VNU Journal of Science: Com. Science & Com. Eng., Vol. 39, No. 2 (2023) 66-78

VNU Journal of Science:
Computer Science and Communication Engineering

Journal homepage: http://www.jcsce.vnu.edu.vn/index.php/jcsce

Original Article

A Bandwidth-Efficient High-Performance RTL-Microarchitecture of 2D-Convolution for Deep Neural Networks

Nguyen Kiem Hung[1,*] , Tran Quoc Long[1]

[1]*University of Engineering and Technology, Vietnam National University, Hanoi, VietNam*
144 Xuan Thuy Street, Cau Giay District, Ha Noi, Vietnam

Received 27 September 2022
Revised 03 February 2023; Accepted 13 March 2023

Abstract: The computation complexity and huge memory access bandwidth of the convolutional layers in convolutional neural networks (CNNs) require specialized hardware architectures to accelerate CNN's computations while keeping hardware costs reasonable for area-constrained embedded applications. This paper presents an RTL (Register Transfer Logic) level microarchitecture of hardware- and bandwidth-efficient high-performance 2D convolution unit for CNN in deep learning. The 2D convolution unit is made up of three main components including a dedicated Loader, a Circle Buffer, and a MAC (Multiplier-Accumulator) unit. The 2D convolution unit has a 2-stage pipeline structure that reduces latency, increases processing throughput, and reduces power consumption. The architecture proposed in the paper eliminates the reloading of both the weights as well as the input image data. The 2D convolution unit is configurable to support 2D convolution operations with different sizes of input image matrix and kernel filter. The architecture can reduce memory access time and power as well as execution time thanks to the efficient reuse of the preloaded input data while simplifying hardware implementation. The 2D convolution unit has been simulated and implemented on Xilinx's FPGA platform to evaluate its superiority. Experimental results show that our design is 1.54× and 13.6× faster in performance than the design in [1] and [2], respectively, at lower hardware cost without using any FPGA's dedicated hardware blocks. By reusing preloaded data, our design achieves a bandwidth reduction ratio between 66.4% and 90.5%.

Keywords: 2D Convolution, RTL microarchitecture, Circle Buffer, Deep Neural Network, MAC, Loader.

* Corresponding author.
E-mail address: kiemhung@vnu.edu.vn

https://doi.org/10.25073/2588-1086/vnucsce.596

1. Introduction

Recently, thanks to the availability of Big Data and the rapid development of high-performance computing systems, artificial intelligence (AI) has been noticed and invested heavily. Deep Learning is a technique widely used in various AI applications. In the past decade, deep learning has been successfully applied to solve many problems in academia and industry, such as image classification, object detection and recognition, audio recognition, disease diagnosis, and control of self-driving cars [3].

Convolutional Neural Network (CNN) is one of the commonly used deep learning algorithms that provide high classification accuracy for image and object detection and recognition [4]. Moreover, they can be easily applied to new applications. However, CNN requires a large amount of memory to store millions of parameters in each CNN model. Furthermore, CNNs are computationally intensive because they require billions of operations per image. The high computational complexity combined with the parallelism inherent in these models makes them a target for hardware acceleration. There are two main challenges involved in implementing hardware for CNNs. The first challenge is the computational cost since their architecture includes many layers of convolution, each of which includes multiple multiplications. The second challenge is huge memory access bandwidth, where the speed of loading data from memory is much lower than the speed of processing. These two challenges have increased the need to develop specialized hardware architectures to accelerate CNN's computation while keeping hardware costs reasonable for area-constrained embedded applications [5].

The 2D convolution is the most basic and important operation in the convolutional layer of the CNN network. According to statistics, this operation consumes more than 90% of the total computation time of CNN. Furthermore, the convolution operation requires a large amount of data to be loaded from memory or stored into memory. Therefore, 2D convolution has always been a focus in the optimization process when designing hardware to accelerate CNNs. It will be of great benefit if we can shorten the computation time in these convolutional layers to achieve the best CNN acceleration.

The 2D convolution of a matrix I of size N×N and a kernel matrix W of size K×K can be represented by equation (1):

$$O(x, y)=\sum_{m=0}^{K-1}\sum_{n=0}^{K-1} I(x+m, y+n) \times W(m,n) \quad (1)$$

where, $0 \le x \le N - K$ and $0 \le y \le N - K$.

Let set $M = N - K + 1$, which is the size of the output matrix.

The process of calculating the convolution is more explicitly described by the algorithm in Fig. 1.

According to equation (1), the total data that needs to be loaded from the input matrix to complete the convolution is (K×K)×(N–K+1)×(N–K+1). However, the input data actually needed for the convolution is only N×N (corresponding to the size of the input matrix). If data repeatability can be exploited efficiently, memory access bandwidth can be reduced while also increasing convolution speed.

```
1   // traverse each row of the output matrix
2   For (x = 0; x < M; x++) do
3   // traverse each column of the output matrix
4       For (y = 0; y < M; y++) do
5   // traverse the rows of the kernel filter
6           For (m = 0; m < K; m++) do
7   // traverse the columns of the kernel filter
8   For (n = 0; n < K; n++) do
9   O(x, y) += I(x+m, y+n)×W(m, n)
10              End for
11          End for
12      End for
13  End for
```

Fig. 1. 2D convolution algorithm.

Fig. 2 shows an illustrative example of the process of taking data from the input matrix to compute the convolution between an input

image matrix with 6×6 size (i.e. N = 6) and a kernel filter matrix with 3×3 size (i.e. K = 3). Notice that in this figure the numbers in squares represent the ordinal number of the elements in the matrix. Each time a 3×3-submatrix (i.e. yellow one) is extracted from the input matrix and convoluted with the kernel filter matrix according to equation (1) to produce an element of the output matrix. There are 16 such submatrices. As a result, after the convolution is complete we get an output matrix with 4×4 size as shown in Fig. 3. Notice that in this figure the numbers in squares represent the value of the element in the matrix. Notice that when processing submatrices in the same Band in order from left to right, between the two convolution times, the submatrices differ only by 3 elements. Therefore, the 6 elements shared between the 2 convolution times should be stored in the convolution unit so that they do not have to be reloaded from external memory.

Therefore, data reuse is an important mechanism to improve performance and reduce memory access bandwidth in hardware implementation for CNNs.

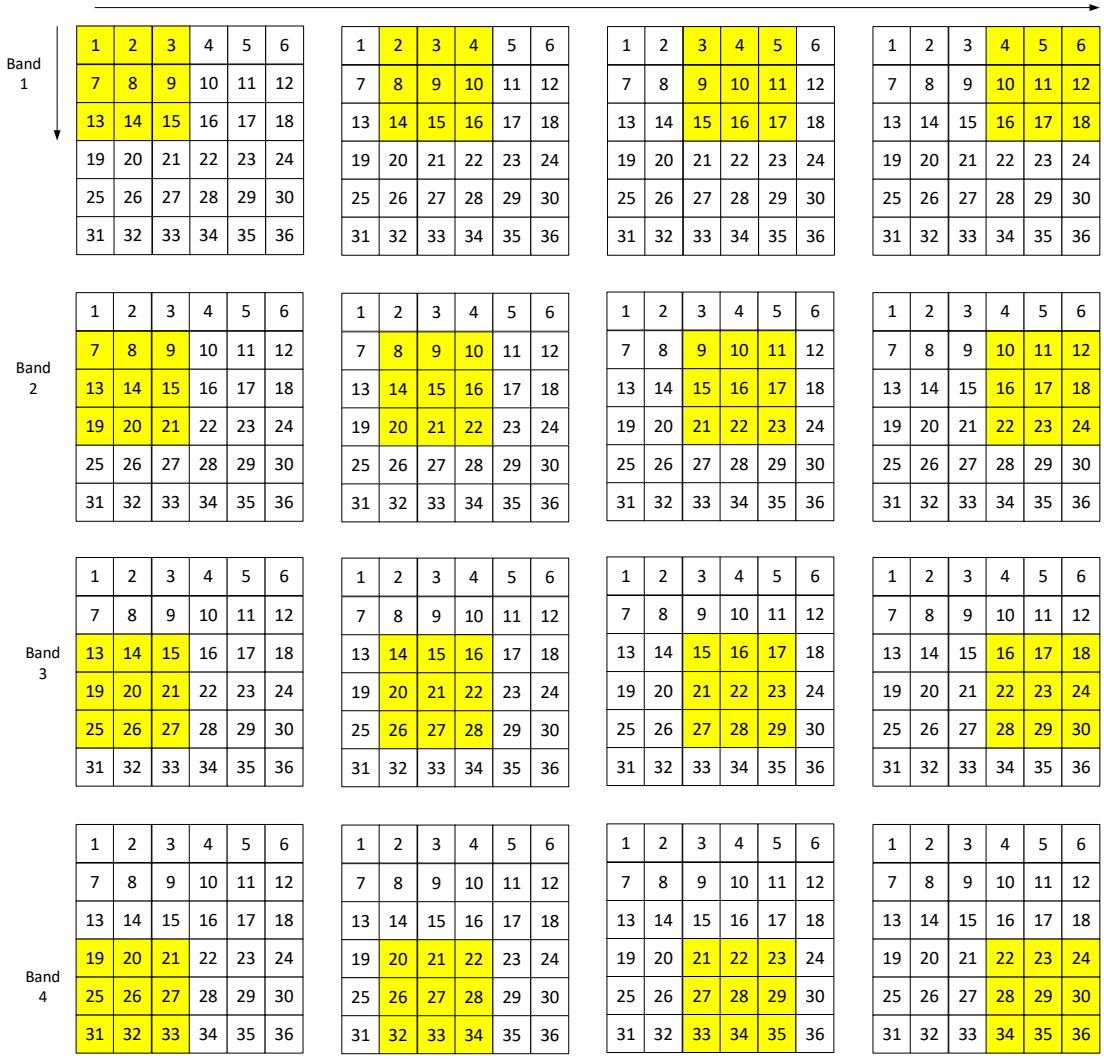

Fig. 2. An example illustrating the process of accessing data from the input matrix.

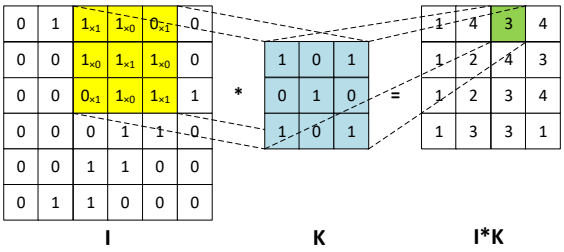

Fig. 3. An Example of a convolution operation.

There are several methods to perform the 2D convolution. The 2D convolution can be performed through matrix multiplication by converting the input matrix into a Toeplitz matrix as done by Yiran Chen et al in [5]. However, the 2D convolution using matrix multiplication introduces redundant data in the Toeplitz matrix. Therefore, the 2D convolution using matrix multiplication takes a long time if performed in serialization, or requires a huge memory access bandwidth if the operation is performed in parallel. The 2D convolution can be performed through the sliding window method where a window is slid over the input matrix to read out the submatrix that will be convoluted with the kernel filter matrix. The traditional sliding window method does not exploit any data reuse resulting in loading some input pixels and weights many times. Solutions like that of Yu-Hsin Chen et al in [6], recommend reusing input weights to avoid having to reload them from off-chip memory multiple times. In the [7], author Lin Bai et al presented an architecture to reduce the latency of the CNN network by reusing the input image data. The architecture uses a Line Buffer to align the input data with the filter weights therefore it only supports input matrices and filters with the fixed size that has been defined at the design time. In the [8], author Anaam Ansari et al. analyzed the redundant data in the 2D convolution and then proposed an architecture for 2D convolution to avoid reading the same data from off-chip memory to on-chip memory multiple times. In the [1], they have improved the processing engine to compute all outputs of the 2D convolution without having to reload any input data. However, this technique results in a hardware design that is relatively complex to implement.

This paper proposes an RTL (Register Transfer Logic) microarchitecture-level design of a 2D convolution unit for convolutional neural networks (CNN). The 2D convolution unit is designed to efficiently exploit the data shared between convolution operations to reduce memory access bandwidth thereby reducing power consumption while also taking into account the complexity and cost of hardware implementation. In addition, the design also must have minimal memory requirements compared to other designs. The microarchitecture proposed in the paper eliminates the reloading of both the weights as well as the input data. The architecture can compute the convolution of the input matrix and kernel filter that have any size with low latency and high throughput compared to other popular techniques. The architecture can reduce memory access time and power as well as execution time through efficient reuse of input data.

From an architectural perspective, the 2D convolution unit includes a dedicated Loader, a Circle Buffer, and a MAC (Multiplier-Accumulator) unit. The 2D convolution unit is designed as a 2-stage pipeline structure to reduce latency and power consumption but increase throughput. More specifically, this paper proposes a Loader's structure that can directly access memory and reads data in a configurable pattern. The Loader allows data to be loaded continuously and automatically from external memory into the convolution unit without the intervention of the central processing unit. In another aspect, the paper also proposes a dedicated Circle Buffer structure that can be configured according to different kernel filter sizes. Buffer allows the reused data between convolution operations to be saved with an efficient caching and management mechanism that is simple to implement in hardware and

requires minimal memory. Its asynchronous operation mechanism between data writing and reading makes it possible for the Loader and the MAC unit to work in parallel at different speeds without conflict.

2. Proposed Architecture

In this paper, the 2D convolution unit is designed to reduce the memory access bandwidth by efficiently using data that repeats between convolution operations. The top-level structure of the 2D convolution unit is shown in Fig. 4. To maintain the accuracy of the CNN's operation, the 2D convolution unit was designed with 24-bit fixed-point architecture. The structure consists of a Loader, a Circle Buffer, a MAC unit, and a file of control registers RF (Register File). The RF register file includes Control Registers to store parameters to control the operation of the convolution unit, and Weight Registers to store the weights of the kernel filter. In this way, the filter weights only need to be loaded once at the time the convolution unit starts working. The Loader is designed to read an image from external memory in a predefined pattern and load it into a Circle Buffer. Each unit is designed and modeled in detail at RTL (Register Transfer Logic) level to get the best performance and lowest power consumption.

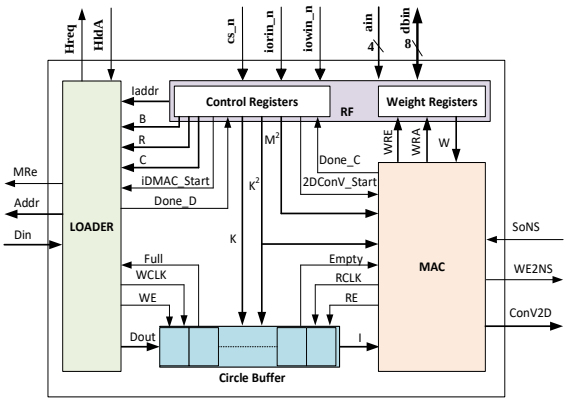

Fig. 4. The overall block diagram of the convolution unit.

Fig. 5 shows a timing diagram describing the operation of the convolution unit. First, the control parameters need to be written to the Control Registers. Right after that, the Loader will be activated to gradually load input data from the external memory to the Circle Buffer. At the same time, the weights of the kernel filter are also loaded into the Weight Registers. As soon as the Weight Registers have been loaded, the Circle Buffer is also ready, so the MAC unit can run.

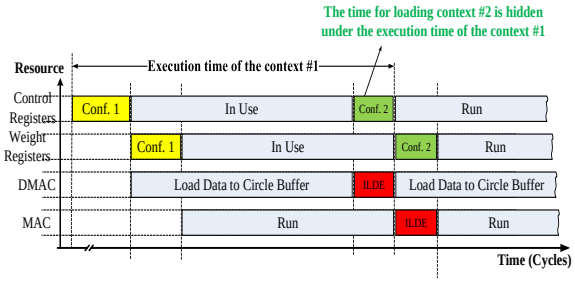

Fig. 5. Operating process of the convolution unit.

Each of the above components in the 2D convolution unit will be described in detail below.

2.1. Circle Buffer

Fig. 6 describes the operation of the Circle Buffer for the case of kernel filter size K = 3. The Circle Buffer is designed to store six values shared between two convolution operations. The yellow squares represent the data elements that need to be loaded from the input data memory to the buffer. The element numbered in red indicates that it is the first element in the submatrix that needs to be convoluted. It can be seen that for the first convolution operation of each band since there is no data in the Circle Buffer yet, the Loader needs to load nine elements from the input data memory to the Circle Buffer. In the next operation, since there are already six elements from the previous load, the Loader only needs to load three new elements.

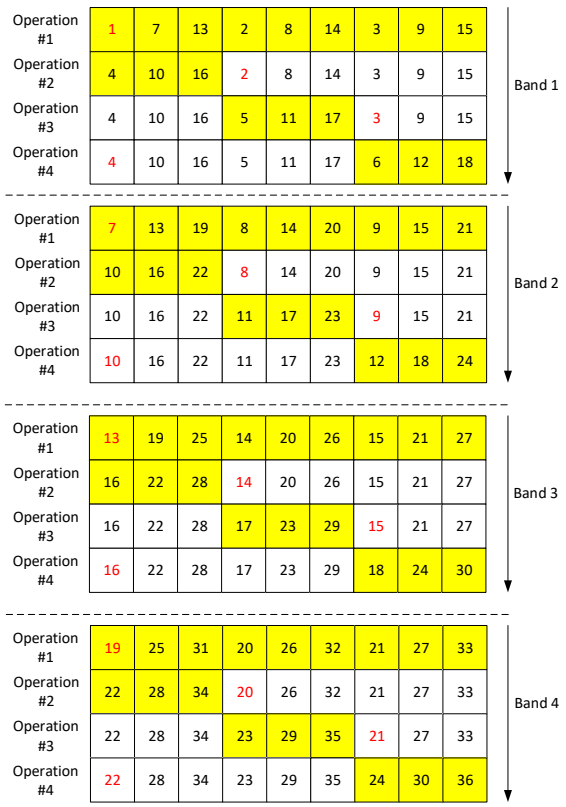

Fig. 6. Loading and buffering data in the circle buffer.

To manage writing data to the buffer and reading data out of the buffer, the Circle Buffer uses the following three pointers:

- The *Write Pointer (WP)* indicates the location in the buffer to which data can be written. After each data write this pointer will increase by 1;
- The *Read Pointer (RP)* indicates the start location in the buffer for a read round. After each read round is completed, this pointer is incremented by K units;
- The *Circle Pointer (CP)* indicates the location in the buffer that can be read in a read round.

Fig. 7 shows the state machine that describes the operation of the Circle Buffer. The Circle Buffer goes through some states that are described in detail as follows:

- *Reset state:* after power-on or reset, the buffer enters the Reset state. In this state, WP = RP = CP;
- *Full buffer state (Full):* occurs when WP + 1 = RP;
- *Empty buffer state (Empty):* occurs when RP = WP (after Reset) or CP = WP (when CP has not moved one round);
- *Write state:* in this state, data from the outside is written to the buffer's memory at the location pointed to by the WP pointer. After writing, WP increments by 1 to point to the next writable location;
- *Read state:* in this state, data is read out of the buffer's memory at the location pointed to by the CP pointer. The CP pointer moves to the next location after each read. The reading ends when the CP has moved one round, which is equivalent to R = K×K positions. At the end of the read, RP is updated to RP + K and CP is assigned with RP.

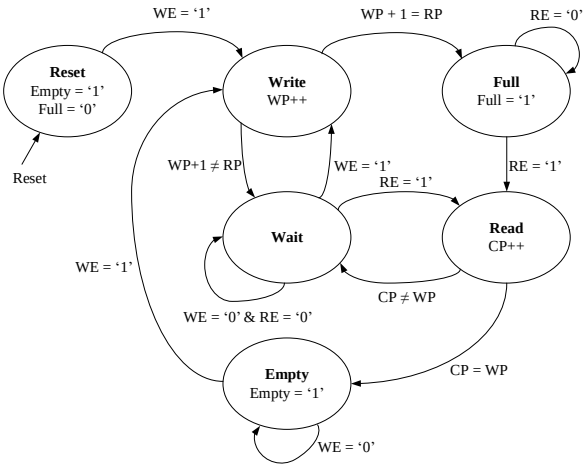

Fig. 7. State machine diagram of the circle buffer.

The architecture of the asynchronous Circle Buffer with two independent clock signals for data write and data read operations is shown in Fig. 8. The functions of the circle buffer's components are as follows:

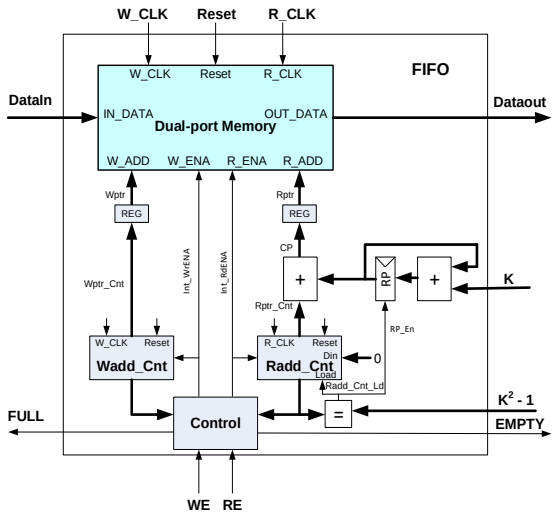

Fig. 8. Block diagram of the circle buffer.

- *Dual-port memory:* Memory is used to temporarily store data in the Circle Buffer. To support reading and writing in parallel, a dual-port memory block is used. The designed memory capacity is equal to C = $K_{max} \times K_{max} + 1$. Where, K_{max} is the largest kernel filter size that can be calculated using the 2D convolution unit.
- *Control Unit:* The control unit is responsible for generating control signals for the process of writing data from the outside to the circle buffer as well as the process of getting data from the Circle Buffer and writing it to the outside as shown in Fig. 9.
- *Address Counters:* There are two address counters (*Radd_Cnt* and *Wadd_Cnt*) that are used for generating addresses to read/write data to/from the buffer's internal memory. Address counters can be designed with either binary or Gray coding, however, the method used must be the same for the read and write address counters to ensure that read and write operations are performed in the same order.

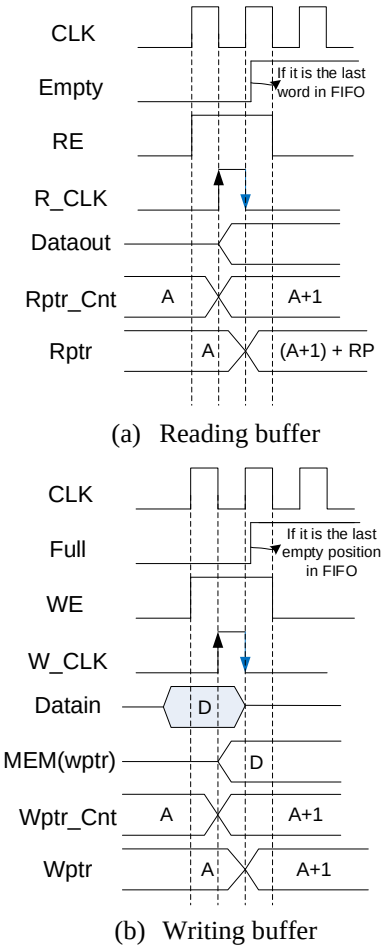

Fig. 9. Timing diagram for buffer read/write control.

2.2. Loader

The function of the Loader is to read the data of the input image from external memory and store it in the Circle Buffer. The input image of size N×N is divided into B bands, each with height R equal to size K of the kernel filter and width C equal to N (as illustrated in Fig. 2 with N = 6 and K= 3). These data Bands will be read into the convolution unit for processing in order from top to bottom. In each Band, the data is read in columns from left to right. The process of reading data from external memory and loading them into the Circle Buffer is illustrated in Fig. 6.

The operation of the Loader is described by the algorithm in Fig. 10.

```
1  //Input: Iaddr – Start address of input matrix
2  //Input: B – Number of Bands
3  //Input: C – Number of columns in a band
4  //Input: R – Number of rows in a band
5  //Output: Dout
6  //Variables: Paddr, Addr
7  While (Start = '1') do
8      Done = '0';
9      Paddr = Iaddr;
10     For i = 0 to B – 1 do
11        For j = 0 to C – 1 do
12           Addr = Paddr
13           For k = 0 to R – 1 do
14              Dout = M(Addr);
15        While (Full_FIFO = '1'); //wait until not full
16              FIFO = Dout;
17              Addr  =  Addr + C;
18           End for k;
19           Paddr ++;
20        End for j;
21     End for i;
22       Done = '1';
23 End while;
```

Fig. 10. Algorithm for Loader.

The diagram of the function blocks of the Loader is shown in Fig. 11. The functions of the Loader's main components are as follows:

- The *Control Unit (CU)* includes the FSM state machine, the counters (*i_cnt*, *k_cnt*, and *j_cnt*), and the comparators that perform the control part of the loops in the algorithm shown in Fig. 10. At the same time, the control unit also generates control signals for reading data from the input image memory and writing data to the Circle Buffer. The operation of the FSM state machine is depicted by the flowchart in Fig. 12. For direct access to external memory, the FSM implements a handshake protocol to receive bus ownership from the central processing unit via HldA and Hreq signals.
- The *Address Generating Unit (AGU)* consists of address registers (*Addr* and *Paddr*), multiplexers, and adders to calculate the address of the memory location that needs to be read from the input image memory.

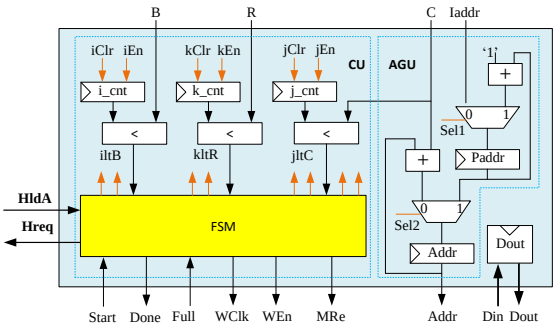

Fig. 11. Block diagram of Loader.

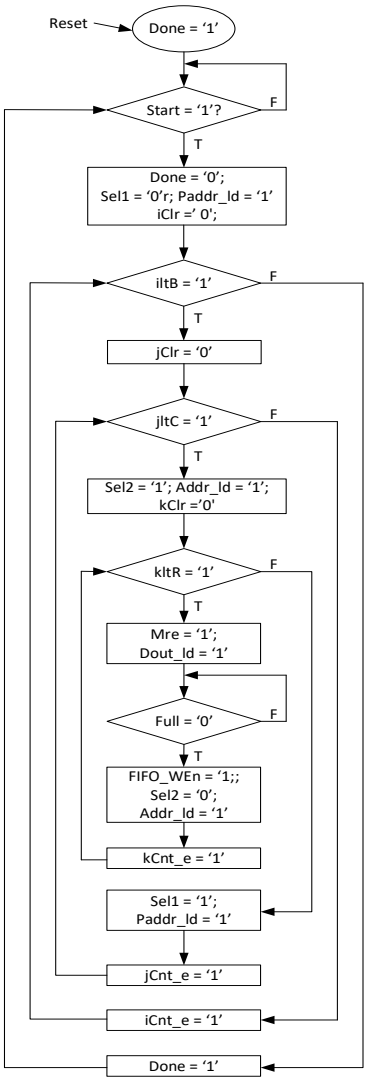

Fig. 12. FSM's flowchart of Loader.

2.3. MAC unit

The MAC unit reads data from the Weight Registers and Circle Buffers and then performs multiply-accumulate operations according to the control parameters. The operation of the MAC is depicted by a pseudo-code program in Fig. 13. Fig. 14 shows the functional block diagram of the MAC unit. The main functional blocks include:

- The *Datapath* is made up of a multiplier, an adder, and an accumulator. The datapath gets two input operands, X and Y, to calculate the product X×Y and add cumulatively to Conv2D. The operation of the datapath is controlled by the signals generated by the control unit.
- The *Control Unit* generates signals to read data I from the Circle Buffer and weight W from the Weight Registers File and controls the Datapath. After completing a convolution value, the Control Unit generates control signals to write this value to external memory. The operation of the Control Unit follows the state machine diagram shown in Fig. 15.

```
1   While (1) do
2     While (Start = '0');  //Wait until Start = '1'
3     Done = '0';
4     While (m < M×M – 1) do
5           WE2NS = '0';
6           ConV2D = 0;
7           For n = 0 to K×K – 1 do
8               I = Circle_Buffer;
9               W = W_RF(i);
10              ConV2D = MAC(I, W);
11          End for n;
12    While (SoNS= '1'); //wait until next stage ready
13          WE2NS = '1';
14          m++;
15    End while;
16    Done = '1';
17  End while;
```

Fig. 13. Pseudo-code describing the operation of MAC unit.

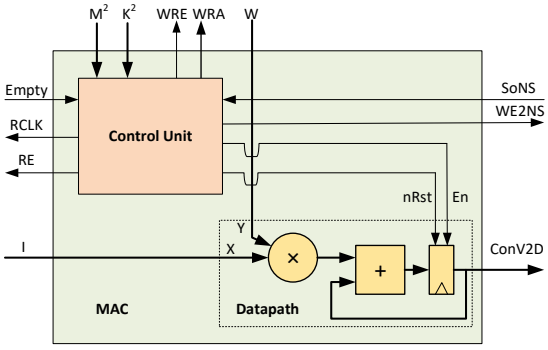

Fig. 14. Block diagram of MAC unit.

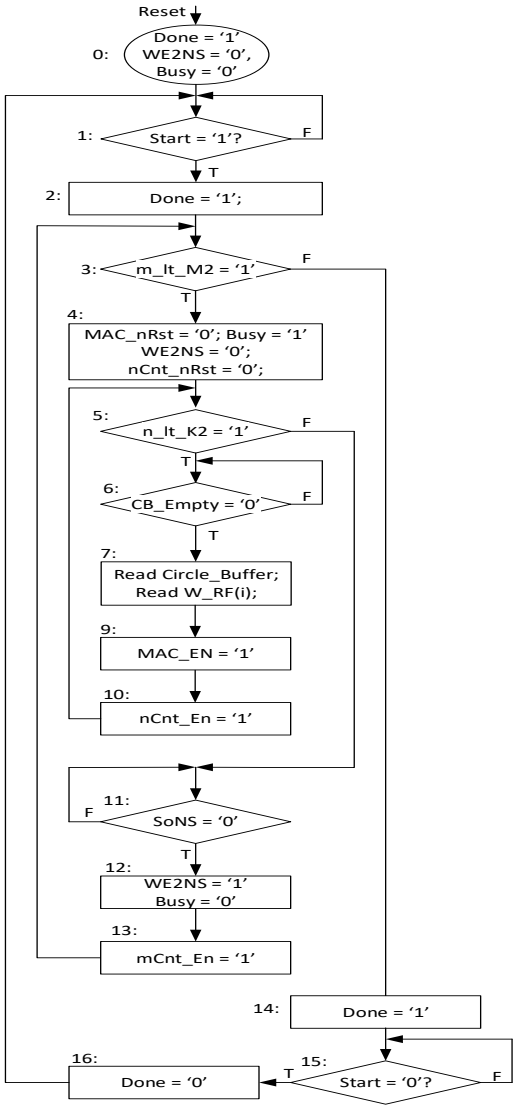

Fig. 15. FSM flowchart of MAC unit.

3. Experiment and Evaluation

The proposed 2DConV unit had been modeled at the RTL level in the VHDL language. Then, the 2DConV unit had been simulated, synthesized, and implemented on FPGA using Vivado Design Suite software from Xilinx. The experimental results and evaluation are presented in the subsections below.

3.1. Evaluation of Bandwidth Reduction Ratio

As described above, loading and buffering the data performed by the Loader and the Circle Buffer are the key to reusing the pre-loaded data and thus reducing memory access bandwidth and power consumption.

The number of times the data is loaded from the input image using the Circle Buffer is determined as follows:

$$(N\text{-}K\text{+}1)\times\{K^2\text{+}[(N\text{-}K\text{+}1)\text{-}1]\times K\}=(N\text{-}K\text{+}1)\times N\times K \quad (2)$$

Thus, the bandwidth reduction ratio achieved when using Circle Buffer is:

$$r=1-\frac{(N-K+1)\times N\times K}{(N-K+1)^2\times K^2}=1-\frac{N}{(N-K+1)\times K} \quad (3)$$

Fig. 16 shows the bandwidth reduction ratio according to different values of K, with N = 256. Where, BWo and BWn are memory access bandwidths with and without Circle Buffer, respectively. For example, for N = 255 and K = 11 then r = 90.5%.

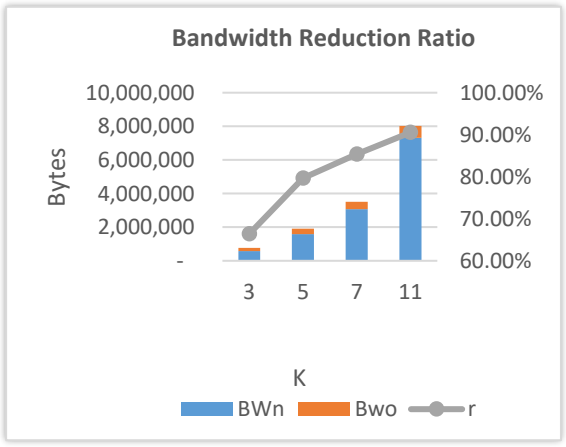

Fig. 16. Evaluation of bandwidth reduction ratio.

3.2. Evaluation of Hardware Cost

The 2DConV unit was also synthesized by Xilinx Vivado Design Suite. The post-implementation results of the 2DConV Unit on the ZynQ-7000 (xc7z020) chip are shown in Table I. The report timing summary also shows that the maximum operating frequency of the 2DConV Unit is about 203.6MHz.

Table I. Implementation results on Xilinx Zynq-7000

Resource	Utilization	Available	Utilization (%)
LUT	363	53200	0.68
LUTRAM	96	17400	0.55
FF	176	106400	0.17
IO	97	200	48.50
BUFG	2	32	6.25

Table II shows a comparison of the resource utilization and maximum frequency between our 2D convolution unit with several other architectures. Notice that the design in [2] and ours is implemented on the same Xilinx xc7z020 FPGA chip, while the design in [1] is implemented on the Xilinx xc7k325 chip.

Table II. Comparison between our ConV2D unit and others

Platform	[1]	[2]	This work
FPGA Device	xc7k325	xc7z020	xc7z020
Frequency	200MHz	173MHz	203.6MHz
Power	0.117W	N/A	0.116W
DSP	25	0	0
LUT	1901	1372	363
LUTRAM	0	N/A	96
FF	3073	2159	176
BRAM	8	0	0

From the above experimental results, the following conclusions can be drawn. The data pattern to be loaded from external memory is simple resulting in reduced complexity and cost of hardware implementation. Minimum memory requirement as the buffer memory is used efficiently for caching the reused data between computations. This helps to reduce the area and

energy consumption for embedded devices, meeting the criteria of compactness and long device usage time. The design does not use the DSP and BRAM blocks available on the FPGA chip, which means that our design is technology-independent. In other words, it can be implemented with any FPGA technology or any CMOS ASIC technology.

3.3. Evaluation of Performance

The 2-stage pipeline structure via the Circle Buffer allows the hardware to be efficiently used to perform data loading and computation in parallel to reduce latency and increase processing throughput.

Fig. 17 shows the waveform obtained from simulating the post-implementation ConV2D unit on the Vivado Design Suite software. The Loader is first configured by writing control information to the registers *BReg*, *CReg*, RReg, and *IAddrReg*. Marker 1 (at 0.38µs) indicates when the Loader has been configured and started working (*InputDMAC_Start = '1'*). The Loader generates the control signals to load data from external memory into the Circle Buffer. At the same time, the MAC unit is also configured through the interface including signals *Coeff_in*, *C_WE_in*, and *C_WA_in*. The configuration process takes place in the period from Marker 1 to Marker 2 (at 2.7µs). At the time of Marker 2, the MAC unit is activated by the signal *ConV2D_Start = '1'*. From here, the operation of the MAC unit and the Loader unit takes place in parallel. Each time the MAC unit completes computing an output convolution value it sends a pulse to the signal *ConV2D_Done*. At the time of Marker 3 (at 36.3502µs) the signal *InputDMAC_Done = '1'* indicates that the Loader's data loading process according to the first configuration has been completed. Immediately after this point, the new configuration is written to the control registers of the Loader. Next signal *InputDMAC_Start = '1'* to allow the Loader to load new data to the Circle Buffer. This operation occurs even if the MAC unit is still performing calculations according to the previous configuration.

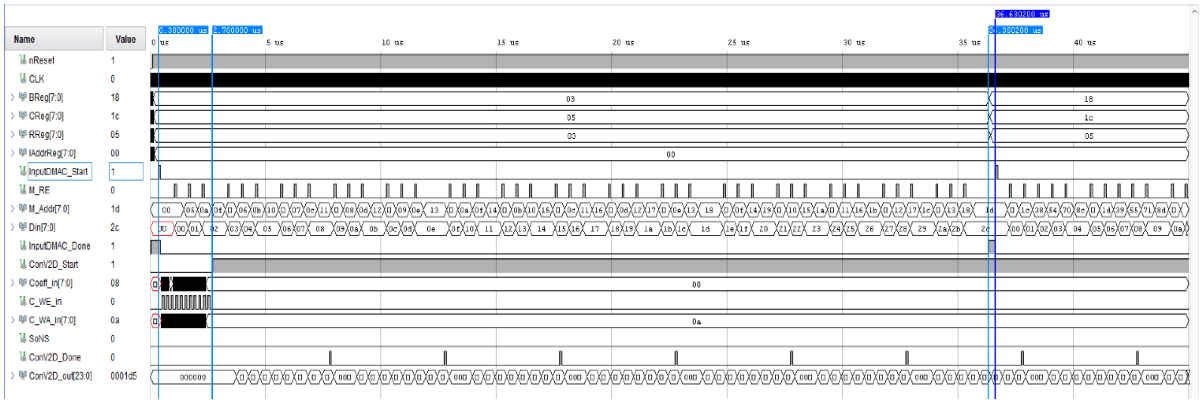

Fig. 17. Post-implementation simulation waveform of ConV2D Unit.

TABLE III. Comparison of the ConV2D unit with the work in [1]

	Input size	Kernel size	MACs	Clock Cycles	Frequency (MHz)	Execution time (s)	GOP/s	Power (W)	GOP/s/W	PEs	GOP/s/PE
[1]	1x28x28	20x5x5	288,000	11,520	200	5.76E-05	5	0.117	42.7	9	0.556
This work	1x28x28	20x5x5	288,000	76,203	227	3.36E-04	0.857	0.014	61.2	1	0.857

TABLE IV. Comparison of the ConV2D unit with the work in [2]

	Input size	Kernel size	MACs	Clock Cycles	Frequency (MHz)	Execution time (s)	GOP/s	Power (W)	GOP/s/W	PEs	GOP/s/PE
[2]	3x224x224	16x3x3x3	7,096,896	N/A	300	4.59E-03	1.55	N/A	N/A	36	0.043
This work	3x224x224	16x3x3x3	7,096,896	2,469,351	203	1.22E-02	0.58	0.016	36.5	1	0.583

TABLE III and TABLE IV show a comparison of our ConV2D unit with the designs in [1] and [2]

In TABLE III, we set up the experiment as follows to match the experimental setup in [1]. The FPGA chip used is xc7k325 of Xilinx. The input image map is a single channel image with a size of 28×28 (i.e. N = 28). There are 20 kernel filters, each is a 5×5 matrix (i.e. K = 5). Thus, the number of MAC operations to perform is 288,000. The design in [1] required 11,520 cycles to complete the work at a clock frequency of 200MHz. In other words, the execution time is 5.76e-5 seconds. Thus, the average performance of the design in [1] is 5 GOP/s (Giga operations per second). The power consumption of the design in [1] is 0.117 W, therefore, it has an energy efficiency of 42.7 GOP/s/W. The design in [1] uses 9 PEs (i.e. parallelism factor = 9), so the average performance per PE is 0.556 GOP/s/PE. Our design completes the same work in 76,203 cycles at a clock frequency of 227MHz. The corresponding execution time is 3.36e-4 seconds. Hence, the average performance of our design is 0.857 GOP/s. However, our design is using only 1 processing element (PE) (i.e. parallelism factor = 1). As a result, the power consumption of our design is 0.014W, therefore, it has an energy efficiency of 61.7 GOP/s/W. The average performance per PE is 0.857 GOP/s/PE. In summary, our design is 1.54× faster in performance and 1.43× more power efficient than the design in [1].

In TABLE IV, we set up the experiment as follows to match the experimental setup in [2]. The FPGA chip used is xc7z020 of Xilinx. The input image map is a three-channel image with a size of 224×224 (i.e. N = 224). There are 16 kernel filters, each is a 3×3×3 3D matrix (i.e. K = 3). Thus, the number of MAC operations to perform is 7,096,896. The design in [2] spent 4.59e-3 seconds to complete the work at a clock frequency of 300MHz. Thus, the average performance of the design in [2] is 1.55 GOP/s. Our design completes the same work in 2,469,351 cycles at a clock frequency of 203MHz. The corresponding execution time is 1.22e-2 seconds. Hence, the average performance of our design is 0.58 GOP/s. However, our design is using only 1 processing element (PE) (i.e. parallelism factor = 1), while the design in [2] uses 36 PEs. As a result, the average performance per PE of the design in [2] and ours is 0.043 GOP/s/PE and 0.583 GOP/s/PE, respectively. The power consumption of our design is 0.016W, therefore, it has an energy efficiency of 36.5 GOP/s/W. The performance of our design is 13.6× faster than the design in [2].

4. Conclusion

This paper presents an RTL microarchitecture level design of a 2D convolution unit for convolutional neural networks (CNN). Experimental results prove that the 2D convolution unit efficiently exploit the data shared between computations to reduce memory access throughput thereby reducing power consumption as well as complexity and cost of hardware implementation, while require minimal memory compared to other designs.

The microarchitecture proposed in the paper eliminates the reloading of both the weights as well as the input data. The architecture can compute the convolution of the input matrix and kernel filter that have any size with low latency and high throughput compared to others. In terms of performance, our design is 1.54× and 13.6× faster than the design in [1] and [2], respectively. Our design also achieves 1.43× more energy efficiency than the design in [2].

Acknowledgment

This work is supported by Vietnam National University, Hanoi under grant number TXTCN.22.02.

References

[1] A. Ansari, T. Ogunfunmi, Hardware Acceleration of a Generalized Fast 2-D Convolution Method for Deep Neural Networks, IEEE Access, 2022,Vo. 10, pp. 16843-16858, https://doi.org/10.1109/ACCESS.2022.3149505.

[2] X. Q. Nguyen, Q. C. Pham, An FPGA-based Convolution IP Core for Deep Neural Networks Acceleration, REV Journal on Electronics and Communications Vol 12, No. 1-2, 2022, http://dx.doi.org/10.21553/rev-jec.286.

[3] V. Sze, Y. H. Chen, T. J. Yang, J. S. Emer, Efficient Processing of Deep Neural Networks, Synthesis Lectures on Computer Architecture, Morgan & Claypool Publishers, Williston. Vol. 15, No. 2, 2020, pp. 1-341, https://doi.org/10.48550/arXiv.1703.09039.

[4] A. Khan, A. Sohail, U. Zahoora, A. S. Qureshi, A, Artificial Intelligence Review, Vol. 53, No. 8, 2020, pp. 5455-5516, https://doi.org/10.1007/s10462-020-09825-6.

[5] Y. Chen, Y. Xie, L. Song, F. Chen, T. A. Tang, Survey of Accelerator Architectures for Deep Neural Networks, Engineering, Vol. 6, No. 3, 2020, pp. 264-274, https://doi.org/10.1016/j.eng.2020.01.007.

[6] Y. H. Chen, T. J. Yang, J. Emer, V. Sze, Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices, IEEE Journal on Emerging and Selected Topics in Circuits and Systems, Vol. 9, No. 2, 2019, pp. 292-308, https://doi.org/10.48550/arXiv.1807.07928.

[7] L. Bai, Y. Lyu, X. A. Huang, Unified Hardware Architecture for Convolutions and Deconvolutions in CNN, In 2020 IEEE International Symposium on Circuits and Systems (ISCAS), 2020, (pp. 1-5). IEEE, https://doi.org/10.48550/arXiv.2006.00053.

[8] A. Ansari, T. Ogunfunmi, A Fast 2-D convolution Technique for Dep neural Networks, In 2020 IEEE International Symposium on Circuits and Systems (ISCAS), 2020, pp. 1-5.