PRTSM: Hardware Data Arrangement Mechanisms for Convolutional Layer Computation on the Systolic Array

Shuquan Wang, Lei Wang, Shiming Li, Tian Shuo, Shasha Guo, Ziyang Kang, Shuzheng Zhang, Weixia Xu

PRTSM: hardware data arrangement mechanisms for convolutional layer computation on the systolic array. *

Shuquan Wang[1], Lei Wang[1], Shiming Li[1], Tian Shuo[1], Shasha Guo[1], Ziyang Kang[1], Shuzheng Zhang[1], and Weixia Xu[1]

National University of Defense Technology wangshuquan3@163.com

Abstract. The systolic array is an array of processing units which share the inner data flow. Since the 2D systolic array fits the operation of multiplication and accumulation (MAC) naturally, there are many groups which use the systolic array to accelerate the computation of DNN (Deep Neural Network). However, the performance of the systolic array is limited by the data bandwidth. Some groups solve this problem with the method of loop tiling and care little about the pixel reuse potential of the convolutional layer. In this paper, we propose a novel method of PRTSM (Pixels Reuse with Time and Spatial Multiplexing) which reuses the pixels of the input feature map with time and spatial multiplexing. With it, we can significantly reduce the pressure of bandwidth and save the time of data preparing for convolutional layers on the systolic array. We propose three algorithms for this method and implement the corresponding hardware mechanisms on Xilinx FPGA XCVU440. Experiments show that our hardware mechanisms can reduce at least 72.03% of the off-chip traffic. The mechanisms proposed by this paper can reach a peak performance of 64.034 GOPS with a frequency of 167MHz.

Keywords: DNN · FPGA · Systolic Array · Hardware Data Arrangement.

1 Introduction

The systolic array is an array of processing units which share the inner data flow. With the development of DNN acceleration technique, many groups use the 2D systolic array to improve the efficiency of processing element (PE) [1] [2] [3]. However, the performance of the systolic array is limited by the data communication bandwidth. Here, we give an example of the systolic array with 8×8 PEs (pixel width: 16-bit, weight width: 16-bit). Each time, both pixels and weights should be fetched. If the work frequency is 100 MHz, the systolic array needs 200 Gbps ($8 \times 8 \times 16 \times 2 \times 100$) to make a full use of the PEs. However, the best performance of DDR4 is 6.4 Gbps. Concerning the fact we typically use much more PEs and design a higher work frequency, the gap of speed will be getting

* Supported by organization x.

bigger. That increases the pressure of bandwidth. On the other hand, the operation of convolution is a data-intensive operation. There are many overlapped parts between different convolutional operations. We can not store all the data on the hardware accelerator because of the limited on-chip store space. Facing this, many groups prefer to use the tiling method to reduce the pressures of bandwidth and on-chip storage. However, this tiling method ignores the potential of pixels reuse in the convolutional layer. On the other hand, the systolic array fetches the pixels one by one. The pixels of the input feature map have to be reordered so that they can fulfill the timing requirement of the systolic array. This procedure is called data arrangement. We typically use software to reorder the pixels on the host. However, this leads to a lot of additional data traffic.

Here, we give an example of 6×6 input feature map with a convolutional kernel of 3×3 and set the pixel to be 16-bit. The stride of convolution is 1. The total data we need is 576 bit $(6 \times 6 \times 16)$. However, the systolic array processes pixels in sequence. We need to reorder these pixels based on the convolutional operation. The final data we transform in the data channel is 2304 bit$((6-3+1) \times (6-3+1) \times 3 \times 3 \times 16)$. Most of the data come from the overlapped parts of different convolution operations. The method of tiling has no idea to reduce this additional off-chip traffic. Facing this, we propose a new method PRTSM (Pixels Reuse with Time and Spatial Multiplexing) and reorder the pixels on-chip. With it, we can reduce the off-chip traffic significantly. In our scheme, the hardware accelerator fetches one input feature map every time. We reorder these pixels to build up the final input sequence.

So far, there are a few of works focusing on the data arrangement optimization for DNN hardware accelerator [6] [7], especially for the systolic array. Google's TPU [7] shows a scheme of WS but does not give many details about how the intermediate data is set up for the computation of the next layer. [6] proposed a method of data mapping for CNN accelerator but didn't show the procedure of data arrangement. Concerning this, we propose three algorithms to reorder the pixels so that these pixels can be used by the systolic array directly. The main contributions of this paper are as follows.

- We propose three reordering algorithms which reuse the pixels of the input feature map to found the input data flows for convolutional layers on the systolic array;
- We implement the corresponding hardware mechanisms with Xilinx FPGA XCVU440. These mechanisms can accomplish the task of pixels reordering.

Experiments show that the algorithms we propose can reuse at least 71% of overlapping pixels. The corresponding mechanisms can reduce at least 72.03% of off-chip traffic with a power of less than 5.623 W. The best throughput of our mechanisms is 64.034 GOPS with a working frequency of 167 MHz. Note, since this work is a follow-up of SNN (Spiking Neural Network) accelerator, we denote the pixel with 1-bit. Our mechanisms can be configured in other representation schemes (for example 16-bit). The baselines are a straightforward algorithm we proposed and [14].

2 Related Works

The research of using the systolic array to accelerate DNN is getting popular. Wenqi Bao et al. [3] proposed a reconfigurable macro-pipelined systolic accelerator architecture and implemented 32-PE accelerator on Xilinx ML605. Samajdar et al. [1] proposed a software simulator called SCALE-Sim to explore the micro-architectural features and parameters configuration optimization. Zhang J et al. [2] analyzed the impact of permanent faults on the systolic array based neural network accelerator. They proposed two strategies to improve the fault-tolerant rate of DNN accelerator with a negligible drop in classification accuracy. So far, most of works focused on the functional design and performance optimization [8] [9] [10]. They paid a little attention to the optimization of overlapped pixels reuse for DNN on the systolic array. Some groups try to arrange the computation order efficiently. J. Qiu et al. [11] proposed a scheme which focused on the loop operations. Y. Chen et al. [12] try to reduce off-chip traffic with the help of input feature map compression. Their scheme is limited by the number of zeros. The others try to mine the data reuse across the layers. M. Alwani et al. [13] proposed fused-layer CNN to reuse the intermediate output feature map tiles. Their scheme needs a double feature map buffer for input-output feature map exchanging. Arash Azizimazreah et al. [14] proposed a scheme of shortcut mining which reuses the input feature maps in the residual networks. They decoupled the logical-physical banks and made a difficulty on the control logic design. Many works are focusing on the area of hardware accelerator optimization. However, a few of them try to reuse the pixels in the inner part of the convolutional layer.

3 Background and Preliminaries

3.1 Tiling and Optimization

The systolic array fetches pixels in sequence and computes the dot product with pixels (activations) and weights. All the pixels must be ordered in the right way so that they can calculate with the corresponding weights. The input feature map has to be reordered. However, the systolic array doesn't have this function. We should reorder the input feature map with software. This leads to a blow-up of pixels, which aggravates the problem of bandwidth limitation. Facing this, many groups choose to split the feature map into many tiles. Each time, the systolic array processes one tile. However, this method destroys the potential of pixels reusing in the convolutional layer. Here, we define the pixels which belong to the same convolution kernel area as one kernel block. There are many overlapped parts between different kernel blocks (see Fig. 1). These pixels can be reused. We divide the situations of pixels reusing in two forms, i.e., horizontal and vertical. We can reuse columns of pixels in the horizontal direction and rows in the vertical direction. Here, we can not use the classic tiling method.

We propose a new scheme which has two steps. First, we change the manner of data fetching. Each time, the systolic array fetches one input feature map

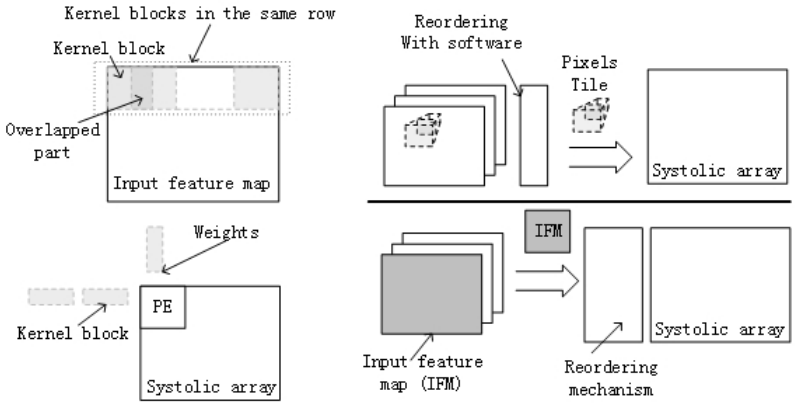

Fig. 1. The left figure shows the detail of pixels reordering and the right shows the principle of our design. Unlike the tiling method (shown in the right-up subfigure), we fetch one input feature map rather than one tile and reorder the pixels on-chip.

completely. For the first convolutional layer of DNN, one input feature map corresponds to one output feature map. The change of data fetching manner leads to a little waste of on-chip storage. As for the convolutional layers in the middle of DNN, many feature maps correspond to one output feature map. We should store the intermediate results on-chip which waste a lot of storage space. The input feature map should be split. Unlike the classic tiling method, we only split one input feature map into smaller feature maps. These smaller feature maps still have the potential of pixels reuse. Second, we should reorder the input feature map on-chip. We need a hardware mechanism to reorder these pixels to fulfill the timing requirement of MAC computation. We decide to reorder the pixels in a kernel block form. The motivation of this paper is reordering pixels as soon as possible and reuses pixels if possible.

3.2 Unfold Data Arrangement

The baseline of our data arrangement scheme is a straightforward algorithm which is called unfold data arrangement (see Algorithm 1). For discussion convenience, we introduce equation (1) [15], where F denotes the size of feature map is $F \times F$, $F^{'}$ represents the size of output feature map is $F^{'} \times F^{'}$, Z represents the zero padding and S denotes the stride. Here, we use three numbers $(F/S/K)$ to denote the parameters of the feature map. We denote the number of pixels in one feature map with N_{pixel} which equals $F \times F$ and the number of kernel blocks with N_{kernel} which equals $F^{'} \times F^{'}$. i denotes the index of current pixel and j denotes the index of current kernel block. We use $KernelB$ to denote the kenel block.

$$F^{'} = \frac{F - K + Z}{S} + 1 \tag{1}$$

We assume the input feature map has been stored on-chip. The size of the reordering buffer equals the size of kernel block, i.e., $K \times K$. The reordering buffer

Algorithm 1 Unfold Data Arrangement

set $i = 0; j = 0$;
repeat
  repeat
    if $Pixel_i$ *belongs to* $KernelB_j$ **then**
      $Buffer\ Pixel_i$;
      $i = i + 1$;
    end if
  until $i > N_{pixel}$
  Pop $KernelB_j$ *into the systolic array*;
  $j = j + 1$;
until $j > N_{kernel}$

fetches the pixels one by one. If the pixel belongs to the current kernel block, it will be buffered. When the reordering buffer gets all the pixels of the current kernel block, it pops these pixels into the systolic array. The time complexity of this algorithm is $O(mn)$ where n equals $(F^{'})^2$ and m equals K^2. The space complexity is $O(m)$.

The hardware mechanism of the unfold data arrangement algorithm has three modules, i.e., RAM, reordering buffer, and fetching address generator. The fetching address generator generates the fetching addresses so that the reordering buffer can fetch the pixels in a given order. Since we store the input feature map in one RAM, the operation of pixels reordering is limited by the reading channel number of RAM. Here, we get two variants, i.e., the unfold data arrangement mechanism with one reading channel ($UnFoldR1$) and the mechanism with two reading channels ($UnFoldR2$). Besides, this algorithm has two weaknesses. First, it works in a serial manner, which means it processes the pixels one by one. Second, it reuses little of the pixels. Since there are overlapped parts between different kernel blocks, we should fetch these pixels again and again. To solve these problems, we propose three improved algorithms.

4 Data Arrangement Algorithm

4.1 Fold Data Arrangement

To reuse as many pixels as we can, we propose the fold data arrangement algorithm (see Algorithm 2). We fetch all the pixels of the input feature map once and reorder them with the help of the arrange information. Here, we divide the kernel block into many places. Each place corresponds to one pixel. The arrange information is a piece of information which denotes the places where the pixel is needed. We use one buffer to store all the pixels of the same place in different kernel blocks. The number of buffers equals $K \times K$. If the current pixel corresponds to the place of one kernel block, we buffer it in the corresponding buffer. Each column of buffer banks corresponds to one kernel block. We reorder several kernel block in parallel. When one kernel block gets all the pixels, we pop them

into the systolic array. The time complexity is $O(n)$ while the space complexity is $O(m \lg n)$. The hardware implementation of the fold data arrangement algo-

Algorithm 2 Fold Data Arrangement

```
1: set i = 0; j = 0;
2: repeat
3:    Fetch Pixel_i into the Buffers based on the arrange information;
4:    if KernelB_j gets all the pixels then
5:       All buffers pop the pixels in their first banks i.e. pop KernelB_j;
6:       The pixels in the rest banks move to their former banks one by one;
7:       j = j + 1;
8:    end if
9:    i = i + 1;
10: until j > N_kernel
```

rithm is shown (see Fig. 2). This mechanism has five parts, i.e., fetching address generator, RAM, switch logic, reordering buffer, and the arrange information related part. The switch logic is used to push the pixel into the right buffer. The decision is made with the help of the arrange information. Since the arrange information can be generated by software or hardware and stored on-chip or off-chip, we get three variants. The first variant is we use software to generate the arrange information and store them on-chip ($FoldS$). The second is we fetch the arrange information and pixels from the off-chip in pair ($FoldN$). The third is we use hardware module to generate the arrange information ($FoldH$). The reordering buffer has K^2 small buffers. Note, the bank number of each buffer is decreased. Each buffer has a bank index pointer to denote the current bank. When a new pixel is arranged into the current buffer, the buffer stores it in the current bank. When one kernel block gets all the pixels, all the buffers pop their first banks. The rest pixels are passed to their former banks one by one.

4.2 Half-Fold Data Arrangement Variant 1

To reorder the pixels as soon as possible, we propose the half-fold data arrangement algorithm. We divide the situations of pixels reusing in two forms, i.e., horizontal and vertical. It means we reuse columns of pixels in the horizontal direction and rows in the vertical direction. Here, we get two variants. The first variant is that we only reuse pixels in the horizontal direction (see Algorithm 3). We organize all the kernel blocks in the same row into one group. All the groups are reordered in parallel. Each time, we fetch one pixel. When we found the first kernel block, the procedure is similar to the fold data arrangement algorithm. However, when the kernel block gets all the pixels, we copy these pixels rather than pop them to the systolic array. When we found the next kernel block, we reuse some pixels which belong to the previous kernel block. The time complexity is $O(\lg n)$ while the space complexity is $O(m \lg n)$. Another variant is we

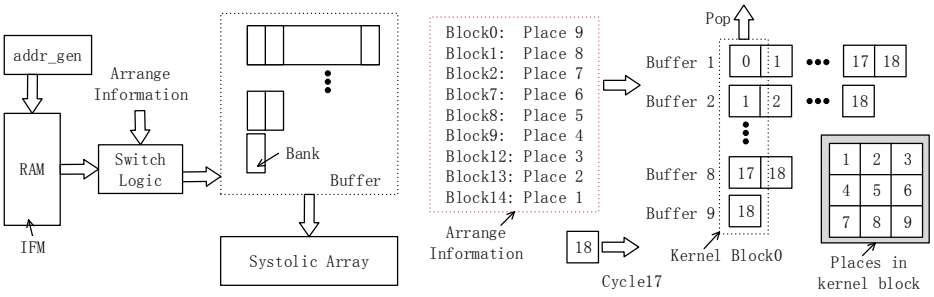

Fig. 2. The detail of fold data arrangement mechanism. The left figure shows the architecture and the right shows the state of reordering buffer (input feature: 6×6, kernel: 3×3, stride: 1).

reuse pixels in both the horizontal and vertical direction, which is discussed in the next section.

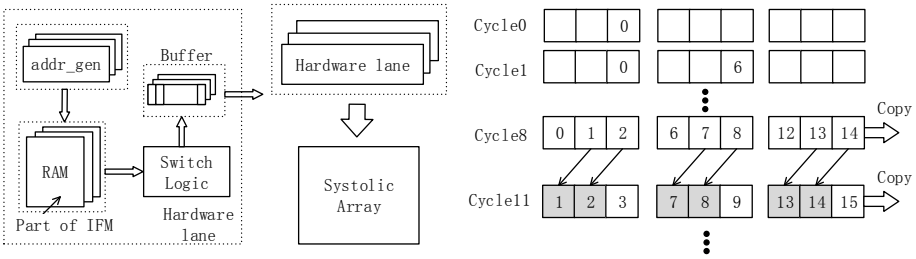

Fig. 3. The detail of half-fold data arrangement variant 1 mechanism (HalfFoldV1).

The hardware implementation of the fold data arrangement algorithm is shown (see Fig. 3). Here, we define a series of hardware modules as one hardware lane. One hardware lane processes one group. Each hardware lane has four parts, i.e., fetching address generators, RAMs, Buffers, and switch logic. We use K small buffer to found reordering buffer. Each small buffer has K banks. Each small buffer corresponds to one RAM and fetching address generator. We reuse the pixels with internal data moving in the small buffer. The internal data moving stride of the pixels in the small buffer is based on S.

4.3 Half-Fold Data Arrangement Variant 2

Variant 2 is shown (see Algorithm 4). Unlike the Variant 1, we organize a series of rows of kernel blocks into one group called one big group. Then, we process them in parallel. Here, we denote the number of big groups with G. One hardware lane processes one big group. There are some changes. First, we use pixels block to speed up the procedure of kernel block founding. In Variant 1, we fetch the pixels one by one. Here, we can fetch the pixels in block form. Second,

Algorithm 3 Half-Fold Data Arrangement Variant 1

set $i = 0; j = 0$;
Organize all the kernel blocks in the same row into one group.
#Process each group in parallel.
Reorder pixels to found the $KernelB_1$ *as same as* $Unfold$ *Data Arrangement Algorithm*;
if $j > 0$ **then**
  repeat
    repeat
      if $Pixel_i$ *belongs to* $KernelB_j$ *and it's a new pixel*; **then**
        $Buffer\ Pixel_i$;
        $i = i + 1$;
      end if
    until $i > N_{pixel}$
    Reuse some pixels of the former kernel block with internal data moving;
    if $KernelB_j$ *gets all the pixels*; **then**
      Copy $KernelB_j$ *into the systolic array*;
    end if
    $j = j + 1$;
  until $j > N_{kernel}$
end if

we reuse the pixels block between different kernel blocks rows. Since there are some overlapping parts in the vertical direction, we reuse these pixels with the help of the history buffer. When we process a new row of kernel blocks, we reuse some pixels in the previous row. The time complexity is $O(\lg n)$ while the space complexity is $O(m \lg n)$. The hardware lane is also different (see Fig. 4). First, we use a specific RAM to store the pixels block. Second, we carefully design the switch logic to make a switch between pixel input and pixels block input. Third, the small buffer has been designed to support the operation of updating all the banks one time. Besides, we use a history buffer to buffer the pixels block of the former rows.

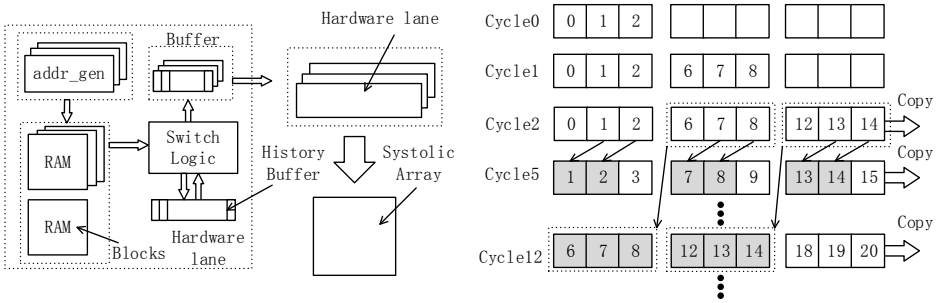

Fig. 4. The detail of half-fold data arrangement Variant 2 mechanism (HalfFoldV2).

Algorithm 4 Half-Fold Data Arrangement Variant 2

set $i = 0; j = 0$;
Organize several groups (defined in Variant 1) into one big group.
#Process each big group in parallel.
Found $KernelB_1$ *similar to the* $Variant$ 1. *Each time, fetch one pixels block.*
Found the rest kernel blocks of the first row like $Variant$ 1.
#The rest rows of kernel blocks.
if *start a new row of kernel blocks* **then**
  repeat
    Reuse some pixels blocks of the former kernel blocks row;
    Reuse some pixels of the former kernel block;
    Fetch new pixels to found the $KernelB_j$;
    Copy $KernelB_j$ *into the systolic array*;
    $j = j + 1$;
  until $j > N_{kernel}$
end if

5 Experimental Setup and Result

We implement all the data arrangement mechanisms at the RTL level with Verilog. We use Vivado 2016.4 for synthesizing and choose FPGA XCVU440 for implementation. For the concerning of performance exploration, we implement all the data arrangement mechanisms in a parameter configurable manner. Since this work is the follow-up of SNN accelerator, the pixel width is 1 bit. Our mechanisms can be configured in other representation schemes (for example 16-bit). The baselines are $UnFoldR1$, $UnFoldR2$ and [14].

5.1 Power Efficiency and Hardware Consumption

The detail of power consumption is shown in Table.2. As it shows, $HalfFoldV1$ wastes the most with a power consumption of 5.62 W. The detail of power efficiency is shown (see Fig. 5). For different feature maps, fold data arrangement mechanisms work the best while $HalfFoldV1$ works the worst. Though it works worse comparing to the others (except $HalfFoldV1$), $HalfFoldV2$ works better than $HalfFoldV1$.

Since 229/2/7 (one layer of ResNet152) is the biggest among all the convolutional layers, which leads to the biggest hardware consumption, we choose it to be the evaluation target. G is $F^{'}/2$, which leads to the biggest hardware consumption among all the settings of $HalfFoldV2$. As it shows in Table. 1, $HalfFoldV1$ wastes the most of hardware resource while $UnFoldR1$ wastes the least. Since we don't store the pixels and arrange information on-chip, $FoldN$ needs little RAM. We find $HalfFoldV1$ wastes the most of RAM while $HalfFoldV2$ gets a 14× reduction. We also notice that $HalfFoldV2$ needs the most of LUT, which is caused by the complexity of switch logic circuit.

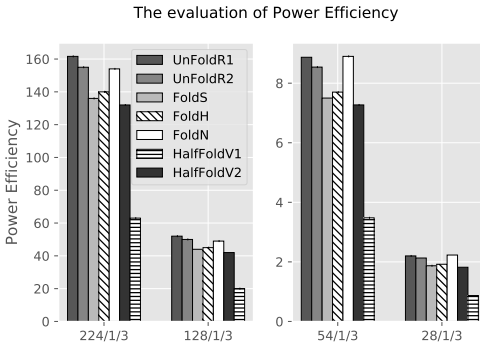

Fig. 5. The detail of power efficiency.

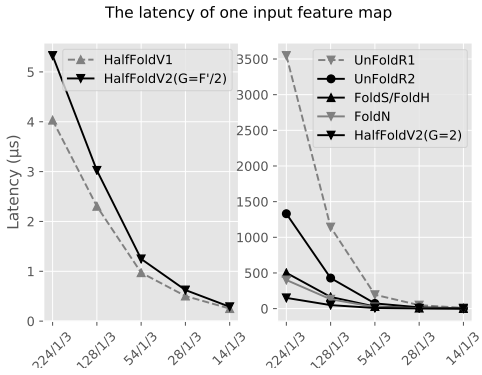

Fig. 6. The detail of reordering latency.

Table 1. Hardware resource consumption (229/2/7).

Mechanism	UnFoldR1	UnFoldR2	FoldS	FoldH	FoldN	HalfFoldV1	HalfFoldV2
LUT	**195**	4803	430	238	125	17421	**20543**
LUTRAM	**0**	4096	9953	0	0	0	**488**
FF	**145**	270	0	9721	9651	20235	**13479**
BRAM	**2**	0	90	2	0	392	**28**
DSP	**1**	2	0	0	0	0	**0**

5.2 Latency and Data Reuse Rate

The latency of one input feature map reordering is defined with $Cycles/Frequency$. The total cycles needed for one feature map and the working frequency is shown in Table. 1. As it shows, $HalfFoldV1$, $HalfFoldV2$ and $UnFoldR2$ get the

Table 2. Total cycles, Frequency and Power.

Mechanism	Cycles	Frequency(MHz)	Power
UnFoldR1/UnFoldR2	$K^2(F^{'})^2, K^2(F^{'})^2/2$	125/167	**2.68**/2.77
FoldS/H/N	F^2	**100**/**100**/125	2.94/2.7/2.71
HalfFoldV1	FK	**167**	**5.62**
HalfFoldV2	$((F^{'}/G-1)S+K)(F-K+1)$	**167**	3.03

best working frequency of 167 MHz while $FoldS$ and $FoldH$ get the worst. The detail of reordering latency is shown (see Fig. 6). $HalfFoldV1$ works the best. As for $HalfFoldV2$, it depends on the parameter G. When we set the G to be $F^{'}/2$, we get the best latency of $HalfFoldV2$. When we set the G to be 2, we get the worst latency of $HalfFoldV2$. All the latencys of $HalfFoldV2$ are better than the others (except $HalfFoldV1$).

The data reuse rate R is defined with equation (2), where D_{rest} denotes the overlapping pixels which the mechanism can not reuse, $D_{overlap}$ denotes the total overlapping pixels which equals $(F^{'})^2K^2 - F^2$. The detail of data reuse rate is shown (see Table.3). We make a comparison between $HalfFoldV1$ and $HalfFoldV2$ (see Fig. 7). When G equals $F^{'}$, the data reuse rate of $HalfFoldV2$ equals the one of $HalfFoldV1$. When G equals 1, $HalfFoldV2$ turns to be a form of fold data arrangement mechanism, and its data reuse rate becomes 100%.

$$R = (D_{overlap} - D_{rest})/D_{overlap} \times 100\% \tag{2}$$

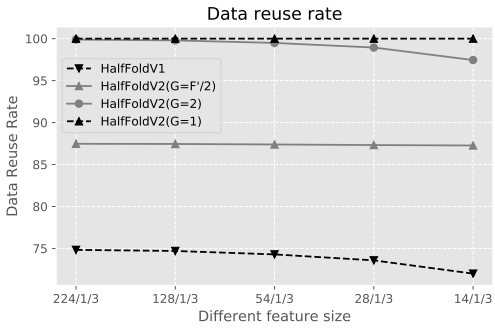

Fig. 7. Comparation of $HalfFoldV1/V2$.

Mechanism	Data reuse rate
UnFoldR1/2	0%
FoldS/H/N	100%
HalfFoldV1	$\frac{(F^{'})^2K^2-F^2-(F^{'}K-F)F}{(F^{'})^2K^2-F^2}$
HalfFoldV2	$\frac{(F^{'})^2K^2-F^2-(G-1)(K-S)F}{(F^{'})^2K^2-F^2}$

Table 3. Data reuse rate.

5.3 Comparison with State-of-the-Art

Here, we set the pixel width to be 16 bit (as same as [14] [13]). We use the layer of 229/2/7 for evaluation. We can reduce 100% overlapping pixels with $FoldS$, $FoldH$ and $FoldN$, 78.8% with $HalfFoldV1$ and 89.6% with $HalfFoldV2(G = 2)$. If we treat the total data to be the off-traffic and concern convolutional layers only, we can reduce 91.5% of the off-chip traffic with $FoldS$, $FoldH$ and $FoldN$, 72.03% with $HalfFoldV1$ and 73.39% with $HalfFoldV2$. All of them are better than [13] with a reduction of 26% and [14] with 43%. We can get at least 1.67× improvement. We compare the working frequency with [14]. $HalfFoldV1$ and $HalfFoldV2$ work with a frequency of 167 MHz, while [14] with 150 MHz. Here, one operation denotes placing one pixel to the corresponding place of kernel block. We can get a throughput of 51.14 GOPS ($HalfFoldV2$) and 64.03 GOPS ($HalfFoldV1$). Though they are lower than [14], our throughput is limited by the number of reordering operations in one convolutional layer. We also compare the on-chip RAM resource consumption with [14]. Our mechanisms waste at most 392 slices of BRAM while [14] needs 3210 BRAM. Our mechanisms waste a lower scale of RAM resource.

6 Conclusions

With the development of DNN research, the systolic array based accelerator is becoming more and more popular. The input data arrangement for convolutional layers on the systolic array turns to be a problem. Concerning this, we

propose three algorithms and implement all the hardware mechanisms on FPGA XCVU440. We finally find that the $HalfFoldV2$ mechanism gets a good balance between reordering speed and hardware resource consumption. It can work in a frequency of 167 MHz and reach the peak performance of 51.14 GOPS while reducing at least 73.39% of off-chip traffic. We can get a better throughput of 64.03 GOPS ($HalfFoldV1$). In the future, we will make further research on pixels reusing.

References

1. Samajdar A, Zhu Y, Whatmough P, et al. SCALE-Sim: Systolic CNN Accelerator[J]. 2018.
2. Zhang J, Gu T, Basu K, et al. Analyzing and Mitigating the Impact of Permanent Faults on a Systolic Array Based Neural Network Accelerator[J]. 2018.
3. Bao W, Jiang J, Fu Y, et al. A reconfigurable macro-pipelined systolic accelerator architecture[C]// 2011 International Conference on Field-Programmable Technology, FPT 2011, New Delhi, India, December 12-14, 2011. IEEE, 2011.
4. Y.-H. Chen, T. Krishna, J. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," in International Solid-State Circuits Conference, ser. ISSCC, 2016.
5. Sze V, Chen Y H, Yang T J, et al. Efficient Processing of Deep Neural Networks: A Tutorial and Survey[J]. Proceedings of the IEEE, 2017, 105(12):2295-2329.
6. Du Z, Fasthuber R, Chen T, et al. ShiDianNao: shifting vision processing closer to the sensor[C]// Acm/IEEE International Symposium on Computer Architecture. 2015.
7. In-Datacenter Performance Analysis of a Tensor Processing Unit[J]. 2017.
8. Razip M I M, Junid S A M A, Halim A K, et al. Sequence alignment using systolic array for an accelerator[C]// Power Engineering and Optimization Conference. IEEE, 2014.
9. Razip M I M, Al Junid S A M, Halim A K, et al. Sequence alignment using systolic array for an accelerator[M]. 2014.
10. Ito M. A power-efficient FPGA accelerator: Systolic array with cache-coherent interface for pair-HMM algorithm[C]// Low-power and High-speed Chips. 2016.
11. J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, Y. Wang and H. Yang. 2016. Going Deeper with Embedded FPGA Platform for Convolutional Neural Network. In Proceedings of the 24th ACM/SIGDA International Symposium on Field- Programmable Gate Arrays.
12. Y. Chen, J. Emer, and V. Sze. 2016. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In Proceedings of 43rd Annual International Symposium on Computer Architecture.
13. M. Alwani, H. Chen, M. Ferdman, and P. Milder. 2016. Fused-layer CNN accelerators. In Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture.
14. Arash Azizimazreah, Lizhong Chen. Shortcut Mining: Exploiting Cross-layer Shortcut Reuse in DCNN Accelerators. 2019 IEEE International Symposium on High-Performance Computer Architecture.
15. Y. Ma, M. Kim, Yu Cao, S. Vrudhula, J. Seo. 2017. End-to-End Scalable FPGA Accelerator for Deep Residual Networks. 2017. IEEE International Symposium on Circuits and Systems.