electronics

IMPACT FACTOR 2.9

CITESCORE 4.7

Article

Editor's Choice

FPGA Implementation for CNN-Based Optical Remote Sensing Object Detection

Ning Zhang, Xin Wei, He Chen and Wenchao Liu

MDPI

https://doi.org/10.3390/electronics10030282

electronics

MDPI

Article

FPGA Implementation for CNN-Based Optical Remote Sensing Object Detection

Ning Zhang [1], Xin Wei [1], He Chen [1] and Wenchao Liu [2,*]

[1] Beijing Key Laboratory of Embedded Real-Time Information Processing Technology, Beijing Institute of Technology, Beijing 100081, China; 3120205375@bit.edu.cn (N.Z.); weixin@bit.edu.cn (X.W.); chenhe@bit.edu.cn (H.C.)

[2] Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China

* Correspondence: liuwenchao@mail.tsinghua.edu.cn; Tel.: +86-1521-051-4721

Abstract: In recent years, convolutional neural network (CNN)-based methods have been widely used for optical remote sensing object detection and have shown excellent performance. Some aerospace systems, such as satellites or aircrafts, need to adopt these methods to observe objects on the ground. Due to the limited budget of the logical resources and power consumption in these systems, an embedded device is a good choice to implement the CNN-based methods. However, it is still a challenge to strike a balance between performance and power consumption. In this paper, we propose an efficient hardware-implementation method for optical remote sensing object detection. Firstly, we optimize the CNN-based model for hardware implementation, which establishes a foundation for efficiently mapping the network on a field-programmable gate array (FPGA). In addition, we propose a hardware architecture for the CNN-based remote sensing object detection model. In this architecture, a general processing engine (PE) is proposed to implement multiple types of convolutions in the network using the uniform module. An efficient data storage and access scheme is also proposed, and it achieves low-latency calculations and a high memory bandwidth utilization rate. Finally, we deployed the improved YOLOv2 network on a Xilinx ZYNQ xc7z035 FPGA to evaluate the performance of our design. The experimental results show that the performance of our implementation on an FPGA is only 0.18% lower than that on a graphics processing unit (GPU) in mean average precision (mAP). Under a 200 MHz working frequency, our design achieves a throughput of 111.5 giga-operations per second (GOP/s) with a 5.96 W on-chip power consumption. Comparison with the related works demonstrates that the proposed design has obvious advantages in terms of energy efficiency and that it is suitable for deployment on embedded devices.

Keywords: object detection; remote sensing; deep learning; CNN; hardware implementation; FPGA; you-only-look-once (YOLO)

Citation: Zhang, N.; Wei, X.; Chen, H.; Liu, W. FPGA Implementation for CNN-Based Optical Remote Sensing Object Detection. *Electronics* **2021**, *10*, 282. https://doi.org/10.3390/electronics10030282

Academic Editor: Dah-Jye Lee
Received: 19 December 2020
Accepted: 20 January 2021
Published: 25 January 2021

1. Introduction

Object detection is an important research topic of remote sensing image processing. Object detection of optical remote sensing images aims to predict the location of the objects belonging to the category of interest in a given remote sensing image [1]. Drawing upon recent advances in computer vision, many researchers have adopted CNN-based methods to remote sensing object detection applications, such as environmental monitoring [2], intelligent transportation [3], and other vital applications [4–7]. Traditional remote sensing image processing systems need to download images to the ground station for processing and analysis from satellites. However, with the development of remote sensing technology, the resolution and the data amount of optical remote sensing images are constantly increasing. The increasing volume of image data puts high pressure on the data downlink [8,9]. The processing delay of the systems may be too long to meet the requirements of timeliness [10,11]. Thus, many works have constructed an onboard remote sensing system to

implement object detection in real time on the satellites or aircraft [8,12]. This solution has been shown to be more efficient than the traditional solutions [12].

The CNN-based object detection requires a high volume of parameters and calculations to extract features in the image and make predictions about the objects [13–15]. To meet this requirement, many researchers adopt high-performance devices, such as graphics processing units (GPUs), central processing units (CPUs), field-programmable gate arrays (FPGAs), and application-specific integrated circuits (ASICs), to build onboard real-time systems [14,16]. It is difficult for CPUs to take full advantage of parallel computing to meet real-time processing [17]. They are rarely used as CNN implementation platforms. While the computing performance of GPUs is fantastic, the high-power consumption hinders their usage in the onboard system with limited resources and power budgets [12,18]. FPGAs and ASICs have the advantages of high performance and energy efficiency [16,19]. Thus, taking these low-power devices as hardware implementation platforms for CNNs has become a research hotspot. However, ASICs require a long development period and high costs to be designed. Therefore, owing to the advantages of the short development period, energy efficiency, and reconfigurability, FPGAs are the ideal implementation platforms for CNNs [17].

In recent years, researchers have proposed a variety of hardware-implementation methods for CNN-based models on FPGAs [20,21]. While these works have successfully implemented the inference phase of CNNs on FPGA, some challenges remain. The first challenge is how to rational design a parallel implementation scheme for CNNs on an FPGA. The second challenge is how to reduce the model complexities of CNNs for hardware implementation while ensuring implementation accuracy.

Some researchers focus on increasing the parallelism and pipeline of their designs. Peemen et al. [22] proposed a novel data access pattern and on-chip buffer management to improve parallelism and computational performance. However, the design needs about 10 s to program the FPGA to shift to the next layer. Thus, processing speed in this design cannot meet the requirement of real-time processing. F. Sun et al. [23] proposed an FPGA-based accelerator composed of multiple processing engines (PEs) and implemented an AlexNet on it. Each PE is used to achieve the computation of one layer in the model. This ensures that the calculations in different layers can be carried out in the pipeline. Similarly, H. Li et al. [24] proposed an end-to-end CNN accelerator, which enables all layers of the AlexNet to work concurrently in the pipeline structure. These designs increased parallelism, which can improve performance significantly.

L. Chen et al. [17] implemented an LeNet-5 on an FPGA for remote sensing image classification. It took 2.29 ms to process a remote sensing image with a size of 126 $\times$ 126. This work adopted full-precision data types to represent the models and features, which needed a high amount of hardware resources and thus are difficult to use for the implementation of complex models. Therefore, extensive studies have made efforts to reduce the model complexities of CNNs to efficiently deploy them on FPGAs. Y. Zhou et al. [25] implemented a 5-layer CNN on an FPGA with 11-bit fixed-point precision using the Xilinx HLS tool. Z. Li et al. [26] presented an 8-bit fixed-point inference engine (Laius) for LeNet-5 on an FPGA. The test results on the MNIST dataset show that the accuracy loss of this implementation exceeds 1%, compared to the original Caffe model. H. Fan et al. [27] proposed an architecture that accelerates an SSDLite-MobileNetV2 object detector on an FPGA. To meet the requirements of real-time processing, this work adopted partial network quantization for hardware optimization. The weights of the convolutional layer were quantized into 8-bit fixed. However, this design has a 1.8% accuracy loss on the COCO dataset. Some researchers are devoted to exploring extreme low-bit compression for hardware implementation [28,29]. J. Li et al. [30] implemented a CNN on an FPGA, where the weights were constrained to only two possible values. In summary, the above-mentioned works improved the calculation efficiency of their designs through different quantization methods.

However, there are still some shortcomings in the hardware implementation. First, while it can increase the pipeline and parallel of the implementation by designing a special architecture for each layer in the network, they are only customized for a specific network structure and cannot implement others. These methods are difficult to use for the hardware implementation of complex networks due to the massive resource overhead. Second, some low-precision quantization strategies are not hardware-friendly, such as the 11-bit fixed-point quantization [25]. Moreover, the low-precision implementation of a CNN will inevitably cause performance degradation, especially binary networks. These problems are more serious in remote sensing image processing than that in the natural scenes [12]. Up to now, our search of the literature shows that few studies are devoted to implement CNN-based object detection on the FPGA for remote sensing. This is still a challenging task and research hotspot.

In this paper, we propose an efficient hardware-implementation method for optical remote sensing object detection. This method enables a CNN-based object detection network to be successfully deployed in an FPGA with a low cost of power consumption and hardware resources. The main contributions of this paper are summarized as follows:

- We optimized an improved YOLOv2 network for hardware implementation. The optimization mainly includes three aspects: network quantization, layer fusion, and a unified implementation of multiple convolutions. Through these optimization methods, we effectively reduced the scale of the network while maintaining detection accuracy.
- We proposed a hardware architecture for the CNN-based remote sensing object detection model. In this architecture, a general PE is proposed to implement multiple types of convolutions in the network. An efficient data storage and access scheme is also proposed, which achieves low-latency calculations and a high memory bandwidth utilization rate.
- We implemented the optimized network on the Xilinx ZYNQ xc7z035 FPGA to evaluate the performance of the proposed hardware architecture. The experimental results tested on the detection in an aerial image (DOTA) [31] dataset illustrate that the performance of our implementation on an FPGA is only 0.18% lower than that on a GPU in mean average precision (mAP). A comparison with related works demonstrates that our design can strike an excellent balance between resource consumption and computing time cost.

2. Background

Several CNN-based methods, Region-CNNs (R-CNNs) [32], the Single Shot MultiBox Detector (SSD) [33], and you-only-look-once (YOLO) [34], have been proposed for object detection. YOLO ensures an excellent trade-off between accuracy and speed compared with other approaches [29,35]. In Reference [4], an improved YOLOv2 network was proposed for remote sensing object detection. This network adopted dilated convolution and transposed convolution to improve performance for multiscale objects in complex optical remote sensing scenes. This network strikes a balance between model complexity and object detection performance. In this paper, we took the improved YOLOv2 as the fundamental network. The structure of the fundamental work is shown in Figure 1. As shown in Figure 1, the fundamental work contains multiple computational layers. These layers are concatenated together. The main layers are the convolutional layer, pooling layer, batch normalization layer, and activation function.

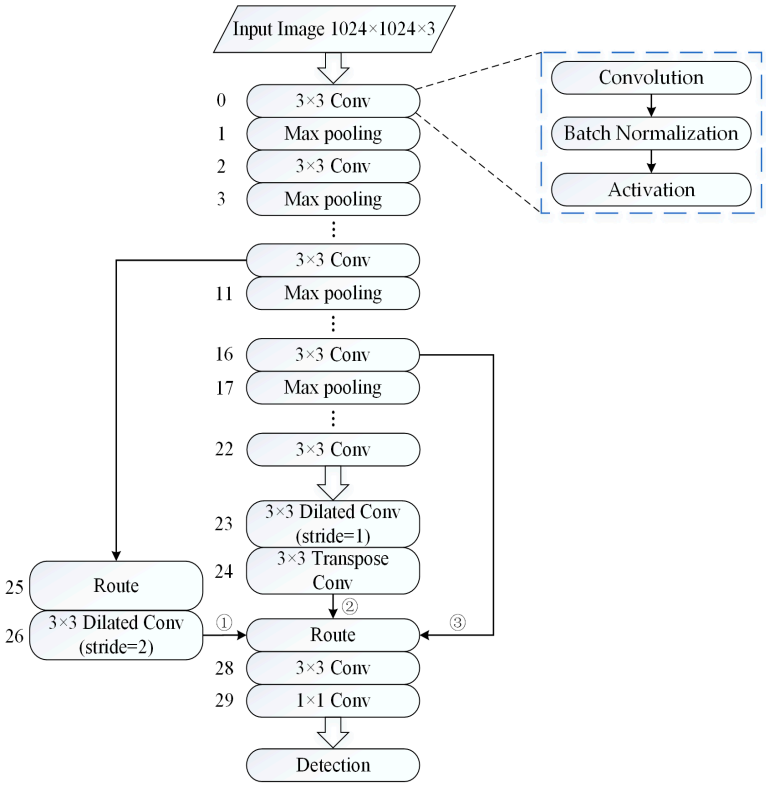


Figure 1. The structure of the improved YOLOv2. Each convolutional layer contains not only convolution but also batch normalization and activation sublayers.

2.1. Convolutional Layer

In CNNs, convolutional layers are used to extract features from input images. The operation in a convolutional layer is a three-dimensional calculation by input feature maps and convolution kernels. The fundamental network has three types of these operations: a standard convolution, a dilated convolution, and a transposed convolution. The details are described in the following subsections.

2.1.1. Standard Convolution

The calculation of standard convolution is defined as

$$O_{x,y} = \sum_{c=0}^{C_{in}-1} \sum_{n=0}^{K-1} \sum_{m=0}^{K-1} I_{x+s\times m, y+s\times n} \times w_{m,n} + b_{x,y} \quad (1)$$

where $I_{x+s\times m, y+s\times n}$ is the value of the input feature map at the point of $(x + s \times m, y + s \times n)$, C_{in} is the number of input channels, s represents the stride of the convolutional layer, $w_{m,n}$ is the corresponding weight in the kernels, K is the size of kernels, $b_{x,y}$ is the corresponding bias, and $O_{x,y}$ is the value of the output feature map at the point of (x, y).

2.1.2. Dilated Convolution

A dilated convolution can expand the receptive field of the feature map without increasing parameters [36]. It is mainly used to introduce fine features and avoid excessive loss of the resolution. The actual size of the kernel can be computed with

$$k_r = k + (k-1) \times (r-1) \quad (2)$$

where r is a hyper-parameter that represents the dilation rate. Taking the dilated convolution with $r = 2$ in the fundamental network as an example, its computation is shown in

Figure 2. A zero is inserted between every two weights in the original kernel. The size of the kernel is changed from 3 × 3 to 5 × 5 after interpolation.

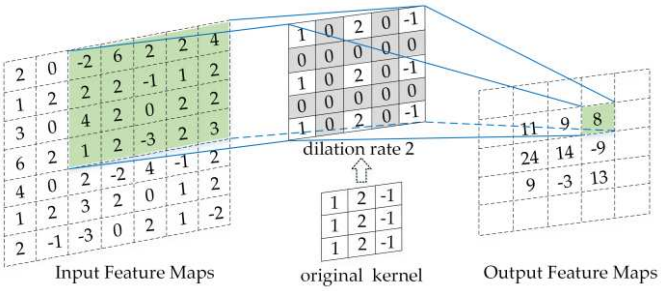


Figure 2. The computation of the dilated convolution with a 3 × 3 kernel and a dilation rate of 2.

2.1.3. Transposed Convolution

A transposed convolution is another type of convolutional operation in CNNs. This convolution can be used to achieve the up-sampling of the feature maps, which can be defined as

$$f_{out} = f_{in} \times k^T \tag{3}$$

where k^T represents the weight matrix after transpose, and f_{in} and f_{out} represent the input and output feature maps, respectively. The essence of the transposed convolution is an operation expanding the size of the input feature map. The size of the input feature map after interpolated can be expressed as follows:

$$\begin{aligned} H'_{in} &= H_{in} \times s + 2 \times pad_{in} + pad_{out} \\ W'_{in} &= W_{in} \times s + 2 \times pad_{in} + pad_{out} \end{aligned} \tag{4}$$

where s is the stride of the convolution, pad_{in} is the amount of zero-padding to both sides of the input, pad_{out} is the amount of zero-padding to one side of the output, and H_{in} and W_{in} represent the height and width of the original input feature map, respectively. H'_{in} and W'_{in} represent the height and width of the input feature map after interpolation, respectively. For the fundamental network, pad_{in} and pad_{out} are both 1 and the stride is equal to 2. The computation of the transposed convolution is shown in Figure 3.

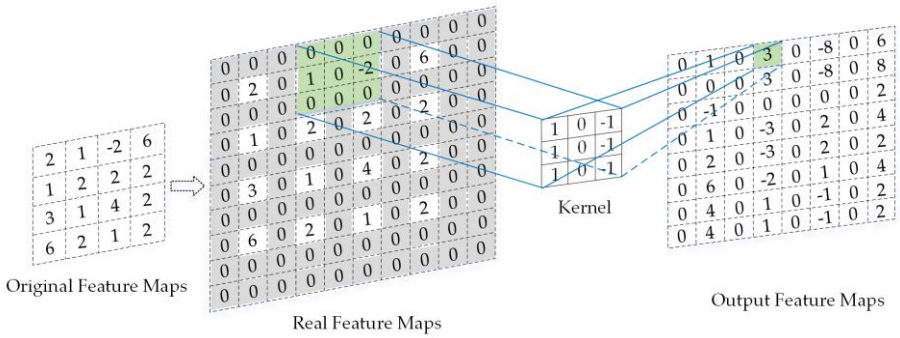


Figure 3. The computation of the transposed convolution (pad_{in} = 1, pad_{out} = 1, s = 2).

2.2. Batch-Normalization Layer

In most CNNs, the batch normalization (BN) layer and activation function [37] are placed after the convolutional layers. BN layers are used to prevent over-fitting and speed up the training [38]. The calculation of BN can be defined as follows:

$$BN(O_{x,y}) = \gamma \left(\frac{O_{x,y} - \overline{O}}{\sqrt{\sigma^2 + \varepsilon}} \right) + \beta \tag{5}$$

where $\bar{O}$ and σ^2 represent the average and variance of the output feature maps of the previous convolution layer, respectively. ε is a minimal number to prevent the denominator from being zero. γ and β are learnable parameters to apply an affine transformation to the normalized output feature maps.

2.3. Activation Function

The activation functions are mainly used to change the output feature maps non-linearly. The most commonly used activation functions are the rectified linear unit (ReLU) and Leaky ReLU. LeakyReLU activation is widely used in the fundamental network, which can be defined as

$$y = \begin{cases} x & x \geq 0 \\ ax & x < 0 \end{cases} \tag{6}$$

where a is the fixed leaky coefficient in the range (0, 1). For ReLU activation, it can be computed in the same way by setting $a = 0$.

2.4. Pooling Layer

The pooling layer can reduce the size of feature maps by discarding redundant information. The most commonly used in CNNs are average pooling and max pooling. In the fundamental network, max pooling is used as the pooling layer. The output neuron of the max-pooling layer can be calculated as

$$N_{x,y}^{out} = \underset{i,j \in [0,m-1]}{Max} (N_{x+i,y+j}^{in}) \tag{7}$$

where the max pooling layers take the maximum value from the region $m \times m$ as the output. The height and the width of the pooling size are both 2, and its vertical and horizontal strides are 2 in the fundamental network.

3. Optimization for Implementation

With the above-mentioned descriptions of the computational layers in the fundamental network, we can use a processing block to achieve all the forward calculations during inference. The block is shown in Figure 4. We can directly deploy the block on the hardware. However, this deployment method has several disadvantages. Firstly, the values involved in the calculations are represented at the floating point and are not hardware friendly. Second, the multiplications and additions in the calculations are very dense, which limits the processing speed and has a high resource requirement. In addition, the fundamental network contains multiple types of convolution operations. If we customize the structure for each convolution, it will consume enormous hardware resources. To solve these problems, we optimize the network, as shown in Figure 4. The optimization contains network quantization, layer fusion, and the unified implementation of multiple convolutions. The details are described in the following subsections.

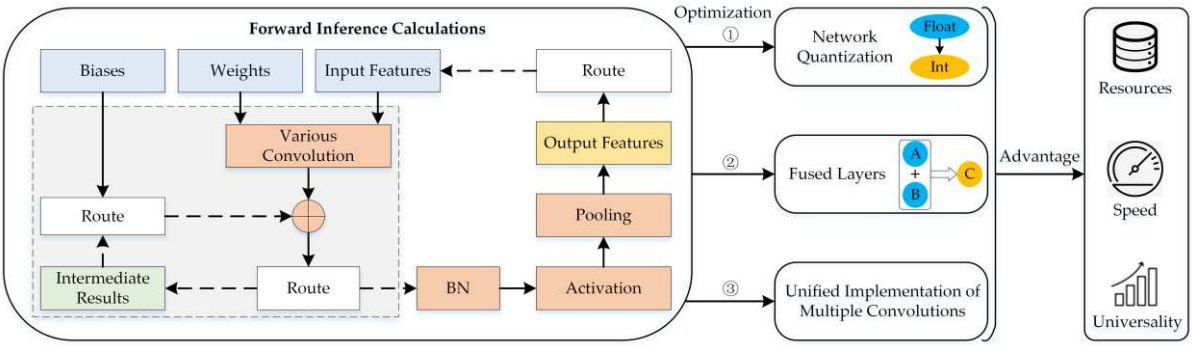


Figure 4. The forward inference calculation process of the fundamental network and an overview of our optimization for implementation.

3.1. Network Quantization

In the fundamental network, the convolutional layers account for most of the calculations. To deploy the network on the FPGA efficiently, we adopted a hardware-awareness symmetric quantization scheme to quantize both the feature maps and weights of the convolutional layers into a low-bit integer. Since other layers require high data accuracy [28,29], we retain them at full precision. Considering the general case of N-bit symmetric quantization, the quantization function is defined by the following:

$$q = clamp\left(Int\left(\frac{r}{S}\right), \left(-2^{N-1}+1\right), \left(2^{N-1}-1\right)\right) \tag{8}$$

where r represents the floating-point element in the input feature matrix or weight matrix, q represents the corresponding quantized value, S indicates the scaling factor of the matrix, and N indicates the quantization bit width. The $clamp(\cdot)$ function is used to limit the quantized values to the range of $\left[\left(-2^{N-1}+1\right), \left(2^{N-1}-1\right)\right]$, and $Int(\cdot)$ function is used to round the data to an integer. The quantized bias can be calculated by the following:

$$b' = clamp\left(Int\left(\frac{b}{S_d \times S_w}\right), \left(-2^{N_b-1}+1\right), \left(2^{N_b-1}-1\right)\right) \tag{9}$$

where b and b' represent the original and quantized bias, respectively. S_w and S_d represent the scaling factor of the feature matrix and weight matrix, respectively. N_b indicates the quantization bit width of the bias matrix. In this paper, the quantization bit widths of the feature maps and weights are set to 8 bits. The quantization bit widths of biases are set to 32 bits. With the quantized feature maps and weights, the convolutional layer can be converted into a quantized version as

$$\hat{O}_{x,y} = \sum_{c=0}^{C_{in}-1} \sum_{n=0}^{K-1} \sum_{m=0}^{K-1} \hat{I}_{x+s\times m, y+s\times n} \times \hat{w}_{m,n} + \hat{b}_{x,y} \tag{10}$$

where $\hat{I}$ and $\hat{w}$ represent the feature values and weights after being quantized to 8-bit fixed-point version, respectively. $\hat{b}$ represents the biases after quantification, and $\hat{O}$ represents the output feature values of the quantized convolutional layer.

The output feature maps of the quantized convolutional layer are 32-bit integers. It is necessary to perform inverse quantization to convert these values into floating-point types for the following layers. The operation of inverse quantization is defined as follows:

$$q = S_d S_w \times q' \tag{11}$$

where q' represents the quantized value in the quantized convolutional results, and q represents the corresponding floating-point value after inverse quantization. With the hardware-awareness symmetric quantization, the volume of floating-point operations in the network can be reduced. Moreover, this hardware-friendly optimization can efficiently reduce the requirement of hardware resources for implementation.

3.2. Layer Fusion

Applying symmetric quantization to the fundamental network eliminates most of the floating-point operations in the convolutional layers. However, there are still dense floating-point operations in the other layers, the quantization layers and the inverse quantization layers. To further reduce the volume of the floating-point calculation in the fundamental network, we present another optimization, layer fusion, to merge the floating-point multiplications and additions in adjacent layers.

In Equation (5), $\bar{O}$, σ^2, γ, and β are determined after the network is trained. With the determined parameters, Equation (5) can be rewritten as

$$BN(O_{x,y}) = \frac{\gamma}{\sqrt{\sigma^2+\varepsilon}} O_{x,y} + \left(\beta - \frac{\gamma\overline{O}}{\sqrt{\sigma^2+\varepsilon}}\right) \tag{12}$$

Compared to Equation (5), Equation (12) has one less addition and multiplication during the inference. This transformation can optimize the hardware design while deploying the BN layer on the FPGA.

Generally, BN layers are placed after convolutional layers. Applying the symmetric quantization will insert an inverse quantization layer between the quantized convolutional layer and BN layer. Considering that the multiplications in the inverse quantization layer and the BN layer can be achieved by only one multiplication, we fused the inverse quantization to the BN layer. The calculation of the fused BN layer is defined as follows:

$$q = \gamma' O_{x,y} + \beta' \tag{13}$$

where γ' and β' represent the multiplication and addition coefficients of the fused BN layer, respectively. They can be calculated by the following:

$$\begin{cases} \gamma' = \frac{\gamma S_d S_w}{\sqrt{\sigma^2+\varepsilon}} \\ \beta' = \beta - \gamma \times \frac{\overline{O}}{\sqrt{\sigma^2+\varepsilon}} \end{cases} \tag{14}$$

These two coefficients can be calculated offline. For hardware implementations, this fused layer only needs to perform one floating-point multiplication and addition. Notably, if the networks do not include BN layers, we can achieve them by setting $\gamma' = S_d S_w$ and $\beta' = 0$ without re-design.

In the fundamental network, LeakyReLU is used to activate the output feature maps of a convolutional layer. With the network quantization, a quantization layer is placed before the following convolutional layer. Both the activation function and the quantization layer involve one floating-point multiplication. Like the previous layer fusion, the activation function can be fused into the following quantization layer. However, the max-pooling layer may make the two layers non-adjacent. Considering that the max-pooling layer only involves a comparison operation, and the change of calculation order will not result in wrong network calculations, we placed the following quantization layer before the current max-pooling layer. The fused activation layer is defined as

$$d' = \begin{cases} clamp\left(Int\left(\frac{1}{S_d} \times x\right), \left(-2^{N_d-1}+1\right), \left(2^{N_d-1}-1\right)\right) & x \geq 0 \\ clamp\left(Int\left(\frac{a}{S_d} \times x\right), \left(-2^{N_d-1}+1\right), \left(2^{N_d-1}-1\right)\right) & x < 0 \end{cases} \tag{15}$$

Similar to the fused BN layer, the multiplying factors $1/S_d$ and a/S_d of this fused layer can be calculated off-line before implementation.

Figure 5 shows the optimized processing block with network quantization and layer fusion. Compared with the original, most float-point calculations have been converted to an 8-bit fixed-point version, and the processing flow is optimized. These optimizations can reduce the requirement of hardware resources.

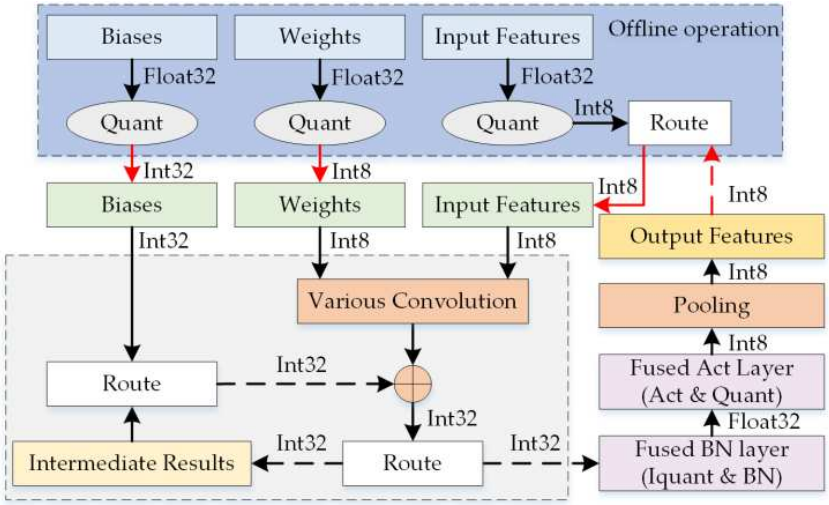


Figure 5. The forward inference calculation process after network quantization and layer fusion.

3.3. Unified Implementation of Multiple Convolutions

As shown in Figure 1, the fundamental network contains many different types of convolutional layers. It is ineffective and inflexible to customize the hardware module for each type of convolution. Furthermore, while the data type is optimized by network quantization, the volume of calculations in the convolutional layers is constant. It is still a great challenge to implement these calculations on an FPGA. To solve these problems, we first optimized the original loop computation in a standard convolutional layer to run it efficiently on an FPGA. We then proposed some transformation methods to convert all other types of convolutions in the fundamental network into a standard convolution by data arrangement. With these transformation methods, only one processing engine needs to be designed to implement all types of convolutional layers in the fundamental network.

3.3.1. Loop Optimization for Standard Convolutions

The execution of convolution exhibits numerous sources of parallelism. Due to hardware constraints, it is impossible to exploit all of the parallelism patterns fully [39]. The standard convolutional layer contains N filters, and each filter consists of M-channels $K \times K$ kernels. We optimize the original loop computation of the standard layer, as shown in Figure 6. When calculating, the $K \times K$ rectangular window slides along the width of the input feature maps, which is called row processing. The extracted pixels need to be calculated with N corresponding kernels. However, the N kernels may not be calculated at the same time due to the limited hardware resources. Thus, we divided the N kernels into multiple groups to perform calculations, and each time n kernels participate in the calculation. This operation repeats $Ni = N/n$ times until all the N intermediate results are obtained. The N intermediate results generated by the calculation are stored in the accumulation buffer. These kernels are reused for subsequent row processing until the rectangular window shifts to the end of the channel. In a row processing, $N \times H_{out}$ intermediate results are obtained. The rectangular window then shifts toward the subsequent channels and repeats the above row processing. For each row processing, the corresponding kernels in the filters are taken out for convolution. Traversing all channels needs M row processing, which is called channel processing. After that, the rectangular window shifts down by one row and repeats the above-mentioned channel processing. The number of repetitions is $H_{in} + 2 \times pad - K + 1$, where pad is the amount of zero-padding to both sides of the input image. Therefore, to process the whole input feature maps, the weights of the filters should be read $H_{in} + 2 \times pad - K + 1$ times. To avoid performance degradation, we hide the waiting time in row processing by weight prefetching. Notably, the intermediate results produced by one row processing will be accumulated in the next row processing. Therefore, the size of the accumulation buffer is $M \times H_{out} \times 32\ bit$. This computing pattern has several

advantages. Firstly, the same feature values will not be read repeatedly from external memories, which avoids frequent memory access. Second, the intermediate results can be accumulated in time, reducing the consumption of on-chip resources. Finally, the feature values of output feature maps can be obtained row by row, which is beneficial to pooling.

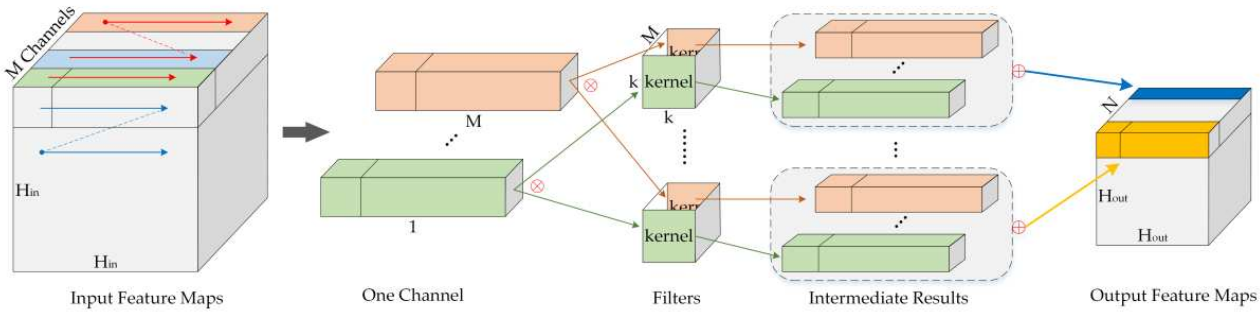


Figure 6. The computing pattern of the standard convolutions.

3.3.2. Transformation Method for Other Convolutions

The standard convolution with a kernel size of 3×3 is representative in the fundamental network. In this section, we focus on converting the remaining types of convolutions into the 3×3 standard convolutions for implementation. For a 1×1 standard convolution, we can convert the size of the kernel from 1×1 to 3×3 by zero-padding. Notably, an additional zero-padding is added to the input feature maps to obtain the correct size of the output feature maps. The calculation process after conversion is shown in Figure 7a. The essence of 3×3 transposed convolution and 3×3 standard convolution is the same. Their basic operations are both the convolution of nine weights and the corresponding feature values. Notably, the 3×3 transposed convolution has two differences. Firstly, its kernels need to be transposed for convolution. We adjust the order of the weights off-line to avoid matrix transpose operation during the calculation. The other difference with the standard convolution is that the input feature maps need to be interpolated by zero according to Equation (4). For the fundamental network, the s, pad_{in} and pad_{out} of the 3×3 transposed convolution is equal to 2, 1 and 1, respectively. Thus, 1-pixel-wide zero-padding is required in the left and top part; 2-pixel-wide zero-padding is required in the right and bottom part of the input feature maps. Moreover, 1-pixel-wide zero-padding is required between two rows and between two columns of the input feature maps. We have designed a reading strategy to efficiently complete the interpolation of feature maps and avoid extra overhead, as shown in Figure 7b. The Enable signal represents a valid signal for the input feature maps, but the feature values are not continuously read from the storage. The read enable signal of the memory is a square-wave signal corresponding to the zero-padding mode, named Rd_en signal. The Rd_en signal is set to zero when the input feature maps need to be interpolated by zero, and the input feature map at this time is set to zero. Through the above method of processing the input feature map and kernel, the transposed convolution can be converted into a 3×3 standard convolution for implementation.

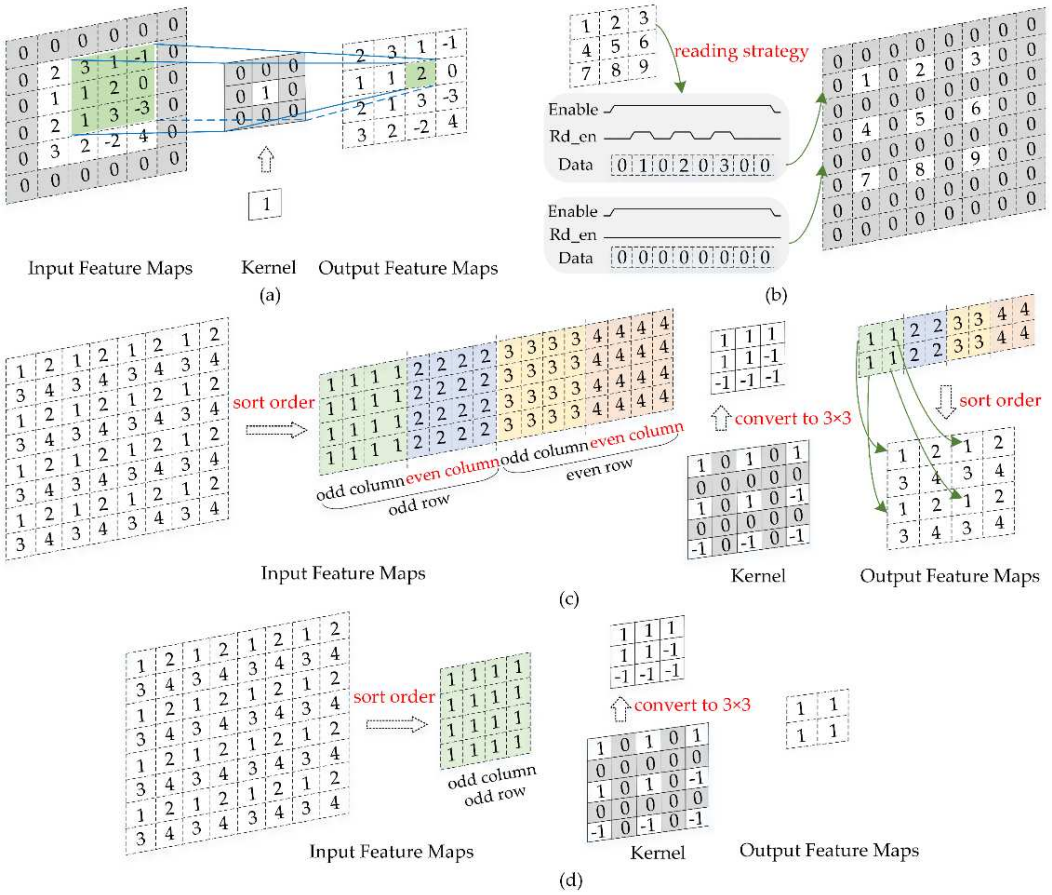


Figure 7. (**a**) The schematic diagram of the modified 1 × 1 convolution operation process. (**b**)The interpolation processing of the input feature map. If the row contains feature values, we will read the feature values and zero at intervals according to the interpolation rules; otherwise, the readout values are all equal to zero. (**c**) The modified computational process of the dilated convolution with a 1-pixel stride. (**d**) The modified computational process of the dilated convolution with a 2-pixel stride.

Moreover, the fundamental network contains two types of dilated convolutions with a stride of 1 and 2, respectively. Figure 2 illustrates the principle of the dilated convolution with a 1-pixel stride. In essence, the operational of a 3 × 3 dilated convolution with a 1-pixel stride is constant with the 3 × 3 standard convolution. Notably, the feature values are extracted from non-adjacent rows and columns in the feature maps due to the expansion of the kernel. Furthermore, the feature values used in two consecutive operations are totally different. If the kernel slides in the original order, like a 3 × 3 standard convolution, frequent memory access will cause long latency as the bandwidth is limited. Therefore, we propose an efficient method to transform the dilated convolution with a 1-pixel stride into a 3 × 3 standard convolution for implementation, as shown in Figure 7c. When reading the input feature map, we will group the feature values according to odd rows, even rows, odd columns, and even columns. The input feature values are reordered according to the group and then convolved with the 3 × 3 kernel. The output feature maps can be obtained by reordering the obtained calculation results according to the groups. In addition, the principle of the dilated convolution with a 2-pixel stride is similar to that of the dilated convolution with a 1-pixel stride. The difference is that only the feature values located in odd rows and columns of the feature maps participate in the operation. Therefore, we only need to retain the corresponding feature values when grouping and perform the same subsequent operations as the dilated convolution with 1 pixel. Its calculation process is shown in Figure 7d. With these transformations, all types of convolutions in the fundamental network are converted to 3 × 3 standard convolutions. Compared to the

method of designing custom architectures for each convolution, our method can reduce resource overhead and increase flexibility.

4. Hardware Implementation

As per the previous section, we optimized the fundamental network to reduce the complexity of FPGA-based hardware implementation. Based on this, in this section, a hardware architecture to implement the optimized fundamental network is presented. As shown in Figure 8, the proposed hardware architecture is composed of an Advanced RISC Machines (ARM)-centric processing system (PS) and programmable logic (PL). The PS contains general-purpose input/output (GPIO), Direct Memory Access (DMA) support, an Ethernet interface, an interrupt controller, etc. The PL contains the following main components: An input data reordering module, a decoding module, a DDR controller, a memory generator interface (MIG), a parameters buffer, and a processing array. Both PS and PL have an external Double Data Rate (DDR) SDRAM memory, called PS-DDR and PL-DDR, respectively. The PS-DDR and PL-DDR communicate with the FPGA through the DMA and the MIG IP core, respectively. The PS-DDR is mainly used to store feature maps, while the PL-DDR is used to store network parameters. A host PC connects with the PS through the Ethernet interface. The host PC is used to provide images to the PS-DDR and parameters to the PL-DDR and to generate the detection results.

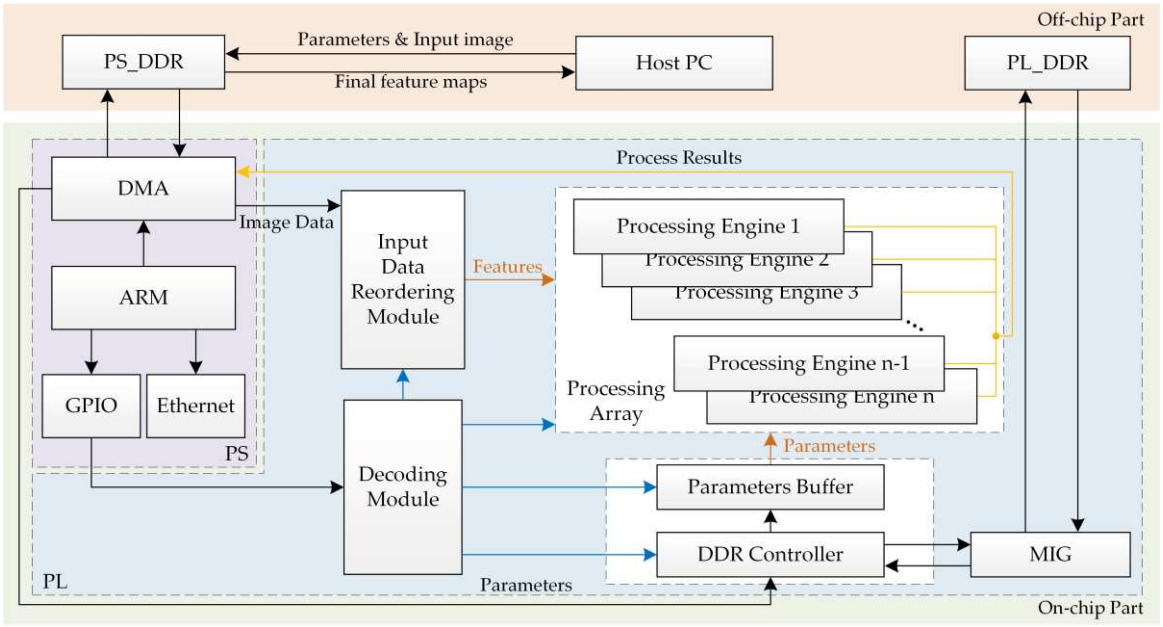


Figure 8. The overall diagram of the proposed hardware architecture.

During work, the configuration instruction of each layer is performed with ARM and transmitted to PL by GPIO. After the decoding module decodes the instruction, control signals are sent to the relevant modules. DMA fetches the original image from the PS-DDR and transmits it to PL. The input data reordering module rearrange the pixels and feeds them to the processing array. The DDR controller in PL fetches the model parameters from the PL-DDR to the parameters buffer, and the parameters buffer then provides parameters to the processing array. In the proposed architecture, N PEs are adopted to build a processing array for parallel computing. These parallel PEs share the same input feature map and calculate for different output channels. These PEs complete the calculation of each layer in parallel. Finally, the output feature maps of the last layer are transferred back to the host PC from the PS-DDR. With the final feature maps, the host PC performs Non-Maximum Suppression (NMS) to obtain the object detection results. The details are discussed in the following subsections.

4.1. Processing Engine Architecture Design

With the optimizations in Section 3, all calculations during the inference phase in the fundamental network are divided into multiple identical processing blocks. Thus, only one hardware module is needed to be designed to deploy the fundamental network on the FPGA. To achieve this goal, we propose an efficient PE. The architecture of the proposed PE is as shown in Figure 9. In the proposed PE, a finite state machine (FSM) is adopted to configure the routers to achieve the calculations in different processing blocks. Two local memories in the PE are used as a data register. The first local memory is used to store the intermediate results during calculations. These intermediate results can be accumulated in the overlap-add operation. The second local memory is used to cache one row of feature maps for the max pooling. These rows will be read out to achieve the pooling operation when the next row of convolutional results is obtained. The convolution unit is used to implement the standard 3×3 convolution. This unit is composed of nine multipliers and an adder tree. The operation bit width of each stage in the adder tree is increased by one bit to prevent overflow. A clamp function is used to prevent the overflow of the 32-bit adders in the adder tree. With this function, the values that exceed the upper and lower bounds are set to the $2^{31} - 1$ and $2^{-31} + 1$, respectively. The pseudo-code of the clamp operation in hardware implementation is shown as Algorithm 1. A fixed-to-float conversion unit is used to convert the fixed-point convolutional results to the floating-point values for the following fused layers. After the floating-point calculations of two fused layers, the results are converted to the integer. In addition, the comparison operation is used to achieve max pooling.

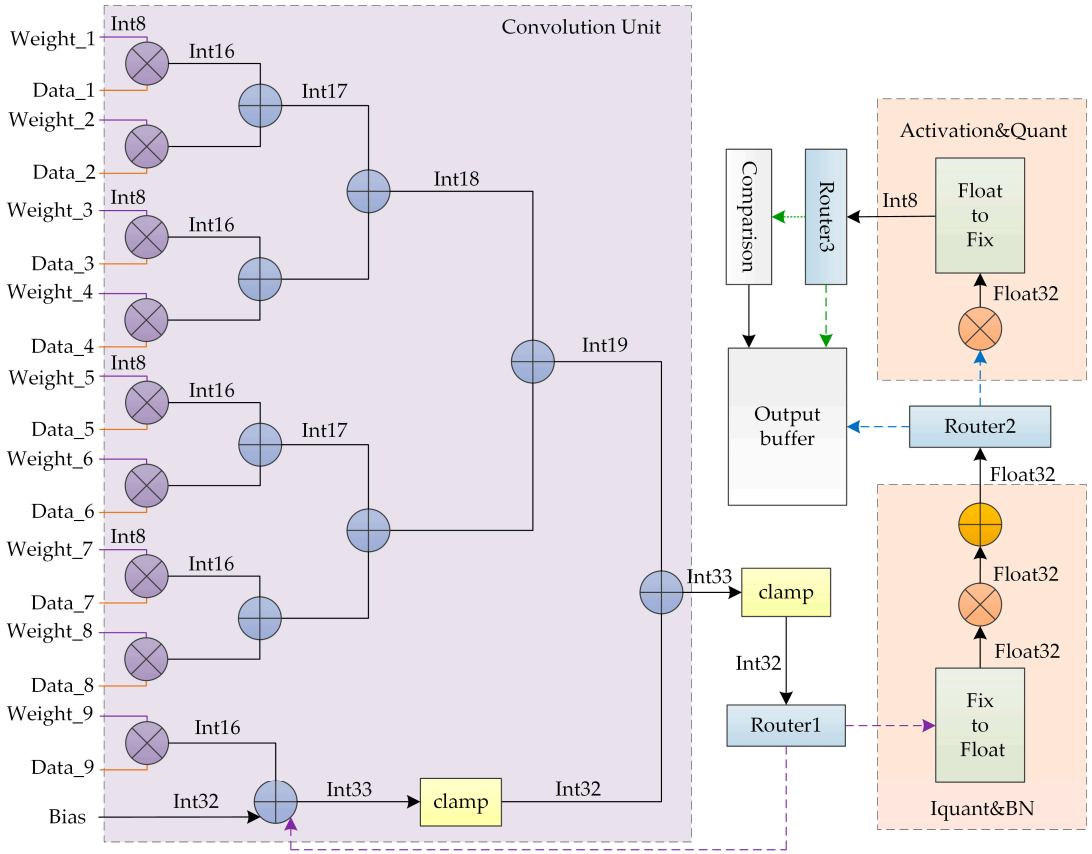


Figure 9. The architecture of the proposed processing engine (PE).

Algorithm 1. The pseudo-code of the clamp operation in hardware implementation.

```
if   Int33 (32) = '0'   then          // int33 is positive
  if   Int33 (31) = '0' then
    Int32 ≤ Int33 (31 downto 0);
  else
    Int32 ≤ x"7fffffff";       // Limit the maximum feature values to 2^31 − 1
  end if;
else                          // int33 is negative
  if   Int33 (32 downto 0) = x"80000000" then
    Int32 ≤ x"80000001";    // Exclude the case of 2^−31
  else if   Int33 (31) = '1' then
    Int32 ≤ Int33 (31 downto 0);
  else
    Int32 ≤ x"80000001";    // Limit the minimum feature values to 2^−31 + 1
  end if;
end if;
```

4.2. Data Storage and Transmission

In this section, we focus on the data storage and transmission scheme to efficiently deploy the network on an FPGA with the proposed PEs. To achieve this goal, a multi-level memory structure and a corresponding data path is designed to reuse calculation data and effectively access external memories. Notably, the design of multi-level memories is extended and based on our previous work [40]. To buffer and rearrange the feature maps, an input data reordering module is designed. As shown in Figure 10, this module is mainly composed of four block random access memories (Brams). The four Brams respectively store four rows of all channels. This module is designed as a Ping-Pong buffer. For example, at the beginning of the calculation, the first three rows of input feature maps are stored in Bram_1, Bram_2, and Bram_3 for calculation. At the same time, the fourth row of input feature maps are written into Bram_4. At this time, Bram_1, Bram_2, and Bram_3 is the Ping buffer; Bram_2, Bram_3, and Bram_4 is the Pong buffer. After the calculation of the current three rows is completed, the fifth row of feature maps are written into Bram_1. At this time, Bram_2, Bram_3, and Bram_4 is Ping buffer; Bram_3 Bram_4 and Bram_1 is Pong buffer. Notably, the Brams of the Ping buffer and Pang buffer are in order and cannot be reversed. This way can continuously provide feature values for the PEs, and hence enables low-latency calculations and improve the DDR bandwidth utilization rate. In addition, the zero-padding is achieved by selecting zero as the outputs of the input data reordering module in the appropriate place.

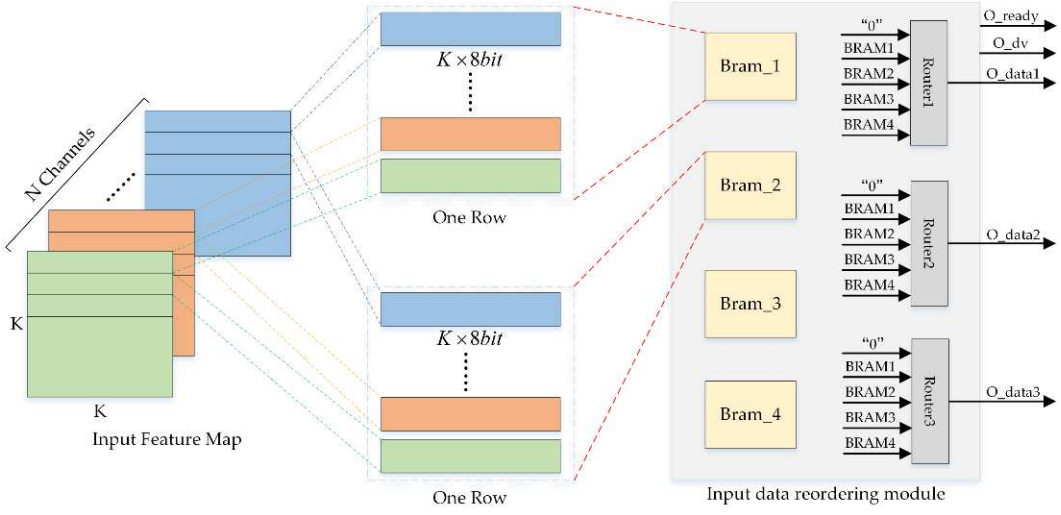


Figure 10. The diagram of the input data reordering module.

The efficient calculation of PEs requires not only feature values but also parameters. For the fundamental network, the multiplying factors of the fused activation layer in each layer are two fixed values, which can be directly initialized in the read-only memory (ROM). The other parameters of each layer are stored in external memory due to the high volume. Figure 11 illustrates the parameter storage module, which contains the DDR controller and parameters buffer. The parameters buffer is used to provide parameters to PEs. The DDR controller is responsible for interacting with the PL-DDR though the MIG IP core. It is used to read parameters from the PL-DDR according to the control signals and store them in the parameters buffer.

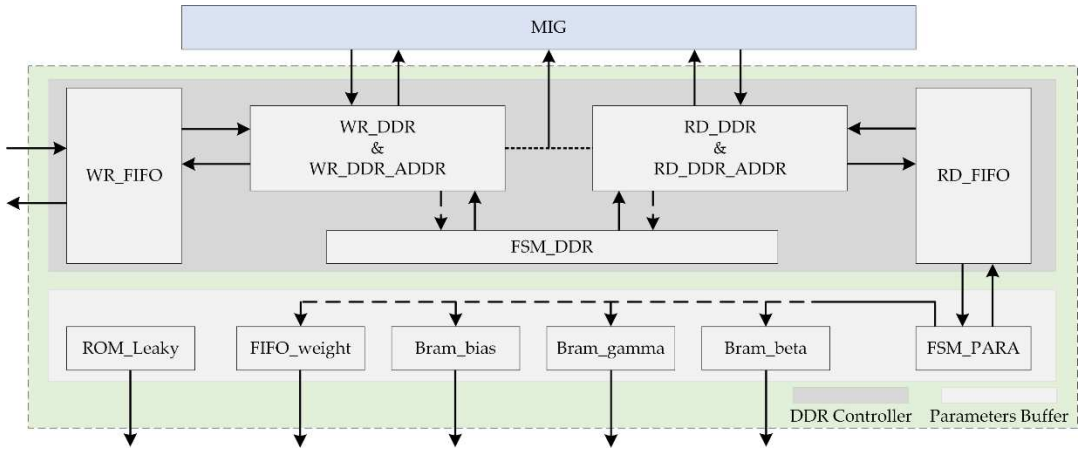


Figure 11. The diagram of the parameter storage module, which is composed of a Double Data Rate (DDR) controller and parameters buffer.

During inference phase, the parameters of each layer need to be reused multiple times in the operation. For the fundamental network, the data amount of $\hat{b}$, γ', and β is small. Thus, these parameters of a layer can be stored in Brams. Nevertheless, it is not feasible to cache all weights in a large layer. Therefore, they are cached in FIFO and discarded immediately after a single use. In this case, we read them from the PL-DDR multiple times to meet the computing demand. The time spent for repeatedly reading weights is completely overlapped by the calculation time, which ensures the acceleration performance.

With the provided parameters and feature maps, the processing array can obtain the corresponding calculation results. These results will be transmitted to the PS-DDR through the DMA. As shown in Figure 12, we divide the storage space of the PS-DDR into several subspaces, which are used to store the results of different stages. Among them, two subspaces constitute a set of memories for alternately storing input and output feature values of a layer. In particular, if the output values are related to the route layer, they are stored in another subspace to avoid being overwritten. The results of the current layers will be rewritten into the input data reordering module as the input feature map of the following layer. When all the network calculations are achieved, the output feature maps of the final layer will be transmitted to the host PC via the Ethernet interface. The host PC runs NMS and obtains the detection results of the remote sensing image.

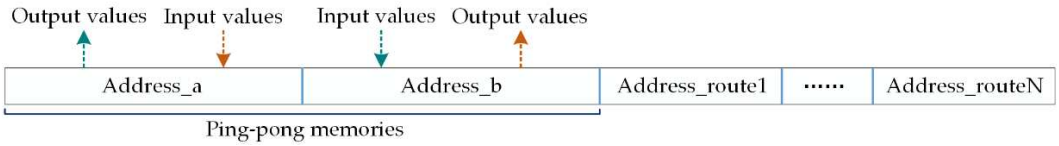


Figure 12. The diagram of the storage strategy in the processing system (PS)-DDR.

5. Experimental Evaluation and Results

In this section, we evaluate the performance of the proposed design by several experiments. The evaluation experiments were divided into two parts. First, the quantized fundamental network was trained and tested on a publicly available remote sensing image scene dataset to evaluate its detection metrics and obtain hybrid-type parameters for FPGA implementation. We then implemented the quantized network on the FPGA using the proposed architecture and tested the implementation processing performance. The experimental settings and detailed experimental results are provided below.

5.1. Experimental Settings

5.1.1. Dataset Description

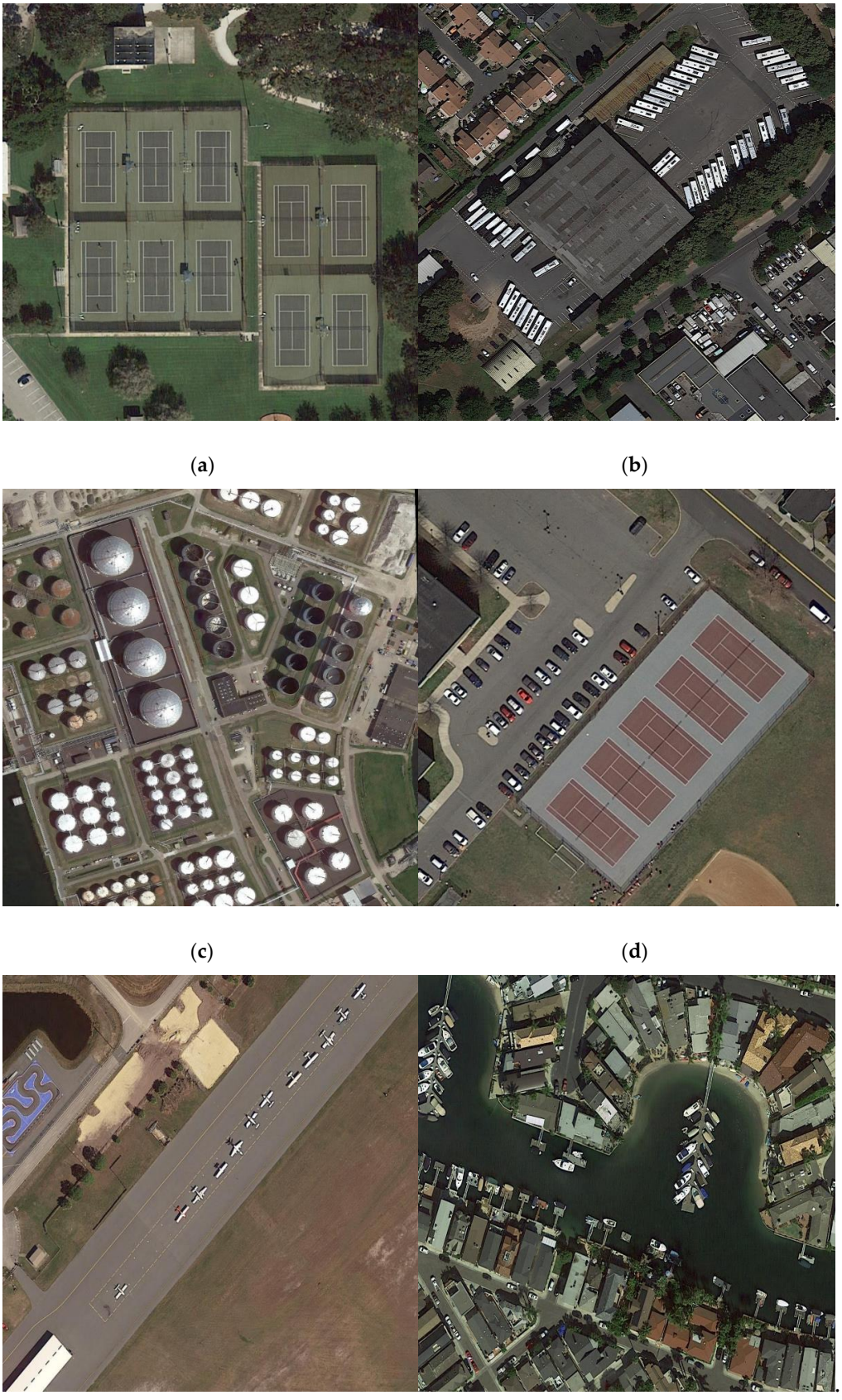
A large-scale and challenging dataset for object detection in aerial images (DOTA-v1.0) was used to evaluate the performance of the quantized fundamental network. This dataset contains 2806 aerial images with resolution ranges from 800 × 800 to 4000 × 4000. The fully annotated images in this dataset contain 188,282 instances of 15 object categories: plane (PL), baseball diamond (BD), bridge (BR), ground track field (GTF), small vehicle (SV), large vehicle (LV), ship (SH), tennis court (TC), basketball court (BC), storage tank (ST), soccer-ball field (SBF), roundabout (RA), harbor (HA), swimming pool (SP), and helicopter (HC). We used the training set of the DOTA dataset to train the quantized fundamental network and used the validation set to test it. In the training phase, all images were cropped to 1024 × 1024-pixel patches by the DOTA development kit in the experiment. Standard data augmentation tricks including random crops, rotations, and hue, saturation, and exposure shifts were applied to the images during the training phase. Following previous work [4], the images were first cropped with a stride of 512 pixels in the testing phase. The detection results of each patch were then combined to obtain the results of the original images. Samples of the DOTA dataset are shown in Figure 13.

5.1.2. Experimental Procedure

To quickly evaluate the quantized fundamental network and obtain the hybrid-type parameters for FPGA implementation, we adopted a fine-tuning for the quantized network training. In our previous work [4], we obtained the weight parameters that perform best on the DOTA validation set. The weight parameter was used to initialize the quantized network. The quantized network was then trained for 20 epochs. The weight parameters were optimized by an Adam optimization method with a weight decay of 0.0005. A multi-step learning rate was adopted. The detailed settings of the learning rate for this experiment are shown in Table 1. The batch size was set to 14. This experiment was performed on two NVIDIA Titan Xp GPUs with PyTorch 1.2.0 and TorchVision 0.4.0.

Table 1. The setting of the learning rate for the experiment.

Epoch	1	2–11	12–20
Learning rate	0.000001	0.0001	0.00001



(**a**) (**b**)

(**c**) (**d**)

(**e**) (**f**)

Figure 13. Some images in the validation set of the dataset for object detection in aerial images (DOTA-v1.0).

To test the hardware implementation processing performance, we performed the quantized network on a hardware platform with a Xilinx ZYNQ xc7z035 SoPC chip and two Micron DDR3 SDRAMs. DDR3 SDRAMs were used as the PL-DDR and PS-DDR, respectively. The proposed design was implemented with Very-High-Speed Integrated Circuit Hardware Description Language (VHDL) and synthesized with Vivado Design Suite 2017.2. The codes on the embedded processor ARM and host PC were designed with C and python, respectively. The power results were obtained by the Vivado Power Analysis tool. For the fundamental network, the number of output channels is a multiple of 32, except for the last layer. Thus, we chose 32 PEs to build the processing array based on a trade-off between resource overhead and calculation time. For the last layer, the unneeded PEs were not activated.

5.2. Performance Evaluation

Table 2 presents the hardware resource utilization of our design. The utilized values of the look-up-table (LUT), Flip Flop (FF), Bram, and digital signal processing (DSP) were 83,240, 108,883, 369, and 192, respectively. The Brams were primarily consumed by the on-chip buffers. The embedded DSP slices were mainly used to implement the calculations of the network. As shown in Table 2, the available resources of the Xilinx ZYNQ xc7z035 SoPC are limited. However, with our implementation method, the large-scale fundamental network was successfully deployed on the SoPC platform with appropriate resource utilization.

Table 2. The hardware resource utilization of the design.

Resource	LUT	FF	Bram	DSP
Available	171,900	343,800	500	900
Utilization	83,240	108,883	369	192
Utilization (%)	48.4%	31.7%	74%	21.3%

To evaluate the performance of the hardware implementation, we used the throughput of the hardware implementation and detection accuracy as evaluation criteria. The throughput performance was defined as the total operations divided by the required execution time. The total operations reflect the complexity of the network in terms of operations. For the FPGA implementation in this design, the total operations of the quantized fundamental network are 379 GOPs. For the input feature maps with a size of $1024 \times 1024 \times 3$, it takes 3.4 s to get the output results of the last layer on the Xilinx ZYNQ xc7z035 FPGA. 'GOP/s' is an abbreviation of giga-operations per second. The proposed design achieved an overall throughput of 111.5 GOP/s for the quantized fundamental network under the 200MHz working frequency of the FPGA. Moreover, the mAP is most commonly used to evaluate the accuracy of object detection [41]. It indicates the mean accuracy of each class considering recall and precision. Generally, a higher mAP indicates better performance. Experiments show that our hardware implementation deployed on a ZYNQ xc7z035 device achieves a 69.32% mAP on the DOTA dataset. Some results on the DOTA dataset are shown in Figure 14. It can be seen that the arbitrarily oriented objects can be correctly detected by our hardware implementation on the FPGA.

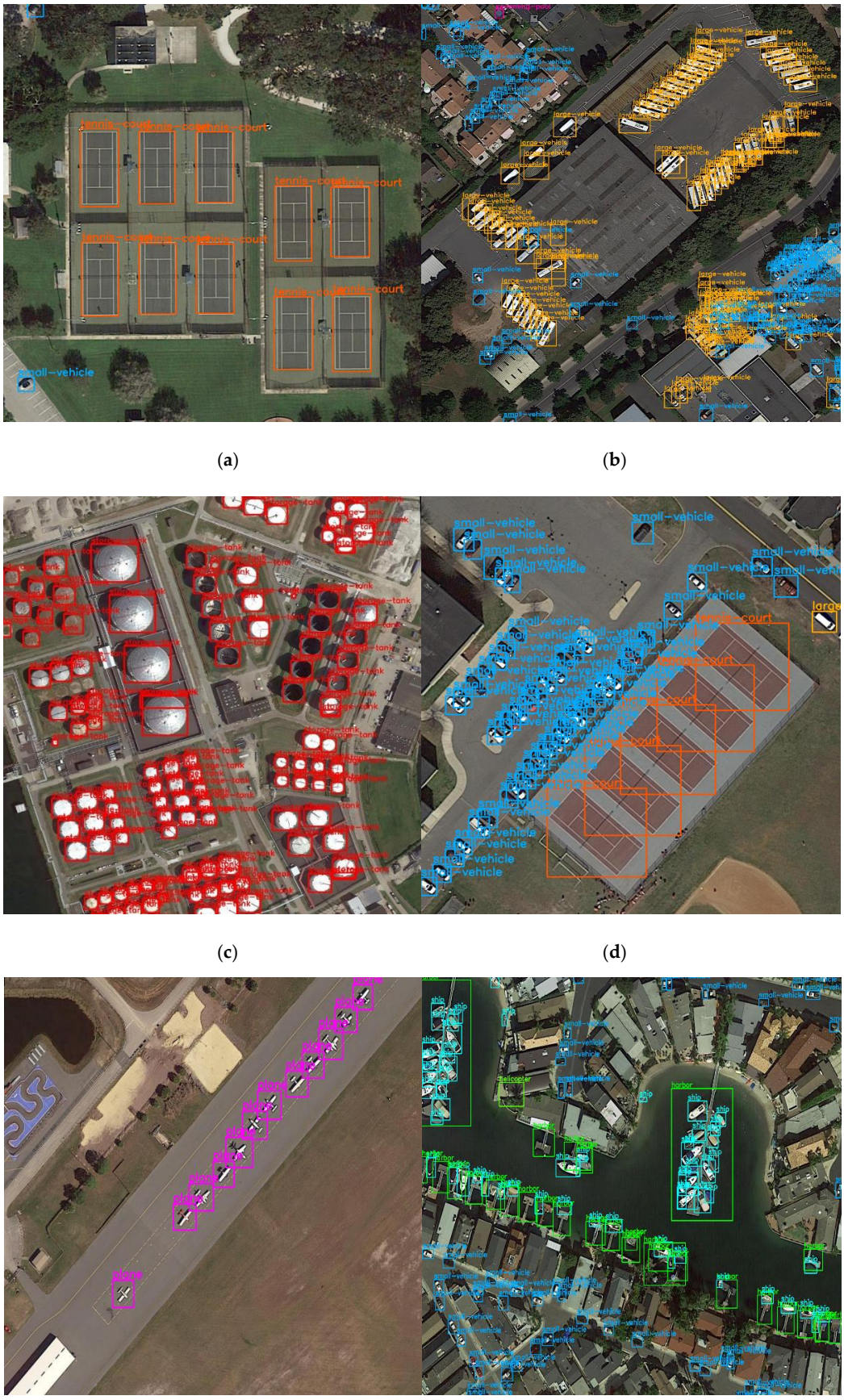


Figure 14. Some results on the DOTA dataset that were detected by our hardware architecture on Xilinx ZYNQ xc7z035 SoPC.

5.3. Performance Comparison

Several comparative experiments were conducted to show the effectiveness of our implementation. Firstly, the proposed FPGA-based hardware design was compared with different off-the-shelf platforms. We implemented the fundamental network on an Intel Xeon Gold 5120T with 128 GB DDR4 DRAM and an NVIDIA Titan Xp GPU with 12 GB GDDR5X memory. The main results of the CPU, the GPU, and the proposed design are listed in Table 3. Notably, a strategy of network quantitation was adopted on these platforms when mapping the fundamental network. The Thermal Design Power (TDP) values of the CPU and GPU were 105 W and 250 W, respectively. According to the power report supplied by the Vivado Design Suite, the total on-chip power of our hardware architecture was only 5.96 W. Therefore, our design is suitable for deployment in power-limited application scenarios. We can see from Table 3 that the GPU has obvious advantages in terms of throughput among the platforms. The throughput of the GPU was 89.6 times that of the CPU and 47.3 times that of the proposed design, respectively. Compared with the CPU, the energy efficiency of our hardware architecture was 33.4 times higher. Additionally, the energy efficiency of our hardware architecture reached 89% of that of the GPU. Thus, our hardware architecture had better performance and energy efficiency than the CPU, and its energy efficiency was comparable with GPU at 1.6 GHz. In addition, our design is scalable. The performance and energy efficiency can be improved by increasing the number of PEs. As shown in the experimental results in Table 3, the detection accuracy of our implementation on the FPGA was only 0.18% lower than the mAP of the quantized fundamental network deployed on the GPU. We concluded that the reason for the limited accuracy loss was that we had fused some operations for hardware implementation. This degree of accuracy loss is acceptable in practical applications.

Table 3. The evaluation results on the Central Processing Unit (CPU), the Graphic Processing Unit (GPU), and the proposed architecture on the field-programmable gate array (FPGA).

Platform	CPU	GPU	FPGA
Device	Intel Xeon Gold 5120T	NVIDIA Titan Xp	Xilinx ZYNQ xc7z035
Technology (nm)	14	16	28
Frequency (MHz)	2200	1582	200
Power (W)	105	250	5.96
Accuracy (mAP)	0.6750	0.6750	0.6732
Throughput (GOP/s)	58.9	5279.4	111.5
Power Efficiency (GOP/s/W)	0.56	21.12	18.71
[Ratio]	[1.0]	[37.7]	[33.4]

The proposed design was also compared with related, state-of-the-art works. The performance comparison is shown in Table 4, wherein the relevant references are indicated. In Reference [42], a novel method to implement the YOLOv1 network framework on an FPGA is presented. However, the implementation allocated independent hardware resources for convolution and fully connected layers. This method limits the utilization of available resources. This design used 800 DSPs and its performance was only 18.82 GOP/s. In Reference [43], a Tiny-YOLOv2 algorithm was implemented on an FPGA, which contains nine convolutional layers and six max-pooling layers. As shown in Table 4, this work reported a low resource overhead, which is hardware-friendly for implementation. However, its processing performance was significantly limited—only 21.6 GOP/s. The authors of [44] successfully deployed Yolov2 on the Xilinx ZYNQ xc7z020 FPGA, a chip with limited resources. However, the design consumes almost all the DSPs on the chip, 211/220. Therefore, this design has limited expansion capabilities. In Reference [45], the YOLOv2 model was implemented on a Xilinx ZCU102 FPGA with an overall performance of 102.5 GOP/s, at a 300 MHz clock frequency. However, this design has extremely high requirements for computing resources—600 DSPs. Among the related works listed in Table 4, the [46] has the highest processing performance, reaching 500 GOP/s. Meanwhile, this design occupies the most logical resources, consuming more than 1000 DSPs and Brams. It

is difficult to deploy the design in resource-limited scenarios. The processing performance of a design is closely related to resource overhead and operating frequency [12]. Hence, for a fair comparison, we considered the performance density, which is defined as the number of operations that one DSP slice executes in one cycle [13]. Compared with the above works, our design not only has superior processing speed but also performs best in performance density, which reached 2.90 OP/DSP/cycle. The comparison with these related works demonstrates that our design can strike a satisfactory balance between resource consumption and computing time cost and is suitable for deployment on embedded devices with a limited resource budget.

Table 4. The performance comparison of our design and those of previous works.

	[42]	[43]	[44]	[45]	[46]	**Our Work**
Platform	ZC706	Cyclone V	ZYNQ XC7Z020	ZCU102	Arria-10 GX1150	ZYNQ xc7z035
Frequency (MHz)	200	117	150	300	190	200
Network	YOLOv1	Tiny-YOLOv2	YOLOv2	YOLOv2	YOLOv2	Improved YOLOv2
Image Size	N/A	416 × 416	416 × 416	416 × 416	288 × 288	1024 × 1024
Precision	32-bit fixed	16-bit fixed	16-bit fixed	16-bit fixed	8-bit fixed	8-bit fixed W float BN&A [3]
LUTs	N/A	113 K	N/A	95 K	145 K [1]	82 K
FFs	N/A	N/A	N/A	90 K	N/A	107 K
Brams	N/A	N/A	N/A	246	1027 [2]	369
DSPs	800	122	211	609	1092	192
Performance (GOP/s)	18.82	21.6	33.49	102.5	500	111.5
Performance Density (OP/DSP/cycle)	0.12	1.51	1.06	0.56	2.41	2.90

[1] The reported logical cells are ALM. [2] The model of the reported RAM is M20K, and its storage size is 20 Kb. We converted it into 36 Kb storage. [3] W: weight, BN&A: batch normalization and activation.

6. Discussion

We implemented an improved YOLOv2 network on an FPGA for large-scale optical remote sensing image object detection. Our implementation is a hardware/software co-design, which improves generalizability and flexibility. Compared with related works, we have significant advantages in hardware resource requirements. Therefore, in pursuit of further improving performance, we can improve the processing speed of our design by appropriately increasing the number of PEs. In future work, we will focus on improving the resource overhead and computing performance of the proposed architecture. To achieve this goal, we aim to implement the proposed design on a Xilinx MP-SoC board with a large on-chip memory. The external memories can be replaced by taking full advantage of the on-chip UltraRAMs. In this case, the design could perform at an even faster rate while using less power.

7. Conclusions

In this paper, we propose a hardware implementation method for CNN-based optical remote sensing object detection under power-limited conditions. First, we optimized the fundamental network for hardware implementation. The optimization mainly includes three aspects: network quantization, layer fusion, and the unified implementation of multiple convolutions. With these optimization methods, we effectively reduced the scale of the network and the resource requirements during deployment. We further propose a hardware architecture for the CNN-based remote sensing object detection model based on these optimizations. In this architecture, a PE is proposed to implement multiple types of convolutions in the network. An efficient data storage and access scheme is also proposed, and it achieves low-latency calculations and a high memory bandwidth utilization rate. We deployed an improved YOLOv2 network on a Xilinx ZYNQ xc7z035 FPGA using the proposed hardware architecture. The experimental results show that our design achieves an overall throughput of 111.5 GOP/s and an energy efficiency of 18.71 GOP/s/W under the 200 MHz working frequency of the FPGA. Compared with the CPU, the proposed accel-

erator improves energy efficiency by 33.4 times. Additionally, the energy efficiency of our hardware architecture can reach 89% of that of the GPU. Moreover, the performance of the proposed accelerator can be further improved by increasing the number of PEs. The total on-chip power of our hardware architecture was only 5.96 W, which is much lower than the power consumption of the CPU and the GPU. In addition, experimental results tested on the DOTA dataset show that the proposed design can strike an excellent balance between hardware resource overhead and time cost. The detection accuracy of our implementation on the FPGA is only 0.18% lower than the mAP of the quantized fundamental network deployed on the GPU. Furthermore, several recent advanced FPGA-based implementations were compared to verify the superiority of the proposed hardware accelerator.

Author Contributions: Conceptualization, N.Z. and X.W.; methodology, N.Z. and X.W.; software, N.Z., X.W., and W.L.; validation, N.Z., X.W., and W.L.; formal analysis, N.Z. and W.L.; investigation, N.Z. and X.W.; resources, H.C. and W.L.; writing—original draft preparation, N.Z.; writing—review and editing, N.Z., X.W., and W.L.; supervision, H.C.; project administration, H.C.; funding acquisition, H.C. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the National Key R & D Program of China under Grant No. 2017YFB0502800 and Grant No. 2017YFB0502804, the MYHT Program of China under Grant No. B0201.

Acknowledgments: This work was supported by the Chang Jiang Scholars Program under Grant T2012122 and the Hundred Leading Talent Project of Beijing Science and Technology under Grant Z141101001514005.

Conflicts of Interest: The authors declare that there is no conflict of interest.

References

1. Cheng, G.; Han, J. A survey on object detection in optical remote sensing images. *ISPRS J. Photogramm. Remote Sens.* **2016**, *117*, 11–28. [CrossRef]
2. Zeng, D.; Zhang, S.; Chen, F.; Wang, Y. Multi-scale CNN based garbage detection of airborne hyperspectral data. *IEEE Access* **2019**, *7*, 104514–104527. [CrossRef]
3. He, H.; Yang, D.; Wang, S.C.; Wang, S.Y.; Li, Y. Road extraction by using atrous spatial pyramid pooling integrated encoder-decoder network and structural similarity loss. *Remote Sens.* **2019**, *11*, 1015. [CrossRef]
4. Liu, W.; Ma, L.; Wang, J.; Chen, H. Detection of Multiclass Objects in Optical Remote Sensing Images. *IEEE Geosci. Remote Sens.* **2018**, *16*, 791–795. [CrossRef]
5. Zhu, M.; Xu, Y.; Ma, S.; Li, S.; Ma, H.; Han, Y. Effective Airplane Detection in Remote Sensing Images Based on Multilayer Feature Fusion and Improved Nonmaximal Suppression Algorithm. *Remote Sens.* **2019**, *11*, 1062. [CrossRef]
6. Liu, W.; Long, M.; He, C. Arbitrary-Oriented Ship Detection Framework in Optical Remote-Sensing Images. *IEEE Geosci. Remote Sens.* **2018**, *15*, 937–941. [CrossRef]
7. Gong, Z.; Zhong, P.; Hu, W.; Hua, Y. Joint learning of the center points and deep metrics for land-use classification in remote sensing. *Remote Sens.* **2019**, *11*, 76. [CrossRef]
8. Qi, B.; Shi, H.; Zhuang, Y.; Chen, H.; Chen, L. On-Board, Real-Time Preprocessing System for Optical Remote-Sensing Imagery. *Sensors* **2018**, *18*, 1328. [CrossRef] [PubMed]
9. Joyce, K.E.; Belliss, S.E.; Samsonov, S.V.; McNeill, S.J.; Glassey, P.J. A Review of the Status of Satellite Remote Sensing and Image Processing Techniques for Mapping Natural Hazards and Disasters. *Prog. Phys. Geogr.* **2009**, *33*, 183–207. [CrossRef]
10. Zhou, G.; Zhang, R.; Liu, N.; Huang, J.; Zhou, X. On-Board Ortho-Rectification for Images Based on an FPGA. *Remote Sens.* **2017**, *9*, 874. [CrossRef]
11. Du, Q.; Nekovei, R. Fast real-time onboard processing of hyperspectral imagery for detection and classification. *J. Real-Time Image Process.* **2009**, *4*, 273–286. [CrossRef]
12. Li, L.; Zhang, S.; Wu, J. Efficient Object Detection Framework and Hardware Architecture for Remote Sensing Images. *Remote Sens.* **2019**, *11*, 2376. [CrossRef]
13. Liu, Z.; Chow, P.; Xu, J.; Jiang, J.; Dou, Y.; Zhou, J. A Uniform Architecture Design for Accelerating 2D and 3D CNNs on FPGAs. *Electronics* **2019**, *8*, 65. [CrossRef]
14. Wei, X.; Liu, W.; Chen, L.; Ma, L.; Chen, H.; Zhuang, Y. FPGA-Based Hybrid-Type Implementation of Quantized Neural Networks for Remote Sensing Applications. *Sensors* **2019**, *19*, 924. [CrossRef] [PubMed]
15. PD, S.M.; Lin, J.; Zhu, S.; Yin, Y.; Liu, X.; Huang, X.; Song, C.; Zhang, W.; Yan, M.; Yu, H.; et al. A scalable network-on-chip microprocessor with 2.5 D integrated memory and accelerator. *IEEE Trans. Circuits Syst. Regul. Pap.* **2017**, *64*, 1432–1443.
16. Li, W.; He, C.; Fu, H.; Zheng, J.; Dong, R.; Xia, M.; Yu, L.; Luk, W. A Real-Time Tree Crown Detection Approach for Large-Scale Remote Sensing Images on FPGAs. *Remote Sens.* **2019**, *11*, 1025. [CrossRef]

17. Lei, C.; Xin, W.; Wenchao, L.; He, C.; Liang, C. Hardware Implementation of Convolutional Neural Network Based Remote Sensing Image Classification Method. In *Proceedings of the 9th International Conference on Communications, Signal Processing, and Systems, Online, 14–16 December 2020*; Liang, Q., Wang, W., Liu, X., Na, Z., Li, X., Zhang, B., Eds.; Springer: Singapore, 2020; pp. 140–148.
18. Mohammadnia, M.R.; Shannon, L. A multi-beam Scan Mode Synthetic Aperture Radar processor suitable for satellite operation. In Proceedings of the 2016 IEEE 27th International Conference on Application-Specific Systems, Architectures and Processors (ASAP), London, UK, 6–8 July 2016; pp. 83–90.
19. Gonzalez, C.; Bernabe, S.; Mozos, D.; Plaza, A. FPGA implementation of an algorithm for automatically detecting targets in remotely sensed hyperspectral images. *IEEE J. Sel. Top. Appl. Earth Obs. Remote Sens.* **2016**, *9*, 4334–4343. [CrossRef]
20. Yap, J.W.; Yussof, Z.M.; Salim, S.I.M. A scalable FPGA based accelerator for Tiny-YOLO-v2 using OpenCL. *Int. J. Embed. Syst.* **2019**, *8*, 206–214.
21. Zhang, C.; Li, P.; Sun, G.; Guan, Y.; Xiao, B.; Cong, J. Optimizing fpga-based accelerator design for deep convolutional neural networks. In Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 22–24 February 2015; pp. 161–170.
22. Peemen, M.; Setio, A.A.; Mesman, B.; Corporaal, H. Memory-centric accelerator design for convolutional neural networks. In Proceedings of the 2013 IEEE 31st International Conference (ICCD), Asheville, NC, USA, 6–9 October 2013; pp. 13–19.
23. Sun, F.; Wang, C.; Gong, L.; Xu, C.; Xu, C.C.; Zhang, Y.W.; Lu, Y.T.; Li, X.; Zhou, X. A High-Performance Accelerator for Large-Scale Convolutional Neural Networks. In Proceedings of the 2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC), Guangzhou, China, 28 May 2018; pp. 1–9.
24. Li, H.; Fan, X.; Li, J.; Wei, C.; Zhou, X.; Wang, L. A high performance FPGA-based accelerator for large-scale Convolutional Neural Networks. In Proceedings of the 26th International Conference on Field Programmable Logic and Applications (FPL), Lausanne, Switzerland, 29 August–2 September 2016; pp. 34–42.
25. Zhou, Y.; Jiang, J. An FPGA-based accelerator implementation for deep convolutional neural networks. In Proceedings of the 2015 4th International Conference on Computer Science and Network Technology, ICCSNT 2015, Harbin, China, 19–20 December 2015; Volume 1, pp. 829–832.
26. Li, Z.; Wang, L.; Guo, S.; Deng, Y.; Dou, Q.; Zhou, H.; Lu, W. Laius: An 8-bit fixed-point CNN hardware inference engine. In Proceedings of the 2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC), Guangzhou, China, 12–15 December 2017.
27. Fan, H.; Liu, S.L.; Ferianc, M.; Ng, H.C.; Que, Z.Q.; Liu, S.; Niu, X.Y.; Luk, W. A real-time object detection accelerator with compressed SSDLite on FPGA. In Proceedings of the 2018 International Conference on Field-Programmable Technology (FPT), Okinawa, Japan, 11–15 December 2018.
28. Liang, S.; Yin, S.; Liu, L.; Luk, W.; Wei, S. FP-BNN: Binarized neural network on FPGA. *Neurocomputing* **2017**, *275*, 1072–1086. [CrossRef]
29. Nguyen, D.T.; Nguyen, T.N.; Kim, H.; Lee, H. A High-Throughput and Power-Efficient FPGA Implementation of YOLO CNN for Object Detection. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2019**, *27*, 1861–1873. [CrossRef]
30. Jiao, L.; Luo, C.; Cao, W.; Zhou, X.; Wang, L. Accelerating low bit-width convolutional neural networks with embedded FPGA. In Proceedings of the 27th International Conference on Field Programmable Logic and Applications (FPL), Ghent, Belgium, 4–8 September 2017; pp. 1–4.
31. Xia, G.S.; Bai, X.; Ding, J.; Zhu, Z.; Belongie, S.; Luo, J.; Datcu, M.; Pelillo, M.; Zhang, L. DOTA: A large-scale dataset for object detection in aerial images. *arXiv* **2017**, arXiv:1711.10398. Available online: https://arxiv.org/abs/1711.10398 (accessed on 27 January 2018).
32. Girshick, R.; Donahue, J.; Darrell, T.; Malik, J. Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation. In Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2014), Columbus, OH, USA, 23–28 June 2014; pp. 580–587.
33. Liu, W.; Anguelov, D.; Erhan, D.; Szegedy, C.; Reed, S.; Fu, C.Y.; Berg, A.C. SSD: Single shot multibox detector. In Proceedings of the Computer Vision–European conference on computer vision 2016, Amsterdam, The Netherlands, 11–14 October 2016; pp. 21–37.
34. Redmon, J.; Farhadi, A. YOLO9000: Better, Faster, Stronger. *arXiv* **2016**, arXiv:1612.08242. Available online: https://arxiv.org/abs/1612.08242 (accessed on 25 December 2016).
35. Nakahara, H.; Yonekawa, H.; Fujii, T.; Sato, S. A Lightweight YOLOv2: A Binarized CNN with A Parallel Support Vector Regression for an FPGA. In Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 25–27 February 2018; pp. 31–40.
36. Li, F.; Chen, H.; Liu, Z.; Zhang, X.D.; Jiang, M.S.; Wu, Z.Z.; Zhou, K.Q. Deep learning-based automated detection of retinal diseases using optical coherence tomography images. *Biomed. Opt. Express* **2019**, *10*, 6204–6226. [CrossRef] [PubMed]
37. Kheradpisheh, S.R.; Ghodrati, M.; Ganjtabesh, M.; Masquelier, T. Deep Networks Can Resemble Human Feed-Forward Vision in Invariant Object Recognition. *Sci. Rep.* **2016**, *6*, 32672. [CrossRef] [PubMed]

38. Ioffe, S.; Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv* **2015**, arXiv:1502.03167v3. Available online: https://arxiv.org/abs/1502.03167 (accessed on 13 February 2015).
39. Abdelouahab, K.; Pelcat, M.; Serot, J.; Berry, F. Accelerating CNN inference on FPGAs: A survey. *arXiv* **2018**, arXiv:1806.01683. Available online: https://arxiv.org/abs/1806.01683 (accessed on 26 May 2018).
40. Zhang, N.; Wei, X.; Chen, L.; Chen, H. Three-level Memory Access Architecture for FPGA-based Real-time Remote Sensing Image Processing System. In Proceedings of the 2019 IEEE International Conference on Signal, Information and Data Processing (ICSIDP), Chongqing, China, 11–13 December 2019; pp. 1–6.
41. Tao, Y.; Ma, R.; Shyu, M.L.; Chen, S.C. Challenges in Energy-Efficient Deep Neural Network Training With FPGA. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops, Seattle, WA, USA, 14–19 June 2020.
42. Zhao, R.; Niu, X.; Wu, Y.; Luk, W.; Liu, Q. Optimizing CNN-based object detection algorithms on embedded FPGA platforms. In Proceedings of the International Symposium on Applied Reconfigurable Computing, Delft, The Netherlands, 3–7 April 2017; pp. 255–267.
43. Wai, Y.J.; bin Mohd Yussof, Z.; bin Salim, S.I.; Chuan, L.K. Fixed Point Implementation of Tiny-Yolo-v2 using OpenCL on FPGA. *Int. J. Adv. Comput. Sci. Appl.* **2018**, *9*, 506–512. [CrossRef]
44. Lv, H.; Zhang, S.; Liu, X.; Liu, S.; Liu, Y.; Han, W.; Xu, S. Research on Dynamic Reconfiguration Technology of Neural Network Accelerator Based on Zynq. *J. Phys. Conf. Ser.* **2020**, *1650*, 032093. [CrossRef]
45. Zhang, S.; Cao, J.; Zhang, Q.; Zhang, Q.; Zhang, Y.; Wang, Y. An FPGA-Based Reconfigurable CNN Accelerator for YOLO. In Proceedings of the 2020 IEEE 3rd International Conference on Electronics Technology (ICET), Chengdu, China, 8–12 May 2020; pp. 74–78.
46. Xu, K.; Wang, X.; Liu, X.; Cao, C.; Li, H.; Peng, H.; Wang, D. A dedicated hardware accelerator for real-time acceleration of YOLOv2. *J. Real Time Image Process.* **2020**, 412–423. [CrossRef]