International Journal of Engineering Research And Management (IJERM)
ISSN: 2349- 2058, Volume-11, Issue-04, April 2024

Design and Implementation of Systolic Array-based Accelerator for Convolutional Neural Networks

Yao Dai, Lingchao Bu, Mingwei Xu

***Abstract*— CNN models are prevalent deep learning models utilized in the fields of machine learning. However, their implementation in hardware has encountered challenges, including high computational complexity, large storage requirements, memory bandwidth limitations, and difficulties with parallel computing. This study introduces a dedicated hardware accelerator designed to enhance the performance of convolutional neural networks, leveraging efficient computational units and memory hierarchy to attain accelerated processing. Initially proposed is an efficient and straightforward Img2Col method, through which convolutions can be unfolded into matrix computations from small size 3×3 to more extensive 16×16. A customizable systolic array is subsequently designed to support the acceleration convolutions. Our well-designed accelerator has been implemented utilizing the hardware description language SpinalHDL and tested on the ZYNQ. The experimental results demonstrate that our accelerator showcases remarkable performances in both CNN and GEMM calculations, delivering up to 37.6 GOPS/W.**

***Index Terms*—Gemm, FPGA, Convolution, Systolic Array**

I. INTRODUCTION

Nowadays, deep learning[1][2]has made significant progress in fields such as speech recognition, image recognition, and natural language processing. Among them, Convolutional Neural Networks (CNN)[3] are commonly used deep learning algorithms that can effectively extract detailed features from images, rendering them extensively implemented in computer vision. Compared to CNN, Transformer exhibits several advantages in computer vision tasks, including the ability to model global visual information, powerful feature representation, scalability and generalization, multi-scale processing. The core idea of the Transformer[5] is to utilize self-attention mechanisms to model sequential data without relying on recurrence or convolutional operations. The self-attention mechanism allows the model to consider all positions in the sequence simultaneously, dynamically assigning different weights to each position based on the context.

The performance of CNN models is limited by the low throughput and computational power of CPU, as well as the low energy efficiency of GPU. Although Application-Specific Integrated Circuit (ASIC) can achieve low latency and high efficiency, they have poor reconfigurability. When there is a need to update the design, ASIC cannot be reconfigured. In contrast, Field-Programmable Gate Array (FPGA) are highly programmable and can be customized flexibly according to specific computational requirements. By writing Hardware Description Language (HDL) code on FPGA, hardware acceleration for specific algorithms and applications can be achieved.

Currently, numerous hardware accelerators for CNN and GEMM have been proposed. However, due to significant differences in their computations, most current accelerators are designed to accelerate either CNN or GEMM separately. This raises a question: Can the acceleration of CNN and GEMM be unified using FPGA? So we propose to design and implement a novel hardware accelerator that can maximize the efficiency and performance of CNN and GEMM.

Our contributions in this paper are the following：

(1) A detailed Img2Col method is introduced to effectively map convolution operations to GEMM (General Matrix Multiplication) operations on the systolic array. enabling the acceleration of convolutions ranging in size from 3×3 to 16×16.

(2) An efficient systolic array architecture with two different computing modes is proposed to support convolution and matrix multiplication computations.

(3) The proposed accelerator demonstrates remarkable performance in both CNN (Convolutional Neural Network) and GEMM (General Matrix Multiplication) calculations, achieving an impressive efficiency of up to 37.6 GOPS/W (Giga Operations Per Second per Watt).

II. RELATED WORKS

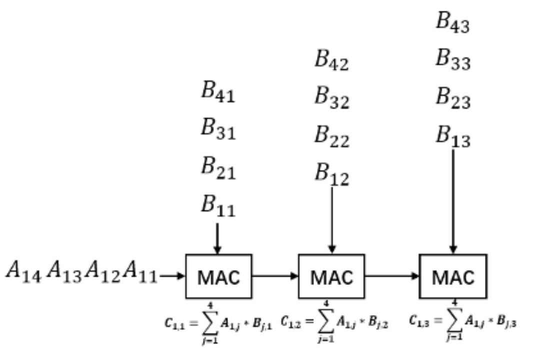
A. Systolic Array

A systolic array[4]represents a computational architecture characterized by a grid of processing elements (PE) interconnected in a systematic and structured manner. It is specifically designed to efficiently execute data-parallel algorithms by enabling data to flow through the array in a systolic manner, where each processing element operates on a portion of the data and passes it along to the next element in a pipelined fashion. Within a systolic array, the processing elements are typically organized in rows and columns,

Manuscript received April 03, 2024

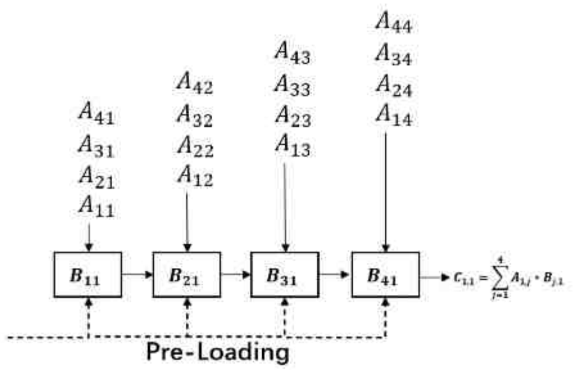
Yao Dai, School of computer science and technology, Tiangong University, Tianjin, China
Lingchao Bu, School of computer science and technology, Tiangong University, Tianjin, China.
Mingwei Xu, School of computer science and technology, Tiangong University, Tianjin, China

forming a grid-like structure. Data is fed into the array and propagated through the PE in a synchronized and regular manner. The processing elements operate in a tightly-coupled fashion, performing computations and passing results to adjacent elements. This regular and efficient data flow enables high throughput and low-latency computations.



B_{41} B_{31} B_{21} B_{11}
B_{42} B_{32} B_{22} B_{12}
B_{43} B_{33} B_{23} B_{13}
$A_{14}A_{13}A_{12}A_{11}$
MAC
MAC
MAC
$C_{1,1}=\sum_{j=1}^{4}A_{1,j}*B_{j,1}$
$C_{1,2}=\sum_{j=1}^{4}A_{1,j}*B_{j,2}$
$C_{1,3}=\sum_{j=1}^{4}A_{1,j}*B_{j,3}$

Figure 1 Output-Stationary One-Dimensional Systolic Array



A_{41} A_{31} A_{21} A_{11}
A_{42} A_{32} A_{22} A_{12}
A_{43} A_{33} A_{23} A_{13}
A_{44} A_{34} A_{24} A_{14}
B_{11}
B_{21}
B_{31}
B_{41}
$C_{1,1}=\sum_{j=1}^{4}A_{1,j}*B_{j,1}$
Pre-Loading

Figure 2 Weight-Stationary One-Dimensional Systolic Array

Systolic arrays generally come in two forms: output-stationary and weight-stationary, as depicted in Figure 1 and Figure 2. For weight-stationary SA, weight parameters are preloaded into the processing elements (PE) while for output- stationary SA, the input data x moves in a consistent manner, shifting by one at each time step. However, unlike the weight-stationary SA, the weight parameters are not preloaded and stored statically, instead, they are streamed into the PE similar to the input data.

Systolic array computing provides high-speed and efficient computational capability through parallel computing. It accelerates the processing speed of computational tasks, improves computational efficiency, and enhances energy efficiency. Systolic arrays exhibit excellent scalability and are applicable to various domains. They are particularly significant for handling large-scale data and complex models.

B. *GEMM*

General Matrix Multiplication (GEMM) involves transforming both the weights and feature data of convolutional calculations into matrix form using the img2col method[6]. The convolution operation is then converted into a matrix multiplication operation, benefiting from the well-established optimization techniques in the field of linear algebra for matrix multiplication. Therefore, utilizing GEMM for convolution operations offers significant advantages in terms of logical implementation.

When performing convolution calculations using GEMM, the convolution kernel slides along the rows and columns of the input feature map based on the specified stride. The img2col method stores the feature data within the sliding window in sequential order in a row of the feature map matrix. Each sliding window operation generates a row of the feature map matrix. The convolution kernel is stored in the weight matrix in the corresponding order to the feature data, with multiple convolution kernels sequentially arranged in the columns of the weight matrix. The matrix multiplication operation is then performed between the feature map matrix and the weight matrix, transforming the convolution operation into a matrix multiplication operation. Therefore, efficient implementation of matrix operations leads to efficient convolution calculations.

Both CPUs and GPUs provide libraries dedicated to matrix multiplication in their underlying logic, enabling efficient matrix computations. Hence, machine learning frameworks such as Caffe[7] and MXNet[8] utilize the img2col+GEMM method for convolution calculations on CPUs and GPUs. However, FPGAs do not offer built-in libraries for matrix multiplication, requiring custom implementation of all matrix operation logic. This undoubtedly increases the deployment complexity on FPGAs. Furthermore, implementing convolution calculations using the img2col+GEMM method on FPGAs requires abundant on-chip storage and computational resources, which are scarce on FPGAs. Therefore, employing GEMM for convolution calculations on FPGAs poses a significant challenge.

III. Architecture

A. *Overview Architecture*

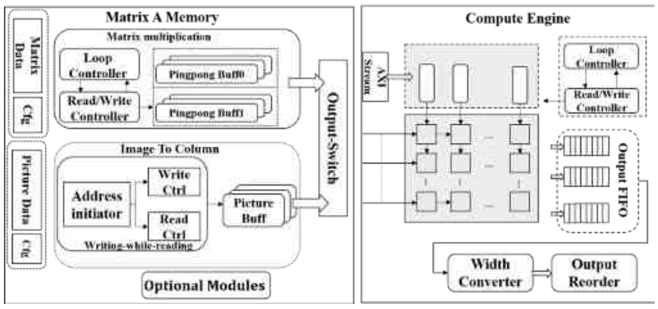


Figure 3 Overview Architecture

Figure 3 represents the overall architecture of the hardware accelerator, which includes an 8×8 systolic array. The architecture consists of two computation modes: Switch Mode for mapping convolutions to matrix multiplications, and Direct Mode for handling general matrix multiplication. The computational structure of the systolic array remains the same in both modes, with the only difference being the input format of matrix.

B. *Design and Implementation of Systolic Array*

This article presents the design of an efficient array architecture, known as a systolic array, which adopts the Output Stationary (OS) computing mode. In this mode, each processing element (PE) completes one multiply-accumulate operation per cycle and stores the partial sum results until the accumulation is completed, and only then outputs the final result. The systolic array serves as the core computational engine of an accelerator and includes modules for input data arrangement, input data buffering, and output data delay. Its main purpose is to accelerate matrix multiplication operations.

Figure 4 illustrates a 3x3 output stationary systolic array architecture. The array consists of interconnected PEs, with each PE performing the basic multiply-accumulate operation. Each PE includes additional registers, such as R_1 for storing partial sum results, R_2 for storing B matrix data, R_3 for storing A matrix data, and a selector that uses a Valid signal to determine whether accumulation is required. These added modules together form a processing element (PE) that combines storage and computation.

To ensure that each point in a row of matrix A corresponds to the corresponding point in a column of matrix B and is processed by the correct PE in a specific cycle, the input data for the i-th row of the systolic array must be advanced by i+1 rows for one cycle in advance. Similarly, the input data for the j-th column of the systolic array must be advanced by j+1 columns for one cycle in advance. Additionally, since matrix A flows from left to right in the systolic array, and matrix B flows from top to bottom, each PE is connected in both horizontal and vertical directions, forming an output stationary systolic array. This way, the data from matrix A enters the array from the left side and the data from matrix B enters from the top, and each data point in matrices A and B is reused three times.

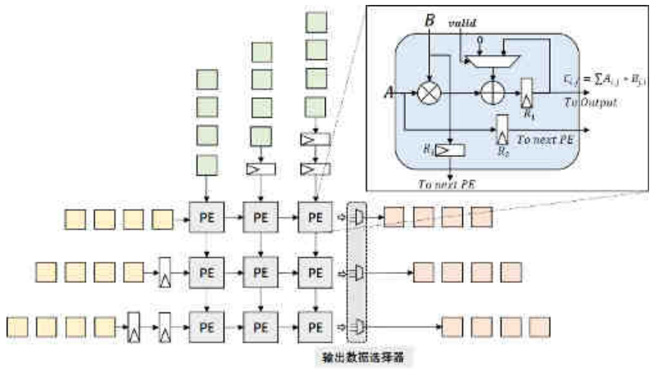


Figure 4 Architecture of systolic array and processing elements

Due to the flow of data from one PE to the adjacent PE in the systolic array, rather than broadcasting to all PEs simultaneously, the result of the matrix multiplication, C, is not obtained instantaneously. As shown in Figure 4, when the top-left PE completes its computation, the other PEs have not finished their accumulation and computation. Therefore, the output data selector only outputs the computation result from the top-left PE. In the next clock cycle, the PEs in the first row, second column, and second row, first column simultaneously complete their computations. The output data selector selects the computation results from these two PEs for output. This process continues, and it takes a total of 5 cycles from the completion of the computation in the top-left PE to the completion of the entire computation for matrix C. Taking into account the time required for data input, which is the time for the last data point of matrix A or matrix B to reach the bottom-right PE, we can determine the total computation time for the matrix multiplication.

Furthermore, due to the fixed size of the systolic array, the systolic array shown in Figure 4 can only compute three rows of matrix A and three columns of matrix B at a time, resulting in a 3x3 submatrix of matrix C. Further analysis reveals that a systolic array of size $[SA_h, SA_w]$ (Height or Width of Systolic Array) can only support computations for SA_h rows of matrix A and SA_w columns of matrix B at a time. Therefore, when the number of rows in matrix A or the number of columns in matrix B exceeds the supported range of the systolic array, the matrix multiplication needs to be performed in a blocked computation manner.

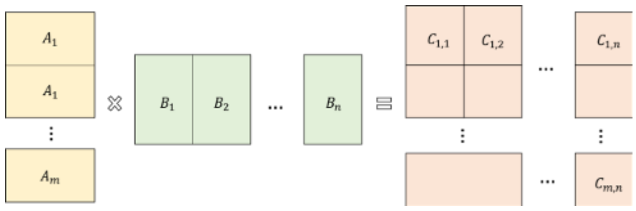


Figure 5 Matrix block computation diagram.

Figure 5 illustrates the blocked computation mode for matrix multiplication A×B=C, which involves two levels of looping.

C. The relationship between convolution and matrix multiplication

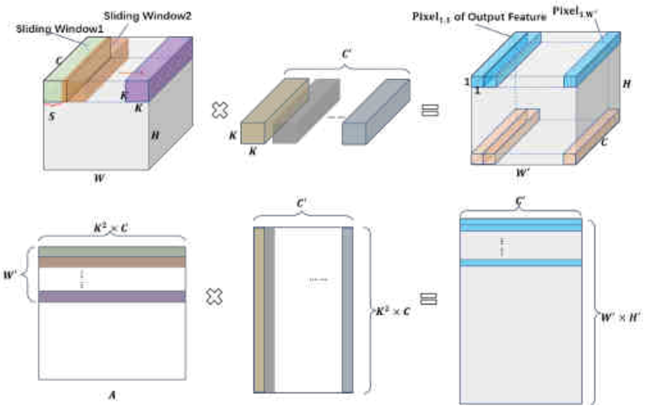


Figure 6 CNN mapped to matrix multiplication.

Let the dimension of the input feature map be $[B, H, W, C]$ (assuming batch size is 1, shown as Fig. 8) , the dimension of the convolution weights be $[C', H', W', C]$ and the dimension of the output feature map be $[B, H', W', C']$, then the input feature map and convolution weights can be expanded into matrix $A, A \in R^{(W' \times H') \times (K^2 \times C)}$ and matrix B, $B \in R^{(K^2 \times C) \times C'}$ respectively. In addition, the final output feature map can also be represented by matrix $C, C \in R^{(W' \times H') \times C'}$, which is the result of matrix multiplication: $A \times B$. Since the image storage format is $[B, H, W, C]$, the weight of the convolution kernel and each sliding window will be expanded along the dimension of the channel, as shown in Fig. 9. By corresponding the three-dimensional convolution calculation with the expanded matrix multiplication, it is obvious that each row of matrix A represents the sliding window flattened in the input feature map, each column of matrix B corresponds to the convolutional weight of each output channel flattened, and in addition, the row of matrix C corresponds to all channels of each pixel in the output feature map. For instance, the Embedding layer of VIT cuts the picture into 14×14 subgraphs through a 16×16 convolution with a step size of 16, and each subgraph will be input into the encoder as a token after flattening, where each token corresponds to each row in matrix C.

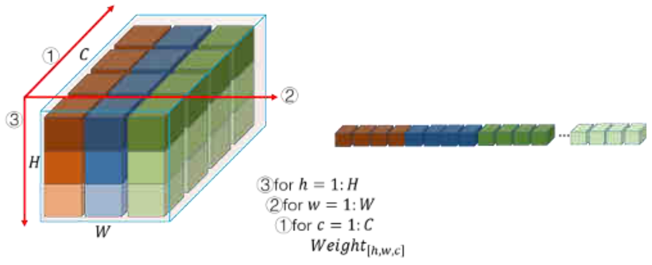


Figure 7 A 3-D tensor is flattened along the channel dimension.

When CNN is mapped to matrix multiplication, each row of matrix A is the flattened convolution sliding window, and the i_{th} row of input matrix $I_{h\times W}$ is the input to the $i \, mod \, N, (i = 1,2 \dots h)$ row of the $N \times M$ systolic array. As a result, when the convolution is mapped to a matrix multiplication on the systolic array, partial sums are accumulated within each PE while the weights and activations are flowing in the systolic array, whose benefit is that the weights and activations are multiplexed for M and N times respectively, so as that the parallel calculation of the data corresponding to N sliding windows and the data of M convolution kernel weights can be realized.

D. *Data storage and access method*

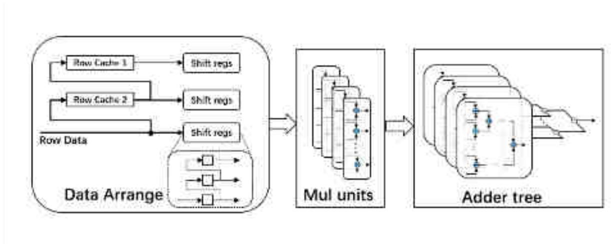


Figure 8 A commonly used hardware architecture for convolutional sliding window.

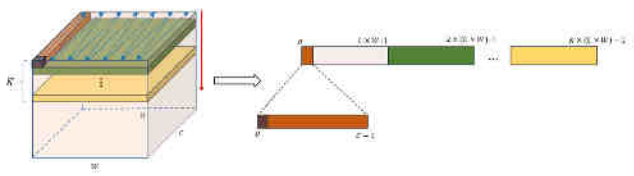


Figure 9 Storage format

The most complicated part of convolution computation is the overlap of sliding windows. As shown in Figure 9, in order to realize overlapping convolution sliding windows on FPGA, horizontal and vertical sliding of convolution is generally realized with $(K-1)$ end-to-head row caches and K shift registers for $K \times K$ convolution, as shown in Figure 8, which is simple to implement but is of low flexibility. To this, this section will introduce a kind of different data storage format and way to access data.

To enhance the computational flexibility and accommodate the calculations of the systolic array, let's consider the convolution operation with a kernel size of $K \times K$, a step size of S (where $S \leq K$), and an input channel count of C. As shown in Figure 9, taking the first K lines of the image, cached in the $[H, W, C]$ format, as an example, we can access the pixel $P_{(0,0,0)}$of the first channel in the first column and first row of the image (depicted as the black cube in the figure) by configuring the read address to 0. Similarly, to retrieve the pixel $P_{i,j,k}$, the following address can be set to $i \times C \times W + j \times C + k - 1$ or Row_Base_Addr + Col_Base_Addr + Channel_Index - 1 to read the pixel $P_{i,j,k}$. It is evident that the horizontal movement of the sliding window can be accomplished by manipulating the value of Col_Base_Addr. For instance, to access the first pixel of the second sliding window (also considered as the first point in the second row of matrix A), denoted as $P_{0,S,0}$, the read address should be set to $0 + S \times C - 1$. Similarly, the vertical displacement of the sliding window can be controlled by adjusting the value of Row_Base_Addr.

Before commencing the computation, it is imperative to cache the initial K rows of the image data. Moreover, to obviate any disruption to the flow of computation, the ensuing S rows of data can be cached concomitantly with the computation. Therefore, the minimum storage capacity required should be $(K + S) \times W \times C$.

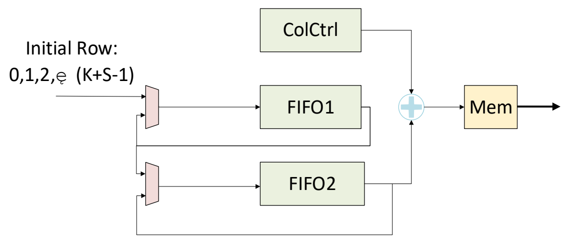


Figure 10 Dual FIFOs for the vertical and horizontal sliding of the sliding window.

Owing to the restricted capacity of the on-chip memory, in order to optimize the repurposing of this storage area for vertical sliding of the sliding window, a pair of FIFOs (shown as Figure 10) have been implemented to cyclically update the row base address. The cache is evenly divided into K+S block storage space for storing K+S rows of the input image. At the same time, FIFO1 caches all *Row_Base_Adrs* required for every computation, which are given by the set $\{i \times W \times C | 0 \leq i \leq K + S - 1\}$, while FIFO2 selectively caches K Row_Base_Addrs. During the processing of row 0 to k-1, FIFO2 only needs to cyclically retrieve Row_Base_Addr corresponding to these rows from its cache.

E. *Systolic Array Dataflow Based on Switch Mode*

Considering the strict formatting requirements of the systolic array for input data and the fact that the input data for the i-th $(1 \leq i \leq N-1)$ row of the array is delayed by one clock cycle compared to the (i-1)th row, it is necessary to delay the input data to meet the array's input requirements. For matrix multiplication, the matrix data can be directly input to the systolic array after being delayed by registers for the appropriate amount of time. However, for convolution, selectors are needed to choose specific rows of the systolic array as inputs.

To illustrate this, let's discuss the data flow format between the Img2Col module and the systolic array with the addition of selectors. For a systolic array with dimensions $[SA_h, SA_w]$, if the i-th row of the array is prepared with SA_w inputs every i-th cycle (where $i = 0,1,2,\dots,N-1$), it can ensure that $SA_h \times SA_w$ PEs can perform one valid multiply-accumulate operation every clock cycle after SA_w cycles.

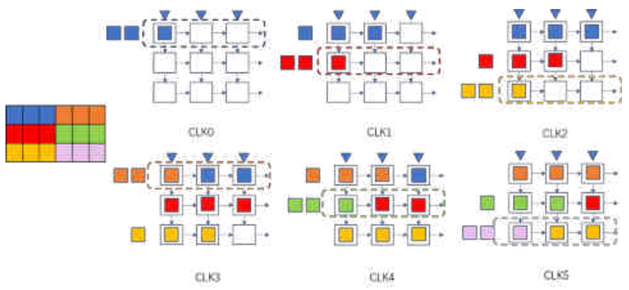


Figure 11 Data Flow of Switch-Based Systolic Array

As shown in Figure 11, the dimensions of the systolic array are 3×3, the input matrix A has dimensions 3×6, and the input matrix B has dimensions $6 \times m$ $(m \geq 1)$. In the first cycle, we only need to fetch the first three points of the first row of matrix A and convert the parallel data into three valid continuous data in a serial manner when inputting to the first row of the systolic array. In the second cycle, the first three points of the second row of matrix A (highlighted in red) are converted into serial data and input to the second row of the systolic array. This process continues, and when the fourth cycle arrives, the top-left PE of the systolic array has consumed three valid data points from matrix A. At this point, we only need to feed the last three data points of the first row of matrix A to the first row of the array to ensure that the PE's calculations are valid and the data flow is uninterrupted. This process continues...

It is evident that after the fifth cycle (clk4), each PE in the systolic array can perform one calculation per clock cycle, and each input is reused $min(m, SA_w)$ times.

F. *Computational Complexity Analysis*

For an systolic array of size $SA_h \times SA_w$, the systolic is capable of calculating the NM submatrix in output matrix C every $(C \times K^2 + SA_w - 1)$ cycles. As a result, based on the current computing architecture, the computational cost of the convolution shown in Figure 1, when mapped to the systolic array, can be expressed as follows:

$$\left\lceil \frac{H'}{SA_h} \right\rceil \times \left\lceil \frac{W'}{SA_w} \right\rceil \times (C \times K^2 + SA_w - 1)\#(1.)$$

G. *Storage Space Analysis*

To implement the convolutional sliding window, commonly applied architecture requires a minimum storage space of $(K-1)W \times C$, wherein head-to-end row cache is employed for vertical sliding. While its storage space is comparatively smaller than the proposed systolic array architecture, it falls short in terms of adaptability. For instance, the systolic array architecture can effortlessly accommodate input channels of 3 × 3 or 4 × 4 convolution for 5×5 convolution without necessitating any supplementary step processing modules, thereby, making it worth increasing the storage space by a small margin to attain greater flexibility.

IV. Experiment And Result

In this section, we will present the outcomes of a matrix computation conducted on systolic arrays. The size of the systolic array is set to be 8×8 and the power consumption of the systolic arrays is 0.67w, running 200MHZ on the Zedboard(xc7z020)[9].

Table 1 Hardware utilization of several crucial modules.

	LUT	FF	BRAM	DSP
Available	277400	554800	755	2020
8x8 SA	5356	41909	0	64
Img2Col	969	503	38	0
GEMM_Cache	614	228	32	0
Weight_Cache(8×8)	407	147	32	0
Output Arrange	953	838	32	0

As shown in Table 1, the systolic array is the main module that occupies most resources. This is because the data needs to be managed accurately, and the data needs to flow continuously in the systolic array in an orderly manner, accumulate in each PE, and output the correct result at the right time, as a result, this part requires more logical resources. For the resource usage of the on-chip memory, the cache space is occupied by three cache modules at the periphery of the array: GEMM_Cache in Direct mode for matrix multiplication calculation, Img2col module for convolutional mapping, and WeightCache module for weight caching. These cache modules efficiently store data on the chip, rapidly supplying data to the systolic array and minimizing the additional overhead of frequent data retrieval from external storage. Of course, if the size of the picture is larger and there are more input and output channels, the overhead of storage resources will increase. In addition, it is also necessary to simply rearrange the output data of the pulsating array: the general idea is to collect N rows of data and then output these collected data line by line, so as to meet the data format required for the next calculation.

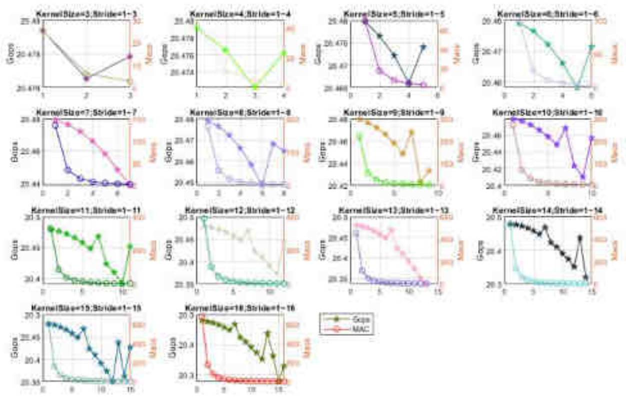


Figure 12 Support for convolution with different sizes and strides.

In addition, to verify the support for convolutions, multiple tests on convolutions ranging from 3x3 to 16x16 were conducted, examining their performance with different strides. The input image size was set to 480, with 8 input channels and 256 output channels. By appropriately configuring the register file, the accelerator efficiently completes the computation tasks as shown in Figure 12. From the graph, it can be observed that as the size of the convolutional kernel increases, the computational workload also increases, ranging from 26G for a 3×3 convolution to 680G for a 16×16 convolution. Additionally, with an increased stride, the computational workload for the same-sized convolution decreases.

Experimental results demonstrate that, the Img2Col module can map convolutions of different sizes and strides to matrix multiplications on the systolic array. Whether it is a small-sized 3×3 convolution or a larger 16×16 convolution, the accelerator efficiently completes the computational tasks without significant performance degradation caused by variations in the convolution kernel size and stride.

Furthermore, the proposed systolic array not only facilitates convolution mapping to matrix calculations but also supports basic matrix computations. Table 2 demonstrates the execution of matrix multiplications with different dimensions using an 8 × 8 systolic array. It is worth mentioning that matrix A is cached in a row-by-row fashion, allowing for flexible expansion of its row count up to 10000. However, due to resource limitations on the chip, if matrix B becomes excessively large, it needs to be partitioned. In such cases, each resulting submatrix is multiplied with matrix A to obtain the final solution.

Table 2 Performance of Matrix multiplication on systolic array

MatrixA	MatrixB	Compute Time(ms)
[4096,256]	[256,256]	21.7
[8192,512]	[512,256]	85.6
[1024,256]	[256,256]	5.41
[2048,1024]	[1024,128]	21.3
[4096,512]	[512,256]	42.62
[197,384]	[384,192]	1.16
[10000,384]	[384,192]	58.73

V. Conclusion

This study aims to introduce a hardware accelerator that can enhance the performance of CNN models by leveraging efficient computational units and memory hierarchy to attain accelerated processing. The implementation of CNN models in hardware has encountered several challenges, including high computational complexity, large storage requirements, memory bandwidth limitations, and difficulties with parallel computing. To address these challenges, an efficient and straightforward Img2Col method was proposed through which convolutions can be unfolded into matrix computations from small size 1×1 to more extensive 16×16.

The experiment results demonstrated that the accelerator delivered remarkable performances in both CNN calculations, delivering up to 37.6 GOPS/W. This study's contribution is significant in enhancing the performance of CNN and by addressing the challenges encountered in their hardware implementation. The hardware accelerator can be employed in various machine learning and natural language processing applications to enhance their speed and efficiency. Further research can explore the scalability of the hardware accelerator to handle more complex tasks and models.

References

[1] LeCun Y, Bengio Y, Hinton G. Deep learning[J]. nature, 2015, 521(7553): 436-444.
[2] Menghani G. Efficient deep learning: A survey on making deep learning models smaller, faster, and better[J]. ACM Computing Surveys, 2023, 55(12): 1-37.
[3] Albawi S, Mohammed T A, Al-Zawi S. Understanding of a convolutional neural network[C]//2017 international conference on engineering and technology (ICET). Ieee, 2017: 1-6.
[4] Kung H T, Leiserson C E. Systolic arrays (for VLSI)[C]//Sparse Matrix Proceedings 1978. Philadelphia, PA, USA: Society for industrial and applied mathematics, 1979, 1: 256-282.
[5] Vaswani A, Shazeer N, Parmar N, et al. Attention is all you need[J]. Advances in neural information processing systems, 2017, 30.
[6] San Juan P, Castelló A, Dolz M F, et al. High performance and portable convolution operators for multicore processors[C]//2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD). IEEE, 2020: 91-98.
[7] Jia Y, Shelhamer E, Donahue J, et al. Caffe: Convolutional architecture for fast feature embedding[C]//Proceedings of the 22nd ACM international conference on Multimedia. 2014: 675-678.
[8] Chen T, Li M, Li Y, et al. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems[J]. arXiv preprint arXiv:1512.01274, 2015.
[9] Deulkar A S, Kolhare N R. Fpga implementation of audio and video processing based on zedboard[C]//2020 International Conference on Smart Innovations in Design, Environment, Management, Planning and Computing (ICSIDEMPC). IEEE, 2020: 305-310.