAXPOOL MODULE DESIGN

The Maxpool module mainly performs the dimensionality reduction operation on the input feature map, which can reduce the number of parameters of the model, save the model arithmetic power, and simplify the complexity of the model. Average pooling also has the function of dimensionality reduction, but it focuses more on feature extraction of background information rather than texture features that our target detection focuses on, so we do not use average pooling in our algorithmic network. Maxpool mainly partitions the whole input feature map into several small blocks of the same size according to the distribution of rows and columns, compares within each block, takes the maximum value as the output, and discards the rest of the values directly. This design refers to the pooling solution proposed by Nguyen *et al*. [17]. The input feature map is divided into small windows of 2×2 according to the rows and columns, and a stride of 2 sliding to the right and sliding down is performed for comparison. The module design architecture is shown in Fig.9.

AXI4-Stream format data enters the Maxpool module and starts to count channels, columns, and rows, and the data is discharged in the order of channel first, then column and row. Firstly, the data enters the column selector, and the column selector puts the data of the odd-numbered column into the temporary buffer Buffer_1, and after Buffer_1 is full of all channel data of one pixel, it starts to enter the even-numbered column, and the column selector directly outputs the data of the even-numbered columns to the comparator, and at the same time reads the data from Buffer_1 and outputs it to the comparator. At this point, the corresponding channel data of adjacent columns will start to be compared, and the larger value will be output to the row selector in the next step, while the smaller value will be discarded directly. After the data reaches the row selector, the data of odd-numbered row data are written to Buffer_2, and the data of even-numbered row data directly to the comparator for comparison with the output value of Buffer_2, and the comparison output result is the final result of Maxpool module. In addition, the current architecture also supports the average pooling operation, just replace the first comparator with an adder and right-shift

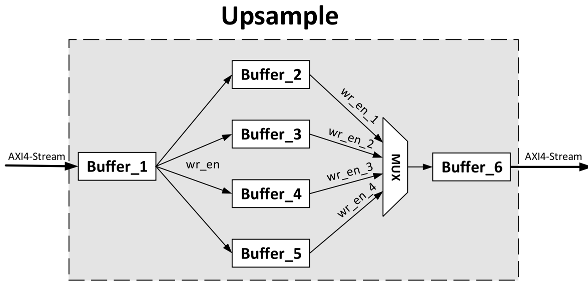

FIGURE 10. Upsample architecture design.

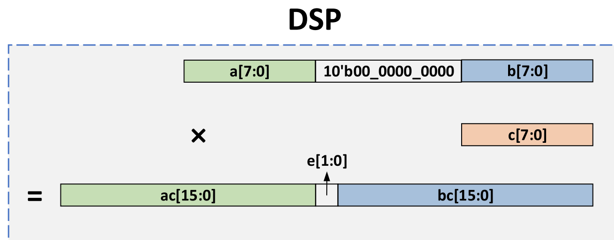

FIGURE 11. DSP optimized design.

operation, and replace the second comparator with an adder to achieve average pooling.

E. UPSAMPLE MODULE DESIGN

The Upsample module is mainly used to expand the input feature map, which can increase the sampling rate and the frequency resolution to facilitate subsequent filtering operations. There are various operations of upsampling, and this design uses the interpolation method of nearest-neighbor interpolation. In contrast to the Maxpool module which takes the maximum value along the aspect of the 2×2 grid as the output, the Upsample module copies a value as four values along the aspect of the 2×2 as the output. The module design architecture is shown in Fig.10.

The data in AXI4-Stream format first enters the Buffer_1 of the Upsample module, and Buffer_1 writes the data to four buffers at the same time, thus realizing the copying of one data into four copies. Next, a complete channel data is first read from Buffer_2 and written to Buffer_6 via the multiplexer. After writing, a complete channel data is then read from Buffer_3 and written to Buffer_6, which converts the data into AXI4-Stream format for output. Then we read Buffer_2 and Buffer_3 back and forth until all the data in a row has been read. We read Buffer_4 and Buffer_5 in the same way, and read Buffer_2 and Buffer_3 again after all the data in a row is read. We repeat this cycle until all the data are executed and we get all the outputs of the Upsample module.

F. DSP OPTIMIZED DESIGN

In convolutional neural networks, the convolutional operations are mainly implemented by a series of multiplication and addition operations. When the inference acceleration of the convolutional neural network is performed on the Xilinx FPGA devices, these massive multiplication and addition operations are all performed by the DSP blocks. However, the DSP resources are very scarce, so we need to make full use of every DSP resource. The DSP48E2 used in this design can implement up to 18×27-bit multiplication and up to 48-bit accumulation operations in one clock cycle, while the convolution calculation in this design is 8×8 bits multiplication. If it is not optimized, the multiplication of 8×8 bits will occupy one DSP block, which will cause a large degree of waste of DSP resources.

According to the optimization scheme proposed in [18], this design uses a single DSP block to perform two 8×8-bit multiplication operations and subsequent accumulation operations simultaneously. In this design, two feature maps are input for calculation at the same time, namely $(a + b)c = ac + bc$. As shown in Fig.11, $a[7 : 0]$ is the data of a feature map, and $b[7 : 0]$ is the data of the same dimension of another feature map, both of which are multiplied with the same weight data $c[7 : 0]$. In the result of multiplication calculation, both $e[0]$ and $e[1]$ are equal to $bc[15]$, which indicates the sign bit of bc. The obtained $bc[15 : 0]$ is the result of bc, while the result of ac needs to be adjusted by adding the sign bit of the result of bc because it may be affected by the low bit feed, i.e. $ac = ac[15 : 0] + e[1]$. The result obtained by the multiplication is then accumulated.

The optimized DSP block can perform two 8×8-bit multiplication operations and subsequent accumulation operations simultaneously, saving half of the DSP resources by performing the same convolution operation.

V. INSTRUCTION GROUP AND HOST COMPUTER SCHEDULING

A. INSTRUCTION GROUP

In this design, the execution of the convolutional neural network accelerator is controlled by instructions, each of which is 32 bits in length and contains different instruction registers for controlling different operations. The host computer realizes the scheduling of the internal operators of the accelerator through configuration instructions.

The host computer sends instructions to the instruction group through the PCIe interface. The instruction group parses each instruction to obtain each instruction register and then sends these instruction registers to the corresponding operator module through SWITCH to decide which operator to perform and how to perform the computation.

Table 2 lists the instructions required for the calculation of the Conv2d module. The four instructions Control, State, Addr, and Num are used to accurately send the weights and input feature maps to the Conv2d calculation module. REG1, REG2, and REG3 are used to control the dimensionality as well as the mode of the convolution calculation, including the length and width of the input feature map, the number of channels, whether to perform Padding, whether to perform Stride, and so on. The accelerator performs the computation methodically according to the order of the instructions until the last instruction is executed, and then the accelerator stops working.

TABLE 2. Conv2d instruction group.

address_offset	register	address_mapping	width	instruction_register
0x00	Control	[15:12]	4	Switch_CNN
		[11:8]	4	Switch_Reshape
		[7:4]	4	Control_Reshape
		[3:0]	4	Control_Conv2d
0x04	State	[7:4]	4	State_Reshape
		[3:0]	4	State_Conv2d
0x08	Addr	[31:0]	32	DMA_Read_Write_Addr
0x0c	Num	[31:0]	32	DMA_Read_Write_Num
0x10	REG1	[31:24]	8	Bias_Num_REG
		[23:8]	16	Weight_Num_REG
		[1:1]	1	CONV_11_Parallel_REG
		[0:0]	1	CONV_11_REG
0x14	REG2	[30:21]	10	Channel_In_Num_REG
		[20:11]	10	Channel_Out_Num_REG
		[10:0]	11	Row_Num_REG
0x18	REG3	[31:21]	11	Column_Num_REG
		[19:19]	1	Leaky_ReLU_REG
		[18:18]	1	Channel_Fill_REG
		[17:17]	1	Padding_REG
		[16:16]	1	Stride_REG
		[15:8]	8	Padding_Num_REG
		[7:0]	8	Zero_Point_REG

B. HOST COMPUTER SCHEDULING

The host computer is realized by the CPU, which mainly includes three parts: pre-processing, task scheduling, and post-processing.

In the pre-processing stage, first, we obtain the input images, perform undistorted resize on the images, and convert them to the 416 × 416 size we need. Then we perform a simple quantization operation on these images and convert the quantized results into the binary files needed for the accelerator calculation.

Task scheduling is to schedule the three accelerators inside the VU9P board, which sends feature maps, weights, and instructions to the specified accelerator, and writes the final calculated prediction results back to the host computer. In this design, our three accelerators are all used to perform the same computation, so they share the same weights and instructions. First, we write the weights and instructions to each of the three accelerators. Then we define a startup function for each accelerator, which is mainly responsible for sending feature maps to the corresponding accelerator and enabling this accelerator. We pass each function object to a thread, and by multi-threading, the three accelerators are started simultaneously. At the same time, these startup functions are constantly listening for information about the computation status of these accelerators and write the computation results back to the host computer as soon as they receive the signal that the computation is complete.

The next step is the post-processing operation. Since the prediction results produced by the above calculation do not match the final predicted box position on the image, decoding is needed to complete it. Decoding is to parse the prediction results, get the important parameters and adjust them to get the center, length, and width of the prediction box. This gives the exact position of the entire prediction box. Many prediction results may be obtained by decoding, so score sorting and NMS filtering are needed to take out the box and score with the largest score in each category. Since each accelerator performs the computation of two images simultaneously, the final detection head generated by each accelerator also contains the prediction results of two images, while both decoding and NMS are computed for one image. Therefore, the first step of the post-processing operation is to split the detection head containing the prediction results of two images. We define a split function to perform the split operation. We create three threads for the split function, pass the detection heads of each of the three accelerators as parameters, and then start a multi-threading. In this way, the prediction results of the six images are obtained simultaneously. In the same way, we perform multi-threading decoding and NMS operations on the prediction results of the six images, and then the final prediction boxs of all six images are obtained.

VI. EXPERIMENT AND RESULTS

A. EXPERIMENTAL ENVIRONMENT AND SETUP

This experiment adopts a CPU+FPGA design scheme to accelerate the target detection algorithm YOLOv4-tiny, and the data interaction is performed through PCIe. The dataset used is the PASCAL VOC dataset. The dataset has a total of 20 categories, where 16551 images from VOC2007 and VOC2012 are used for training and validation, and 4952 images from VOC2007 are used for testing. The CPU adopts Intel core i7-4770 processor, and the FPGA adopts the FX609QL accelerator card from Hangzhou Flying Chip Technology, based on the VU9P chip design of the Xilinx Virtex UltraScale+ series. VU9P has rich on-chip resources, including 6840 DSPs, 2160 BRAMs, 960 URAMs, etc., evenly distributed in three SLRs.

In the experiment, each SLR is designed as a separate chip, while maintaining efficient communication between each SLR, achieving full utilization of the board resources. In addition, the experiments also use only a single SLR on the VU9P board for a single accelerator design to make a fairer comparison in terms of resources. The single accelerator solution can be deployed on other FPGA boards with fewer resources. This design is developed using Verilog HDL, synthesized and placed, and routed through Vivado development tools.

The operation and parameter configuration of each layer of YOLOv4-tiny are shown in Table 3 and Table 4, where Table 4 is the Resblock module in Table 3. The structure of

TABLE 3. Parameter configuration of the YOLOv4-tiny model used in this paper.

Layer	Type	Input	Filters	Size/Stride	Output
1	Conv2d	416×416×3	32	3×3/2	208×208×32
2	Conv2d	208×208×32	64	3×3/2	104×104×64
3	Resblock1	104×104×64			52×52×128
4	Resblock2	52×52×128			26×26×256
5	Resblock3	26×26×256			13×13×512
6	Conv2d	13×13×512	512	3×3/1	13×13×512
7	Conv2d	13×13×512	256	1×1/1	13×13×256
8	Conv2d	13×13×256	128	1×1/1	13×13×128
9	Upsample	13×13×128			26×26×128
10	Concat	26×26×256 26×26×128			26×26×384
11	Conv2d	26×26×384	256	3×3/1	26×26×256
12	Conv2d	26×26×256	75	1×1/1	26×26×75
13	Conv2d	13×13×256	512	3×3/1	13×13×512
14	Conv2d	13×13×512	75	1×1/1	13×13×75

TABLE 4. Parameter configuration of the Resblock module.

Layer	Type	Input	Filters	Size/Stride	Output
1	Conv2d	H×W×C	C	3×3/1	H×W×C
2	Split	H×W×C			H×W×(C/2)
3	Conv2d	H×W×(C/2)	(C/2)	3×3/1	H×W×(C/2)
4	Conv2d	H×W×(C/2)	(C/2)	3×3/1	H×W×(C/2)
5	Concat	H×W×(C/2) H×W×(C/2)			H×W×C
6	Conv2d	H×W×C	C	1×1/1	H×W×C
7	Concat	H×W×C H×W×C			H×W×(C×2)
8	Maxpool	H×W×(C×2)		2×2/2	(H/2)×(W/2)×(C×2)

the three Resblock modules is the same, and according to the three different input feature maps, the Resblock with three different parameter configurations from Layer3 to Layer5 in Table 3 can be obtained.

Except for Layer1, Layer12, and Layer14, the parameters used in this accelerator for each layer calculation are the same as those listed in Table 3 and Table 4. For Layer1, since the input channel parallelism used in this accelerator is 16, and the input channel of Layer1 is only 3, the number of channels of input feature maps and weights of Layer1 needs to be supplemented to 16 for calculation. In addition, the parallelism of the output channel of this accelerator is 8, while the output channel of Layer12 and Layer14 is 75, which is not a multiple of 8. Therefore, for the convenience of calculation, 80 output channels were calculated for these two layers, and only the valid 75 output channels were taken for calculation in post-processing.

In addition, our single accelerator is performing the computation of two images at the same time, so in the configuration parameters of the accelerator, we also have to add the dimension of the image as 2.

TABLE 5. Performance comparison of different devices.

	i7-11800H	RTX3060	Jetson nano	VU9P (Single)
Platform Type	CPU	GPU	GPU	FPGA
Frequency	2.30GHz	1.32GHz	922MHz	200MHz
Precision	FP32	FP32	FP32	INT8
FPS	17.7	161	13.2	49.38
Power	45W	170W	12.2W	12.689W
mAP	83.99%	83.99%	83.99%	81.54%
Total Clock Cycle	129.9M	8.2M	69.8M	4.7M

B. EXPERIMENTAL RESULTS ANALYSIS

According to the parameters used by the accelerator described in Section VI-A, we can calculate the memory access of the entire model, including the accesses to the weights, the accesses to the quantization parameters M, B, and Z, and the accesses to the feature maps. The accesses to the weights and quantization parameters in the off-chip DDR occur only in the Conv2d module, and we maximize the reuse of the weights and quantization parameters by reading all the weights and quantization parameters from the DDR only once, so we can calculate the accesses to the DDR for the weights and quantization parameters to be 5.68 MB. The accesses to the feature maps include the accesses to the DDR for the first two layers and the last layer and the accesses to the on-chip URAM for all other layers. We calculate the number of output feature maps for all layers to get the access memory of the feature map calculated for one image. Since we calculate two images at the same time, the final access memory of the feature map is 21.71 MB. Therefore, our accelerator performs a complete computation with 27.39 MB of access memory.

In our accelerator, each DSP block implements two multiply-accumulate operations simultaneously, and our accelerator runs at a clock frequency of 200 MHz. Therefore, the peak throughput of a single accelerator is $2 \times P_i \times P_o \times P_p \times 2 \times Freq = 921.6$ GOPS. And our multi-accelerator scheme is composed of three single accelerators, and these three accelerators are executed in parallel, so the peak throughput of the multi-accelerator scheme is 2764.8 GOPS.

In addition, we measured the computation time of the single accelerator on the VU9P board to be 40.5 ms, and since it computes two images simultaneously, the FPS is 49.38. Similarly, we obtained an FPS of 148.14 for the multi-accelerator scheme.

C. COMPARISON OF EXPERIMENTAL RESULTS

YOLOv4-tiny is implemented by many operators. This design implements all operators of the YOLOv4-tiny algorithm and implements the inference of YOLOv4-tiny through instruction control.

Table 5 shows the performance of the YOLOv4-tiny algorithm for object detection on four different hardware platforms. As can be seen from the table, the FPGA uses the

TABLE 6. Compare results with similar designs.

	ICM 2021 [20]	IEEE Access 2021 [21]	INDICON 2021 [22]	This Work (Single)	This Work (Multi)
Platform	Virtex UltraScale+ VCU118	Kintex UltraScale XCKU040	Virtex UltraScale+ VU9P	Virtex UltraScale+ VU9P	Virtex UltraScale+ VU9P
Target Network	YOLOv4-Tiny	YOLOv4-Tiny	YOLOv3-Tiny	YOLOv4-Tiny	YOLOv4-Tiny
Freq(MHz)	100	143	200	200	200
Image Size	-	416×416	416×416	416×416	416×416
Precision(W,F)	(10,13)	(16,16)	(32,32)	(8,8)	(8,8)
BRAM(36Kb)	1888	447.5	-	454	1262
URAM(288Kb)	0	0	0	128	384
DSP	5741	1255	2693	1486	4458
LUT	750k	182k	177k	156k	422k
FF	189K	149k	146k	274k	770k
Throughput(GOPS)	-	357	166.4	921.6	2764.8
FPS	3.33	31.2	32.1	49.38	148.14
Power(W)	19.5	-	-	12.689	29.68

INT8 format for computation, and its accuracy loss is within an acceptable 3% range compared to the FP32, but it can deliver several times the performance improvement at very low power consumption. Although the RTX3060's FPS is much greater than that of the FPGA, its extremely high power consumption makes it impossible to be used at the edge. Compared to the Jetson nano, an edge-side GPU device, the FPGA delivers as much as three times the FPS with a small difference in power consumption. In addition, the experiment also compares the computational performance of accelerators by calculating the number of clock cycles required for an inference process, even if their clock frequencies are different due to the influence of platform choice. In this experiment, fewer cycles mean better architecture [19]. It can be seen that the accelerator we designed on the VU9P platform has the least number of clock cycles, which indicates our accelerator performs better.

Table 6 compares the single-accelerator and multi-accelerator schemes of this design with the currently existing accelerators related to the YOLOv4-tiny algorithm. The accelerator proposed in [20] uses far more resources than the single accelerator of this design, and even more resources than the multi-accelerator, but its FPS is only about 1/15 of the single accelerator and about 1/45 of the multi-accelerator. The resources used by the accelerator proposed in [21] are not much different from that of a single accelerator, but our single accelerator has 2.6 times the throughput and higher FPS. This performance is inferior to that of a single accelerator mainly for two reasons: one is that its operating frequency is too low; the other is that it cannot make full use of DSP because it uses 16-bit quantization for calculation. The accelerator proposed in [22] uses a lot of DSP resources, but its throughput is less than 1/5 of that of a single accelerator, and its FPS is also lower than that of a single accelerator. In addition, the multi-accelerator is composed of three single accelerators and they are executed in parallel. Although the power consumption is higher, the performance is improved by three times and the energy efficiency ratio reaches 93.15 GOPS/W.

VII. CONCLUSION

In this paper, a multi-die-based convolutional neural network accelerator is designed to accelerate the YOLOv4-tiny algorithm. The accelerator makes full use of on-chip resources, deploys an accelerator in each SLR, and controls the three accelerators through the host computer. By quantizing the input feature map and weight to 8 bits, the calculation pressure of the hardware is reduced, and the DSP block is optimized at the same time. Two 8-bit calculations are performed per clock cycle, which improves the efficiency of the DSP block. In the calculation process, the design fully reuses the feature maps and weights, and writes the intermediate results of each layer to the on-chip buffer, which eliminates the off-chip access of intermediate data, reduces the pressure on bandwidth, and reduces power consumption. Furthermore, an instruction group is designed, and the host computer realizes the control of the accelerator through the configuration of the instruction group.

Tests have verified that the accelerator can achieve a frame rate of 148.14 FPS and a peak throughput of 2.76 TOPS on the YOLOv4-tiny algorithm, and the energy efficiency ratio reaches 93.15 GOPS/W, which has achieved good results in the field of real-time target detection.

REFERENCES

[1] L. Jiao, F. Zhang, F. Liu, S. Yang, L. Li, Z. Feng, and R. Qu, ''A survey of deep learning-based object detection,'' *IEEE Access*, vol. 7, pp. 128837–128868, 2019.

[2] Y. Liu, Z. Ma, X. Liu, S. Ma, and K. Ren, ''Privacy-preserving object detection for medical images with faster R-CNN,'' *IEEE Trans. Inf. Forensics Security*, vol. 17, pp. 69–84, 2022.

[3] M. Zhu and Q. Chen, ''Big data image classification based on distributed deep representation learning model,'' *IEEE Access*, vol. 8, pp. 133890–133904, 2020.

[4] E. Basaran, Z. Cömert, and Y. Çelik, ''Convolutional neural network approach for automatic tympanic membrane detection and classification,'' *Biomed. Signal Process. Control*, vol. 56, Feb. 2020, Art. no. 101734.

[5] P. Yin, R. Yuan, Y. Cheng, and Q. Wu, ''Deep guidance network for biomedical image segmentation,'' *IEEE Access*, vol. 8, pp. 116106–116116, 2020.

[6] P. Gong, W. Yu, Q. Sun, R. Zhao, and J. Hu, ''Unsupervised domain adaptation network with category-centric prototype aligner for biomedical image segmentation,'' *IEEE Access*, vol. 9, pp. 36500–36511, 2021.

[7] S. McCann, H. H. Lee, G. Refai-Ahmed, T. Lee, and S. Ramalingam, "Warpage and reliability challenges for stacked silicon interconnect technology in large packages," in *Proc. IEEE 68th Electron. Compon. Technol. Conf. (ECTC)*, May 2018, pp. 2345–2350.

[8] J. Pang, K. Chen, J. Shi, H. Feng, W. Ouyang, and D. Lin, "Libra R-CNN: Towards balanced learning for object detection," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2019, pp. 821–830.

[9] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards real-time object detection with region proposal networks," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 39, no. 6, pp. 1137–1149, Jun. 2017.

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

[11] X. Sun, T. Liu, X. Yu, and B. Pang, "Unmanned surface vessel visual object detection under all-weather conditions with optimized feature fusion network in YOLOv4," *J. Intell. Robotic Syst.*, vol. 103, no. 3, pp. 1–16, Nov. 2021.

[12] N. Murad, M.-C. Pan, and Y.-F. Hsu, "Reconstruction and localization of tumors in breast optical imaging via convolution neural network based on batch normalization layers," *IEEE Access*, vol. 10, pp. 57850–57864, 2022.

[13] S. Mittal, "A survey of FPGA-based accelerators for convolutional neural networks," *Neural Comput. Appl.*, vol. 32, no. 4, pp. 1109–1139, Feb. 2020.

[14] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2018, pp. 2704–2713.

[15] L. Sun, M.-H. Chen, L. Zhang, P. He, and L.-S. Xie, "Recent progress in SLID bonding in novel 3D-IC technologies," *J. Alloys Compounds*, vol. 818, Mar. 2020, Art. no. 152825.

[16] G. Zhang, K. Zhao, B. Wu, Y. Sun, L. Sun, and F. Liang, "A RISC-V based hardware accelerator designed for Yolo object detection system," in *Proc. IEEE Int. Conf. Intell. Appl. Syst. Eng. (ICIASE)*, Apr. 2019, pp. 9–11.

[17] D. T. Nguyen, T. N. Nguyen, H. Kim, and H. J. Lee, "A high-throughput and power-efficient FPGA implementation of YOLO CNN for object detection," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 27, no. 8, pp. 1861–1873, Aug. 2019.

[18] Y. Fu, E. Wu, A. Sirasao, S. Attia, K. Khan, and R. Wittig, "Deep learning with int8 optimization on Xilinx devices," Xilinx, San Jose, CA, USA, White Paper WP486, 2016.

[19] T. Adiono, A. Putra, N. Sutisna, I. Syafalni, and R. Mulyawan, "Low latency YOLOv3-tiny accelerator for low-cost FPGA using general matrix multiplication principle," *IEEE Access*, vol. 9, pp. 141890–141913, 2021.

[20] O. Eid and M. A. A. El Ghany, "Hardware implementation of Yolov4-tiny for object detection," in *Proc. Int. Conf. Microelectron. (ICM)*, Dec. 2021, pp. 270–275.

[21] D. Pestana, P. R. Miranda, J. D. Lopes, R. P. Duarte, M. P. Vestias, H. C. Neto, and J. T. D. Sousa, "A full featured configurable accelerator for object detection with Yolo," *IEEE Access*, vol. 9, pp. 75864–75877, 2021.

[22] M. Sharma, R. Rahul, S. Madhusudan, S. P. Deepu, and D. S. Sumam, "Hardware accelerator for object detection using tiny YOLO-v3," in *Proc. IEEE 18th India Council Int. Conf. (INDICON)*, Dec. 2021, pp. 1–6.

QINGZENG SONG was born in Hebei, China, in 1980. He received the B.S. degree in information and computing sciences, the M.S. degree in applied mathematics, and the Ph.D. degree in electrical engineering from the Hebei University of Technology. From 2012 to 2017, he was a Lecturer with the School of Computer Science and Software Engineering, Tiangong University. Since 2018, he has been an Associate Professor with the School of Computer Science and Technology, Tiangong University. His research interests include artificial intelligence, embedded systems, and IC design.

JIABING ZHANG was born in Henan, China, in 1996. He received the B.S. degree from the Luoyang Institute of Science and Technology, Henan, in 2020. He is currently pursuing the M.S. degree with the School of Computer Science and Technology, Tiangong University. His research interests include edge computing and FPGA acceleration.

LIANKUN SUN was born in Tianjin, China, in 1979. He received the M.S. degree in control theory and control engineering from Tiangong University, Tianjin, in 2005, and the Ph.D. degree in control theory and control engineering from Tianjin University, Tianjin, in 2009. In March 2009, he joined Tiangong University, where he is currently an Associate Professor. His research interests include the analysis and synthesis of networked control systems and deep learning.

GUANGHAO JIN (Member, IEEE) was born in Jilin, China, in 1979. He received the B.S. degree in applied mathematics from Peking University, in 2002, the M.S. degree in computer software and theory from the China Academy of Engineering Physics, in 2005, and the Ph.D. degree in mathematics and computing science from the Tokyo Institute of Technology, in 2014. From 2014 to 2015, he worked as a Postdoctoral Researcher with the Tokyo Institute of Technology. From 2016 to 2020, he was a Lecturer at Tiangong University. Since 2021, he has been working as a Lecturer with Beijing Polytechnic. In artificial intelligence-related fields, he has published 25 papers in domestic and foreign journals and conferences.