electronics

MDPI

Article

A Sliding Window for Data Reuse in Deep Convolution Operations to Reduce Bandwidth Requirements and Resource Utilization

Yiqi Sun [1,*], Yaoyang Ma [2,†], Zixuan Chen [3,†], Zhiyu Liu [4,†], Boxin Chen [5] and Rui Song [5]

[1] School of Physics, Xidian University, Xi'an 710126, China
[2] Faculty of Integrated Circuit, Xidian University, Xi'an 710126, China; 22009100559@stu.xidian.edu.cn
[3] School of Computer Science and Technology, Xidian University, Xi'an 710126, China; 22009102255@stu.xidian.edu.cn
[4] School of Telecommunications Engineering, Xidian University, Xi'an 710126, China; 22009101058@stu.xidian.edu.cn
[5] School of Telecommunications Engineering, Xidian University, Xi'an 710071, China; 23011211085@stu.xidian.edu.cn (B.C.); rsong@xidian.edu.cn (R.S.)
[*] Correspondence: 22009101120@stu.xidian.edu.cn
[†] These authors contributed equally to this work.

Abstract: Convolutional Neural Networks (CNNs) have demonstrated high accuracy in applications such as object detection, classification, and image processing. However, convolutional layers account for the majority of computations within CNNs. Typically, these layers are executed on GPUs, resulting in higher-power consumption and hindering lightweight deployment. This paper presents a design that deploys convolutional layers on FPGAs with adjustable parameters. In this FPGA deployment, a 4 × 4 3D sliding window is used to traverse the data, reducing bandwidth requirements and facilitating seamless integration with subsequent processing stages. A three-dimensional plane buffer design is proposed, which implements data reuse. Compared to directly inputting the feature map and performing the computation, it reduces the on-chip memory bandwidth requirement by 75%. Additionally, a new addressing strategy is introduced to map 3D feature maps to RAM addresses, eliminating addressing time. Due to the resource-intensive nature of high-level synthesis (HLS) technology, HDL design is used for the convolutional layers. This design achieves an inference speed of 121.36 GOPS at a 16-bit width, providing a 39.10 times increase in performance compared to CPU implementations.

Keywords: FPGA; CNN optimization; 3D sliding window; data reuse; data feeder; bandwidth reduction

Academic Editor: Alexander Barkalov

Received: 27 November 2024
Revised: 16 January 2025
Accepted: 30 January 2025
Published: 1 February 2025

Citation: Sun, Y.; Ma, Y.; Chen, Z.; Liu, Z.; Chen, B.; Song, R. A Sliding Window for Data Reuse in Deep Convolution Operations to Reduce Bandwidth Requirements and Resource Utilization. *Electronics* **2025**, *14*, 582. https://doi.org/10.3390/electronics14030582

Copyright: © 2025 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

With the rapid advancement of deep learning technologies, Convolutional Neural Networks (CNNs) have demonstrated exceptional performance in fields such as image recognition and item classification. However, the high computational complexity and vast number of parameters in CNNs have limited their application in resource-constrained environments. This issue is particularly critical in real-time processing scenarios, where efficiently executing CNNs has become an urgent challenge. Addressing this challenge, field-programmable gate arrays (FPGAs) have recently emerged as a popular research focus for CNN acceleration due to their programmability and parallel computing capabilities [1–3].

Currently, methods for accelerating convolution on FPGAs can be mainly categorized into algorithm-based acceleration (e.g., Winograd, FFT, FFA) [4] and memory-based con-

volution acceleration. The mainstream memory-based convolution acceleration methods primarily include General Matrix Multiplication (GEMM) and sliding windows. References [5,6] utilize the GEMM approach, yet this method is often accompanied by complex memory access patterns that require software platforms to manage, resulting in significant delays [5].

The sliding window method is adopted in [7–9], and it is mainly divided into 3×3 and 4×4 traversal modes. Both [8,9] employed a 3×3 sliding window. In [8], due to the inability to utilize the adder tree, the processing elements (PEs) are inefficient during computation. Conversely, ref. [9] uses the Winograd algorithm, achieving more efficient computation, though the specialized nature of the algorithm reduces its portability. Additionally, the Winograd method often results in the higher utilization of DSPs and look-up tables [10]. Kim et al. [7] employed a 4×4 traversal MAC array as the computational unit, effectively combining convolution and max-pooling. However, because it is deployed on a small FPGA board with its specific optimizations, and it employs an 8-bit quantization strategy, its precision is lower than the widely used 16-bit standard [11–13]. Consequently, it lacks an advantage in large-scale convolution computations.

Due to the convenience and ease of development offered by High-Level Synthesis (HLS) tools in the implementation of convolutional layers, many works have used HLS for this purpose [8,9,14]. However, HLS tools often consume additional resources during synthesis [15]. For instance, Ma et al. [16] demonstrate that, by transitioning from HLS to HDL development, a performance improvement of over two times was achieved while utilizing a comparable amount of resources. Additionally, HLS does not facilitate a deep understanding of the underlying resource scheduling of convolutions. Therefore, this work adopts the HDL approach for development.

Kim et al. [7] summarized methods for feature map reuse and weight reuse, and discusses strategies for their application with different feature map sizes and depths, effectively hiding loading times. Nguyen et al. [10] proposed a row-based weight reuse strategy, which requires all weights to be stored in on-chip SRAM. This approach demands significant on-chip buffer resources and is therefore not suitable for computations involving large-depth feature map blocks.

Traditional row-based caching methods are widely used to align rows and columns during convolution, and have been employed in the development of 2D filters in some earlier works. However, for multi-channel convolutions in deep learning algorithms, the 2D row cache strategy is no longer suitable. This paper proposes an addressing strategy that extends the 2D row buffer to a 3D buffer, enabling traversal in 3D space.

Main contributions:

- **Convolution computation architecture:** A new addressing method is proposed based on the 4×4 convolutional traversal scheme, leading to the development of a convolutional computing architecture optimized for large-scale convolution. This architecture is implemented on an FPGA (XC7Z035), making it suitable for deployment across various neural networks.
- **New Data Feeder Design:** A novel 3D data feeder is introduced, which represents the first extension of the traditional 2D row buffer. This design achieves a 75% reduction in on-chip bandwidth requirements and significantly decreases address computation latency for pointer addressing.
- **Performance:** Achieved a processing speed of 121.36 GOPS using minimal resources at a clock frequency of 200 MHz, resulting in a 39.10 times increase in performance compared to CPU computations.

2. Related Work

2.1. Sliding Window

The sliding window is a widely used data acquisition pattern. For instance, with a 3×3 convolution kernel, it manifests as a $3 \times 3 \times D$ window sliding over the feature map with a stride of 1. On a feature map of size $W \times H \times C_{in}$ with padding P, the total number of iterations required would be:

$$(W + P - 1) \times (H + P - 1) \times \lceil \frac{C_{in}}{D} \rceil \times C_{out} \tag{1}$$

Due to the irregular nature of data access in the sliding window pattern, more pointers are required for data fetching, leading to increased resource consumption. Additionally, this irregular address access mode results in computational delays. When implemented on the ZYNQ platform, the processing speed of the ARM core may become a bottleneck [17].

2.2. GEMM Acceleration and im2col Method

The GEMM convolution acceleration method uses the im2col technique to store each window corresponding to a convolution kernel into RAM, thereby transforming the convolution operation into a matrix multiplication [18]. Figure 1 is a simple schematic diagram. This matrix multiplication is then processed using methods such as pulsed arrays (e.g., GOOGLE TPU) [19].

This method requires converting the weights and feature map blocks into matrix forms, an operation that is mostly performed in software and involves complex memory access patterns. Consequently, when implemented on heterogeneous platforms, the storage and retrieval of data can become a bottleneck. Additionally, the GEMM method lacks data reuse, meaning that during convolution acceleration using the GEMM method, central data is stored nine times, leading to inefficient memory utilization.

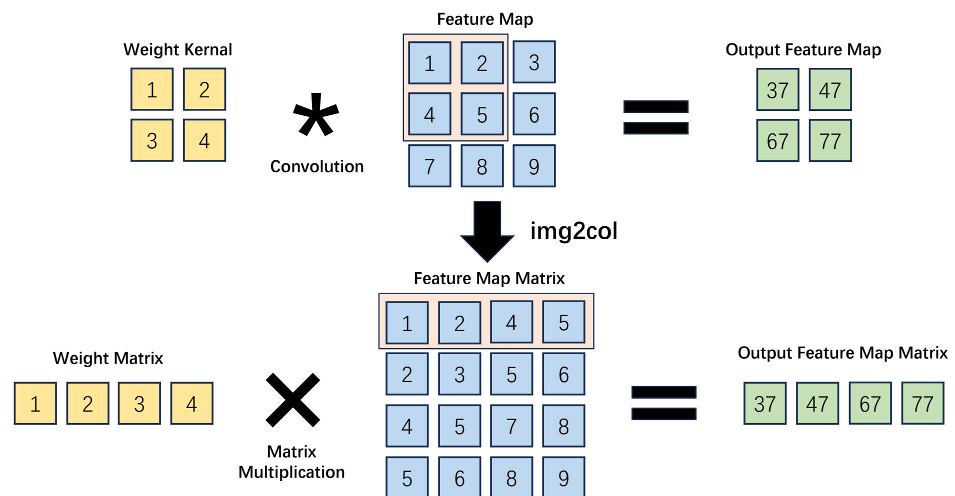

Figure 1. GEMM computation and im2col storage strategy.

2.3. Row Buffer Strategy for 2D Convolution

When performing convolution operations, if the kernel size is $K \times K$, it is necessary to ensure that $K \times K$ data points can be fed into the convolver within one cycle. However, the bandwidth between memory and the computing structure might not be sufficient to meet this requirement. Therefore, 2D convolution relies on FIFO to achieve data reuse, which reduces the bandwidth demand between the storage structure and the computing structure [20,21], enabling multiplication to be performed in one cycle. Using this method as

the data feeding structure also reduces the need for address pointer calls and computations, facilitating image traversal in a simple manner.

Take 3×3 convolution as an example. As depicted in Figure 2, on a 2D plane, the specific approach is to store the data from the first two rows into a buffer while inputting one data point at a time. Then, when the data for the third row are input, three data points are fetched from two buffers and one RAM input, and then stored in registers. After two buffer cycles, a 3×3 window can be obtained when the third data point is input. Additionally, each data point is subsequently transferred to the buffer of the previous row upon input, which is easily implemented when RAM is configured in FIFO mode.

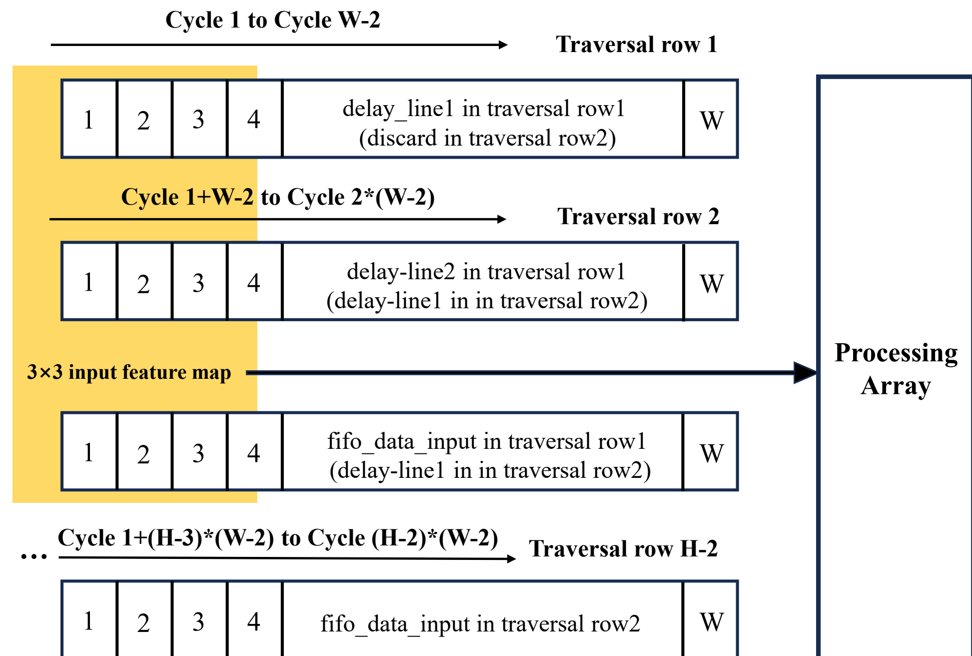

Figure 2. Line buffer strategy, used to reduce on-chip bandwidth.

Previous studies have implemented convolution operations on a 2D plane. This paper proposes an addressing strategy that extends this method to feature maps with a depth dimension. This reduces data bandwidth and avoids complex pointer calculations during the execution of depthwise convolutions.

3. Design

This section introduces a hardware design that employs a 4×4 sliding window strategy for traversal. It proposes a new surface buffer strategy and describes a corresponding address calculation method.

3.1. The 4 × 4 Sliding Window

The 4×4 sliding window structure was proposed in 2023 by reference [7]. In this structure, a $4 \times 4 \times D$ window is used as the traversal unit when performing 3×3 convolution operations. Each feature map window executes four convolution operations, effectively performing a stride-1 traversal with a stride-2 execution. This approach reduces the required traversal steps in both the height and width dimensions by 50%, allowing the traversal of the entire feature map block in just a quarter of the time.

As depicted in Figure 3, in this traversal mode, the $3 \times 3 \times D$ weights are replicated four times to align with the $4 \times 4 \times D$ feature map blocks. The central four data points are reused four times, while the peripheral (non-corner) data points are reused twice, reducing the bandwidth requirement by 55.56% (calculated as $1 - \frac{4\times4}{3\times3\times4}$). Additionally, this strategy

can seamlessly integrate with 2 × 2 pooling layers, facilitating a high-level integration of processing elements (PEs). The 4 × 4 sliding window feeds four times the data into the multiplication array in each cycle, resulting in higher DSP resource utilization.

Figure 3. Design and processing sequence of a 4 × 4 convolution window.

We have modified the PE in our method by replacing the LEAKYRELU activation unit with RELU. This change leverages the properties of two's complement, making activation calculations more convenient. Additionally, tour design adopts a 16-bit architecture and employs the LSQ (learned step size quantization) strategy to achieve higher precision [22,23]. The quantization parameters used by the LSQ method are derived from training.

As illustrated in Figure 4, the overall design of the computational units is similar to [7]. We declare $3 \times 3 \times D$ DSP multipliers to form a MAC array and then replicate this array four times to align with each 3 × 3 structure within the 4 × 4 window. In this work, we set D to 16, as most convolutional neural network depths are common multiples of 16. Each MAC array contains 144 DSPs, with each DSP multiplying 16-bit weights and feature pixels. In total, 576 DSPs are generated during synthesis. Given our ample on-chip DSP resources, there is no need to use LUT resources for fitting and mapping.

Figure 4. The adopted PE structure.

After each MAC multiplication array, an adder tree is connected to sum the computed data. Subsequently, an activation and quantization module is employed, leveraging the properties of two's complement to quantize and activate the data. After convolution, a 4×4 window is transformed into a 2×2 matrix, which can then be directly fed into the max pooling module for pooling operations.

3.2. Plane Buffer Strategy

Before introducing the surface buffer strategy for a 4×4 window, we first present the classical plane buffer strategy for a 3×3 window convolution. In the $3 \times 3 \times D$ buffering strategy, we introduce two $W \times 1 \times D$ buffers and define six registers. Using $1 \times 1 \times D$ as a data unit, we input one data unit at a time.

At the start of the convolution, $1 \times 1 \times D$ data units are sequentially extracted from RAM and input into the buffer. After the data input for the second row is complete, the data of size $W \times 2 \times D$ is stored in the plane buffer. When the third row of data is input, the corresponding data from the first two buffers is extracted and stored into registers. When traversing to the third row and third column, the six data points in the registers, the two data points in the FIFO, and the one data point from RAM are combined into a $3 \times 3 \times D$ data block and fed into the convolver.

With each data input, the $1 \times 1 \times D$ data unit fetched from RAM is written into the rear plane buffer, the data from the rear buffer is written into the front buffer, and the data from the front buffer is discarded. The buffer operates using a FIFO structure, making it easy to handle data insertion and deletion. During the traversal from the third row onward (including the third row), the total data in the buffers remain constant. After traversing each $W \times H \times D$ sized feature map, the next $W \times H \times D$ sized feature map is processed, requiring a total of $\lceil \frac{C_{in}}{D} \rceil$ such operations.

Next, we introduce the traversal strategy for the 4×4 window. As shown in Figure 5, to implement the sliding window for a 4×4 structure while reducing the bandwidth requirements between memory and computational structures, a row buffer is declared to cache the $W \times 2 \times D$ space. For the $4 \times 4 \times D$ structure, we use $2 \times 2 \times D$ as a data unit and feed it into the FIFO buffer. Two registers are declared to temporarily cache data from the previous cycle.

Similarly to the 3×3 case, at the start of the convolution, the $2 \times 2 \times D$ data units are sequentially extracted from RAM and input into the buffer. Once the traversal of the $W \times 2 \times D$ space is complete, the data from the next two rows are input as $2 \times 2 \times D$ units. Meanwhile, the corresponding column data in the buffer are extracted, aligned, and stored in the registers. This operation is repeated in the next cycle, producing $2 \times 2 \times 4 \times D$ data, which are then concatenated into $4 \times 4 \times D$ data. The 4×4 data are split into four $3 \times 3 \times D$ blocks and fed into four MAC arrays for convolution. The traversal mode can be referenced in Figure 6.

Each time a $2 \times 2 \times D$ data unit is output, it is also written into the buffer. Thus, after completing each $W \times 2 \times D$ traversal, the subsequent two rows of data can be processed seamlessly. After traversing each $W \times H \times D$ feature map, the next $W \times H \times D$ feature map is processed, requiring a total of $\lceil \frac{C_{in}}{D} \rceil$ such operations. Therefore, the step size for reading data to the right and downward is 2, while the computational step size is 1, reducing the traversal time by 75% compared to the 3×3 mode.

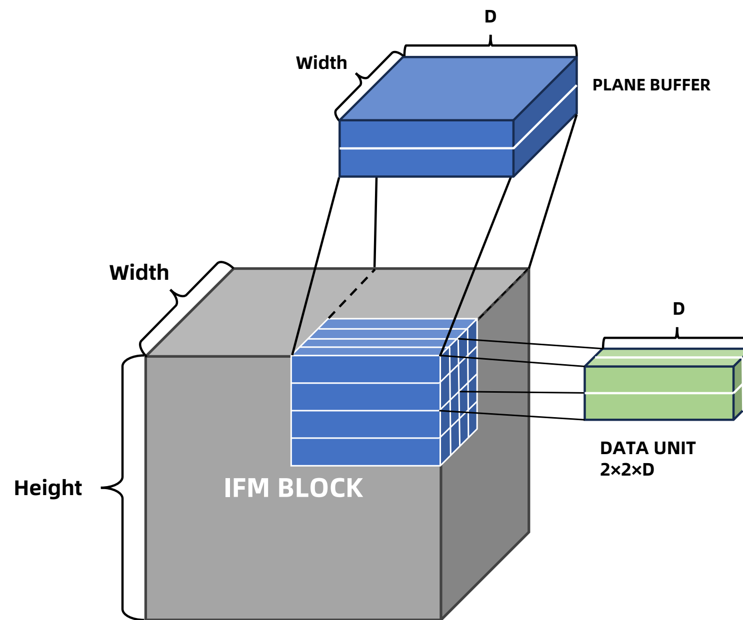

Figure 5. Buffer design of the data unit and plane Buffer.

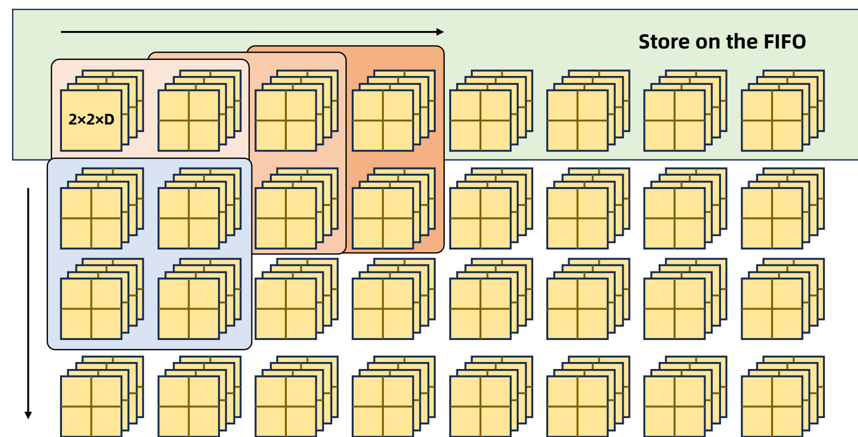

Figure 6. Implementation of the frame buffer.

3.3. Addressing and Storage Strategy

When processing data, the output order for the weight reuse strategy [10] is from width to height to depth, which is inconsistent with our data extraction sequence. Using address pointers during data extraction would result in a significant amount of computation, causing pipeline stalls. To enable burst data extraction and avoid the excessive usage of pointers during reads, we implement an addressing strategy where the pixel addresses are calculated and assigned at the end of computation. By rectifying the addresses as data are input, we can avoid address computation during data extraction, thereby enhancing the efficiency of the sliding window.

The specific address calculation strategy is as follows:

The following are given in the direction of increasing addresses:

- **Loop 1:** Increase within the same pixel up to the D-th channel.
- **Loop 2:** Traverse a 2×2 plane in a left-to-right, top-to-bottom order for $1 \times 1 \times D$.
- **Loop 3:** Traverse a $W \times H \times D$ block for $2 \times 2 \times D$.
- **Loop 4:** Traverse a $W \times H \times C_{in}$ block for $W \times H \times D$

As illustrated in the Figure 7:

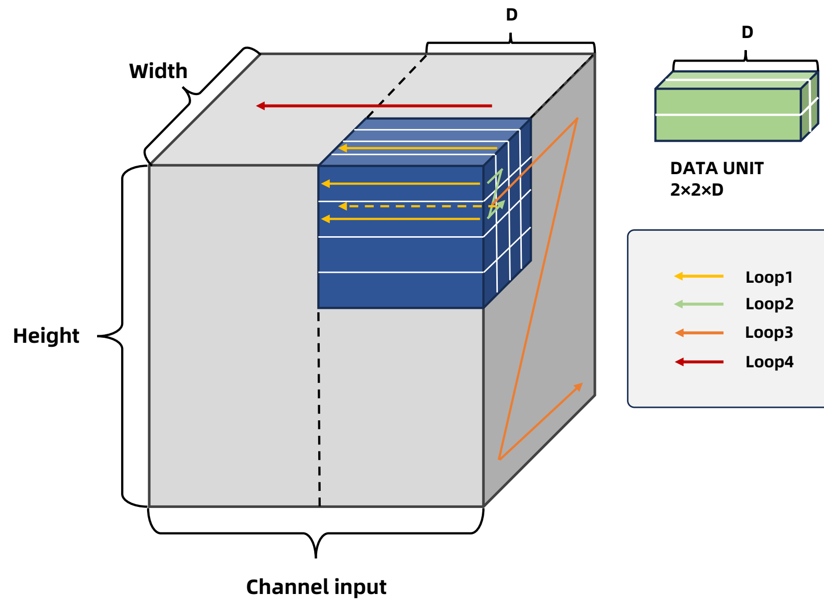

Figure 7. Addressing and storage strategy.

For different padding sizes, simply leave varying numbers of address spaces around the feature map block. The calculation mode remains the same as described.

3.4. Buffer Structure Design

To ensure the design is free from bubble cycles and pipeline stalls, the following buffer structures are necessary.

3.4.1. DDR Weight Buffer

To enable the extensive reuse of weights, feature map blocks are stored in on-chip SRAM, while weights are fetched from DDR. To feed $K \times K \times D$ weights in a single cycle, a buffer is declared to store and concatenate DDR weights, allowing simultaneous DDR weight fetching during computation. This approach ensures that weights are ready by the time that computations are performed, particularly when the feature map size exceeds a certain threshold.

The feature map size that can be hidden must satisfy the following inequality:

$$\lceil \frac{W}{2} \rceil \times \lceil \frac{H}{2} \rceil \geqslant \frac{3 \times 3 \times D \times N}{V_{data}} \tag{2}$$

where V_{data} represents the interaction rate between DDR and on-chip storage, measured in $bits/CLK$, N represents the number of bits, i.e., the precision, in the computation process. Under this condition, the computation time exceeds the weight fetching time, enabling the DDR fetching time to be effectively hidden. For D set to 16 channels and V_{data} at 32 $bits/CLK$, the minimum feature map size that can be hidden is 17×17.

Considering that larger feature maps in most convolutional neural networks require more operations, this method becomes increasingly suitable as the feature map size grows.

3.4.2. Concatenation Transfer Buffer

In our design, using $2 \times 2 \times D$ as an input corresponds to $3 \times 3 \times D$ weights. To handle this, we utilize LUTRAM to buffer and combine two data units stored on registers, one FIFO data unit, and one RAM data unit. For each pixel in a $4 \times 4 \times D$ block, we define a $D \times 16$ bit register, allowing access to 16 pixels. These pixels can then be fed into four MAC arrays.

3.5. Overall Structural Framework

In this design (Figure 8), a ping-pong operation RAM is used to perform data read and write operations. Feature maps are stored in on-chip read RAM to enable extensive weight reuse. The address calculation module computes the addresses for output data in real-time and stores the read-out data into write RAM. FIFO and registers are used for plane buffering to implement a sliding window. On the PS side, DDR reads out weights while convolution computations are performed and stores them in the weight buffer. The ARM core transmits control signals via AXI-lite at the start of computation to control parameters, such as the size of the feature map, whether to perform pooling layers, and padding size. The computation module PE consists of four MACs, an activation module, and a max pooling module. After each MAC array, an adder tree is connected to reduce the instability caused by high fan-out.

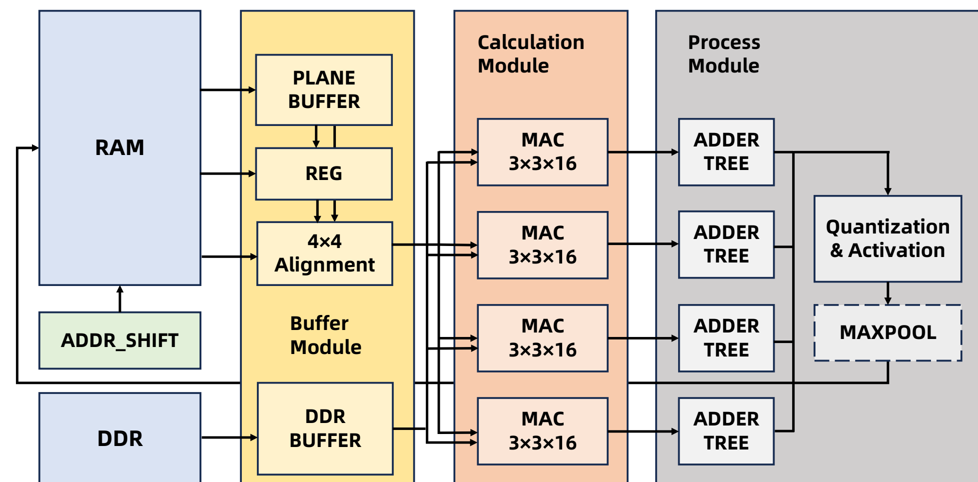

Figure 8. Overall structural framework.

It is worth noting that, if the feature map size is large, we prefer to store it in DDR. Even in this scenario, our approach can still reduce the bit-width communication between DDR and the computing structure by 75%.

4. Evaluation and Experiments

4.1. Evaluation

Firstly, we employed a 4×4 traversal mode to compute 27 pixel locations while transmitting 16 pixel locations. Compared to the traditional 3×3 traversal mode, this approach reduces the on-chip bandwidth requirement by 55.56%. Next, by utilizing plane buffers for data reuse, we decomposed the $4 \times 4 \times D$ window into a $2 \times 2 \times D \times 4$ pattern, further reducing the bandwidth requirement by 75%. Figure 9 presents a comparison of the attributes of different traversal modes.

By avoiding traversal operations with the ARM core, we eliminated the read delay caused by the ARM core's address pointer traversal. Our approach achieves extensive data reuse through the 4×4 window size transmission and line buffer scheme. Compared to the im2col operation in GEMM, our method reduces storage space usage by approximately 88.89%.

Ergodic mode	3×3×D	4×4×D	4×4×D With Buffer
Total Operand	W×H×C_{in}/D	W/2×H/2×C_{in}/D	W/2×H/2×C_{in}/D
Memory Consumption	W×H×C_{in}	W×H×C_{in}	W×H×C_{in}+2×W×D
Output Size	1	2×2 or 1 (if pooling)	2×2 or 1 (if pooling)
Bandwidth in Feeding Calculation	3×3×D×B	4×4×D×B	4×4×D×B
Bandwidth in Ergodic	3×3×D	4×4×D	2×2×D

Figure 9. Evaluation.

4.2. Performance and Resource Utilization

We deployed our design on the ZYNQ 7035 FPGA board. Using the second layer of the YOLOv3-tiny model (104 × 104 × 16) as an example, we achieved an inference speed of 121.36 GOPS with a 16-bit width. The resource utilization is shown in the Table 1.

Table 1. Resource utilization.

Resource	Estimation	Available	Utilization %
LUT	9653	171,900	5.62
LUTRAM	1024	70,400	1.45
FF	15,823	343,800	4.60
BRAM	214.50	500	42.90
DSP	576	900	64.00
BUFG	3	32	9.38
MMCM	1	8	12.50

4.3. Comparison with CPU and GPU Platforms

We also computed the second layer of the YOLOv3-tiny model on both CPU and GPU platforms to evaluate their computational efficiency. To account for the startup latency typically observed in GPU and CPU computations, we replicated the test data ten times and used the stable middle results as our computation time for a fairer comparison. The result is shown in the Table 2.

Table 2. Comparison with CPU and GPU platforms.

Device	Intel Core i5 10s	Nvidia Tesla P100	This Work
Platform type	CPU	GPU	FPGA
Precision (bit)	double (64)	double (64)	uint16 (16)
Clock frequency	2.0/3.8 GHz	1328 MHz	200 MHz
Latency	67.80 ms	0.45 ms	1.73 ms
Total clock cycles	135.6 M	597.6 K	346.8 K
GOPS	3.10 GOPS	467.64 GOPS	121.36 GOPS

According to [5], power consumption actually depends on the selected platform, making power consumption comparisons less meaningful. Our data show that our accelerator achieves a 39.10 times speedup over the CPU. While it does not outperform the Tesla P100 in terms of performance, our design operates at a clock frequency of 200 MHz, significantly lower than the P100's 1328 MHz, and occupies considerably fewer computational resources

than Tesla P100. Since lower precision does not lead to noticeable degradation [24], we quantize data to unsigned 16-bit integers for on-chip transfer, enhancing on-chip cache resource utilization efficiency compared to both CPU and GPU.

4.4. Comparison with Other Accelerators

To demonstrate the performance of our approach in convolution acceleration, we compared it with recent convolution accelerators, evaluating both performance and resource utilization.

Table 3 outlines the performance of various FPGA accelerators under different resource usage conditions. Since our design is structurally optimized for classic convolution (CONV), acceleration methods based on algorithms like Winograd and FFT are not included in the comparison. Refs. [5,6] employed the GEMM method. Adiono et al. [5] used compressed matrix storage for data, which results in higher memory usage and longer data processing times. Additionally, the DDR read time is not adequately hidden, causing pipeline stalls. Ahmad et al. [8] used a 3×3 sliding window to traverse the data, and employed a pipelined architecture to eliminate issues related to offloading data to a soft core. However, this architecture requires 2304 DSPs, making it unsuitable for deployment on smaller FPGA boards. Compared to [7], our data traversal model reduces the processing time on the soft core, achieving a 27.66% performance improvement. This design processes data layer by layer, resulting in lower LUT and FF usage compared to the implementation of an entire accelerator. This layer-by-layer approach allows the convolution layer to be invoked multiple times across different neural networks.

Table 3. Comparison with other accelerators.

Article	[1]	[3]	[4]	[2]	**Ours**
Year	2019	2020	2021	2023	2024
Platform	XC7Z020	VC707	Ultra96 V2	A7-100T	XC7Z035
CNN model	MNIST	YOLOv3-Tiny	YOLOv3-Tiny	YOLOv3-Tiny	YOLOv3-Tiny-Layer2
Clock (MHz)	N/A	200	200	100	200
LUTs	9.4 K	48.6 K	27.3 K	50.2 K	9.6 K
FFs	35.1 K	93.2 K	38.5 K	58.1 K	15.86 K
BRAMs	9	141 (18 K)	248	185	214.50
DSPs	168	2304	242	240	576

It is worth noting that our work primarily focuses on optimization at the resource scheduling level. Therefore, performance improvements are mainly derived from preventing data read times from becoming a computational bottleneck, without using additional LUTs and DSPs. Compared to GEMM-based convolution accelerators, our architecture is more advantageous for storing larger and deeper feature map tiles directly in on-chip RAM, without excessive resource consumption. This makes it more suitable for implementing small-to-medium-sized convolutional networks.

5. Conclusions

In this paper, we first made several improvements based on a 4×4 sliding window architecture, including enhancing the precision to 16 bits and utilizing the properties of two's complement for activation. We proposed a new plane buffer structure, expanding the original 2D filter into the depthwise convolution direction, achieving a 75% performance improvement. To accommodate the new buffer structure, we designed a new address

computation method for the 4 × 4 window mode, allowing address computation and data storage to be performed in parallel to avoid performance bottlenecks during data loading. This work was tested on YOLOv3-Tiny-Layer2, achieving a peak processing speed of 121.36 GOPS at a clock frequency of 200 MHz. Future work can leverage the proposed method to change the feature map storage from RAM to DDR, enabling the extension to larger feature maps or higher-dimensional convolutions. In this case, the method can effectively reduce the DDR read/write bandwidth requirements as well.

Author Contributions: Conceptualization, Y.S.; methodology, Y.S. and Y.M.; coding, Y.S. and Y.M.; validation, Y.S. and Y.M.; formal analysis, Y.S. and B.C.; investigation, Y.S. and Z.L.; resources, Y.S. and R.S.; data curation, Y.S. and B.C.; writing—original draft preparation, Y.S.; writing—review and editing, Y.S., Y.M. and Z.C.; visualization, Y.S. and Z.L.; supervision, Y.S. and R.S.; project administration, Y.S.; funding acquisition, Y.S. and R.S. All authors have read and agreed to the published version of the manuscript.

Funding: Supported by "the Fundamental Research Funds for the Central Universities" Project ID: XD2024101098.

Data Availability Statement: Data are contained within the article.

Acknowledgments: Thanks to the reviewers and editor from the journal.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. 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 (FPGA '15), Monterey, CA, USA, 22–24 February 2015; Association for Computing Machinery: New York, NY, USA, 2015; pp. 161–170. [CrossRef]
2. Liu, Z.; Liu, Q.; Yan, S.; Cheung, R.C.C. An Efficient FPGA-based Depthwise Separable Convolutional Neural Network Accelerator with Hardware Pruning. *ACM Trans. Reconfig. Technol. Syst.* **2024**, *17*, 15. [CrossRef]
3. Liu, F.; Li, H.; Hu, W.; He, Y. Review of neural network model acceleration techniques based on FPGA platforms. *Neurocomputing* **2024**, *610*, 128511. [CrossRef]
4. Yang, C.; Wang, Y.; Wang, X.; Geng, L. A Stride-Based Convolution Decomposition Method to Stretch CNN Acceleration Algorithms for Efficient and Flexible Hardware Implementation. *IEEE Trans. Circuits Syst. I Regul. Pap.* **2020**, *67*, 3007–3020. [CrossRef]
5. Adiono, T.; Putra, A.; Sutisna, N.; Syafalni, I.; Mulyawan, R. Low Latency YOLOv3-Tiny Accelerator for Low-Cost FPGA Using General Matrix Multiplication Principle. *IEEE Access* **2021**, *9*, 141890–141913. [CrossRef]
6. Sudrajat, M.R.D.; Adiono, T.; Syafalni, I. GEMM-Based Quantized Neural Network FPGA Accelerator Design. In Proceedings of the 2019 International Symposium on Electronics and Smart Devices (ISESD), Badung, Indonesia, 8–9 October 2019; pp. 1–5. [CrossRef]
7. Kim, M.; Oh, K.; Cho, Y.; Seo, H.; Nguyen, X.T.; Lee, H.-J. A Low-Latency FPGA Accelerator for YOLOv3-Tiny With Flexible Layerwise Mapping and Dataflow. *IEEE Trans. Circuits Syst. I Regul. Pap.* **2023**, *71*, 1158–1171. [CrossRef]
8. Ahmad, A.; Pasha, M.A.; Raza, G.J. Accelerating Tiny YOLOv3 using FPGA-Based Hardware/Software Co-Design. In Proceedings of the 2020 IEEE International Symposium on Circuits and Systems (ISCAS), Seville, Spain, 12–14 October 2020; pp. 1–5. [CrossRef]
9. Bai, Z.; Fan, H.; Liu, L.; Liu, L.; Wang, D. An OpenCL-Based FPGA Accelerator with the Winograd's Minimal Filtering Algorithm for Convolution Neuron Networks. In Proceedings of the 2019 IEEE 5th International Conference on Computer and Communications (ICCC), Chengdu, China, 6–9 December 2019; pp. 277–282. [CrossRef]
10. Nguyen, D.T.; Nguyen, T.N.; Kim, H.; Lee, H.-J. 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]
11. Chen, Y.; Luo, T.; Liu, S.; Zhang, S.; He, L.; Wang, J.; Li, L.; Chen, T.; Xu, Z.; Sun, N.; et al. DaDianNao: A Machine-Learning Supercomputer. In Proceedings of the 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture, Cambridge, UK, 13–17 December 2014; pp. 609–622. [CrossRef]
12. Li, H.; Fan, X.; Jiao, L.; Cao, W.; Zhou, X.; Wang, L. A high performance FPGA-based accelerator for large-scale convolutional neural networks. In Proceedings of the 2016 26th International Conference on Field Programmable Logic and Applications (FPL), Lausanne, Switzerland, 29 August–2 September 2016; pp. 1–9. [CrossRef]

13. Bai, L.; Zhao, Y.; Huang, X. A CNN Accelerator on FPGA Using Depthwise Separable Convolution. *IEEE Trans. Circuits Syst. II Express Briefs* **2018**, *65*, 1415–1419. [CrossRef]
14. Sun, F.; Wang, C.; Gong, L.; Xu, C.; Zhang, Y.; Lu, Y.; 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, 12–15 December 2017; pp. 622–629. [CrossRef]
15. Srilakshmi, S.; Madhumati, G.L. A Comparative Analysis of HDL and HLS for Developing CNN Accelerators. In Proceedings of the 2023 Third International Conference on Artificial Intelligence and Smart Energy (ICAIS), Coimbatore, India, 2–4 February 2023; pp. 1060–1065. [CrossRef]
16. Ma, Y.; Suda, N.; Cao, Y.; Seo, J.-S.; Vrudhula, S. Scalable and modularized RTL compilation of Convolutional Neural Networks onto FPGA. In Proceedings of the 2016 26th International Conference on Field Programmable Logic and Applications (FPL), Lausanne, Switzerland, 29 August–2 September 2016; pp. 1–8. [CrossRef]
17. Wang, B.; Li, M. A Structure to Effectively Prepare the Data for Sliding Window in Deep Learning. In Proceedings of the 2021 IEEE 6th International Conference on Signal and Image Processing (ICSIP), Nanjing, China, 22–24 October 2021; pp. 1025–1028. [CrossRef]
18. Chellapilla, K.; Puri, S.; Simard, P. High Performance Convolutional Neural Networks for Document Processing. Tenth International Workshop on Frontiers in Handwriting Recognition, Université de Rennes 1, 2006, La Baule (France). inria-00112631. Available online: https://inria.hal.science/inria-00112631v1/document (accessed on 29 January 2025).
19. Jouppi, N.P.; Young, C.; Patil, N.; Patterson, D.; Agrawal, G.; Bajwa, R.; Bates, S.; Bhatia, S.; Boden, N.; Borchers, A.; et al. In-Datacenter Performance Analysis of a Tensor Processing Unit. In Proceedings of the 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA), Toronto, ON, Canada, 24–28 June 2017; pp. 1–12. [CrossRef]
20. Zhang, H.; Xia, M.; Hu, G. A Multiwindow Partial Buffering Scheme for FPGA-Based 2-D Convolvers. *IEEE Trans. Circuits Syst. II Express Briefs* **2007**, *54*, 200–204. [CrossRef]
21. Bosi, B.; Bois, G.; Savaria, Y. Reconfigurable pipelined 2-D convolvers for fast digital signal processing. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **1999**, *7*, 299–308. [CrossRef]
22. Nagel, M.; Fournarakis, M.; Amjad, R.A.; Bondarenko, Y.; van Baalen, M.; Blankevoort, T. A White Paper on Neural Network Quantization. *arXiv* **2021**, arXiv:2106.08295.
23. Esser, S.K.; McKinstry, J.L.; Bablani, D.; Appuswamy, R.; Modha, D.S. Learned Step Size Quantization. *arXiv* **2019**, arXiv:1902.08153.
24. Courbariaux, M.; Bengio, Y.; David, J.-P. Training deep neural networks with low precision multiplications. *arXiv* **2015**, arXiv:1412.7024v5.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.