Instruction Driven Cross-Layer CNN Accelerator with Winograd Transformation on FPGA

Jincheng Yu, Yiming Hu, Xuefei Ning, Jiantao Qiu, Kaiyuan Guo, Yu Wang, Huazhong Yang
Electronic Engineering Department, Tsinghua University, Beijing, China

***Abstract*—In recent years, Convolutional Neural Network (CNN) has been widely applied in computer vision tasks. FPGAs have been widely explored to accelerate CNNs due to its high performance, high energy efficiency, and flexibility. By fusing multiple layers in CNN, the intermediate data transfer can be reduced. With a faster algorithm using Winograd transformation, the computation of convolution can be further accelerated. However, previous accelerators with cross-layer or Winograd algorithm are designed for a particular CNN model. The FPGA should be reprogrammed when running another CNN model on the hardware. In this work, we design an instruction driven CNN accelerator supporting Winograd algorithm and cross-layer scheduling. We firstly modify the cross-layer loop unrolling order to extract basic operations as instructions, and then improve the on-chip memory architecture for higher computation units utilization rate in Winograd.**

We evaluate the hardware architecture and scheduling policy on Xilinx Virtex-7 690t FPGA platform. As a case study, the intermediate data transfer can be reduced by over 90% on VGG-D CNN model with cross-layer policy. The performance of our hardware accelerator reaches 1500 GOP/s. Experimental results show that our design achieves a 7× speed-up than previous cross-layer FPGA accelerator on the same platform. The performance can be further improved by 78% if larger Winograd transformation sizes are used.

***Index Terms*—FPGA, CNN, Winograd, Instruction, Cross-Layer**

I. Introduction

Convolutional neural network (CNN) has achieved significant improvement in image recognition and detection and is thus widely applied in computer vision tasks. CNN has even proved its potentials in regions other than image processing, such as language processing [1].

However, CNN requires a large computation complexity and a huge storage capacity compared with conventional algorithms. A typical CNN, VGG-D [2] has 20M parameters. A single forward pass of VGG-D needs more than 30G operations. Thus, specific hardware architectures are designed to accelerate CNN on FPGA [3]–[5].

Two main factors affect the performance of an accelerator, one is the peak performance, and the other is utilization ratio. Transformation in mathematics can improve the peak performance with the same computation consumption, and the reduction of data transfer between on-chip and off-chip memory can improve the utilization. Furthermore, because the data transfer between on-chip and off-chip memory is also energy consuming, reduction in data transfer can decrease the energy consumption of the system. Previous work like [5] fuses layers to reduce data transfer, but it is designed for a particular CNN, lacking flexibility.

To leverage the computation and bandwidth resources on FPGA, and make the system more flexible, we propose an instruction driven CNN accelerator with the following contributions:

- We optimize cross-layer strategy for instruction support and propose a network dividing method to minimize intermediate data transfer.
- We improve the on-chip memory architecture for high-efficiency Winograd with cross-layer scheduling and design an accelerator system for CNN supporting instructions.

Experimental results show that our design is 7× faster than previous cross-layer FPGA accelerator [5].

II. Related Work

The accelerator of CNN is a widely researched technology in FPGA design. [6] designs a CNN accelerator with all the layers mapped on one chip, and this pipelined FPGA design results in high latency. In [7], Winograd algorithm is proposed and [8] implements a CNN accelerator with fast Winograd algorithm. However, these accelerators are designed for a particular CNN, so that the FPGA platform should be reconfigured when running another CNN, lacking flexibility. [3] proposes an instruction-driven accelerator, but the large data transfer between on-chip and off-chip memory limits the performance.

In general, there are four types of loop optimization techniques [9] in these hardware accelerators, exploiting different types of parallelism: kernel spatial, data spatial, input channel, and output channel. [3], [10] that utilize off-chip memory discuss the scheduling or data arrangement strategy to overcome the bandwidth limitation between the external and the on-chip memory. The work focuses on the intra-layer scheduling. Recently, [5] introduces a fuse-layer policy scheduling CNN inter-layer and reduces most of the intermediate data transfer.

III. Cross Layer With Instructions

[5] proposes a cross-layer scheduling policy and reduce most of the intermediate data transfer. However, this work is designed for the first five layers of VGG and cannot run other CNN. We improve the strategy of cross-layer scheduling and propose an instruction set with an instruction generator for various CNN.

978–1–5090–4825–0/17/$31.00 © 2017 IEEE

A. Cross-layer Strategy

The scheduling flow of the fuse layer in [5] is illustrated in Listing 1.

```
1:  for (row=0; row<$H$; row++) {
2:      for (col=0; col<$W$; col++) {
3:          for (layer=0; layer<$L$; layer++) {
4:              for (fi=0; fi<$N_i$; fi++) {
5:                  for (fo=0; fo<$N_o$; fo++) {
6:                      for (i=0; i<$k_x$; i++) {
7:                          for (j=0; j<$k_y$; j++) {
8:                              out[row][col][fo]+= \
9:                                  weight[fi][fo][i][j]*
10:                                 in[row+i][col+j][fi]
```

Listing 1. conventional convolution layer in [5]

For a net with L convolution layers, a conventional computation algorithm is shown in Listing 1. The N_i input featuremaps with size $W \times H$ are convolved with $k_x \times k_y$ weights in order to get the N_o output feature maps. [5] modifies the traditional loop unrolling order that put the loop of the layer inside the loop of row and column(line 4). Because the computation structure of each layer differs from others, it designs different computation units and data reuse units for different layers. The FPGA accelerator in [5] can only run a specific network and needs to be reprogrammed to run another network.

We modify the loop unrolling strategy in [5] to make the cross-layer scheduling policy more flexible. We propose the cross-layer scheduling strategy in line, which is shown in Listing 2.

```
1:  for (row=0; row<$H$; row+=2) {
2:      for (layer=0; layer<$L$; layer++) {
3:          for (fo=0; fo<$N_o$; fo++) {
4:              for (fi=0; fi<$N_i$; fi++) {
5:                  for (col=0; col<$W$; col+=2) {
6:                      out[row+:1][col+:1][fo]+= \
7:                          Winograd(in[row+:$k_x$][col+:$k_y$][fi]
8:                           ,weight[fi][fo])
```

Listing 2. convolution layer computation with Winograd

The row is regarded as the outermost loop variable (line 1). Each time the computation window moves down by two lines and performs the computation through the whole CNN until output. We package line 5 to 8 as an atomic operation and the parameters like W are set in the instruction.

B. Network divide method

To schedule CNN as Listing 2, the weights of the whole CNN should be stored on chip. However, the weights of CNN is usually much larger than the on chip buffer, so we propose a network divide method to arrange the CNN layers into several **Layer Blobs** and each layer blob can be scheduled as Listing 2. Layers can be merged into a Layer Blob (whose begin layer id is i and end layer id is j) when all the weights of the layers can be stored in the weight buffer on-chip as Equation (1). The volume of weights of the i^{th} layer is $weight_k$ and the size of the on-chip weight buffer is $Buffer_{weight}$.

$$\sum_{i \leq k < j} weight_k \leq Buffer_{weight} \tag{1}$$

There is no intermediate featuremap transfer inside a Layer Blob. The data volume to be transferred at the border of each Layer Blob is decided by the size of the output featuremap. The Blob transfers no data when the output featuremap ($Data_{out}$) can be stored on-chip, or double of the output featuremap when it cannot be stored on-chip. The data transfer of the Layer Blob ending with layer i is noted as BT_i:

$$BT_i = \begin{cases} 0 & Data_{out,i} \leq Buffer_{data} \\ 2 * Data_{out,i} & Data_{out,i} > Buffer_{data} \end{cases} \tag{2}$$

We can define the description function of total data transfer of a CNN with L layers ($DT_L(M)$) under a division method (M). It is the sum of data transfer in each Layer Blob. DT_n means the minimum data transfer of the first n layers.

$$DT_L(M) = \sum_{i \in blobs} BT_i \tag{3}$$

Our goal is to find a solution M^* under the hardware constraints governed by Equation (1), such that:

$$DT_L(M^*) = min(DT(M)) \tag{4}$$

We add a property $lastnode$ to each layer. $lastnode$ indicates whether each layer is the end layer of a Layer blob or not, and if this layer is the end layer of a Layer Blob, $lastnode$ also indicates the beginning layer of the same layer blob. The $lastnode$ of all layers form the dividing method M of a CNN network.

We use dynamic programming to find out the target M. For each layer in the network, the optimal division solution is one of the optimal division solutions for layer j plus the weight at layer j. Thus we can describe our process using pseudo codes in Listing 3.

```
$DT_0$ = 0;
$lastnode$[0] = 0;
for (i=1; i<L−1; i++) {
    $DT_i$ = $+\infty$;
    for (j=1; j<i−1; j++){
        if (Check(i,j) ){
            if ($DT_j$ + $BT_j$ < $DT_i$){
                $DT_i$ = $DT_j$ + $BT_i$;
                $lastnode$[i] = j;
$Layer\_Blob\_List$ = divide($lastnode$);
```

Listing 3. Transfer minimum division method

The *Check ()* function checks the satisfiability with Equation (1), and guarantees that the network division method shows the minimum data transfer under the constraint of on-chip memory.

IV. Hardware Design

In this section, we propose a structure to combine Winograd and cross-layer scheduling.

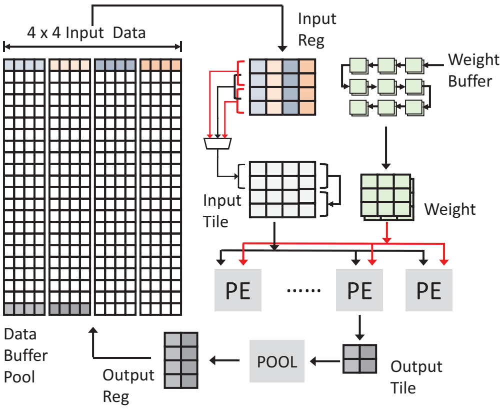

Fig. 1. Architecture overview

A. Architecture Overview

Figure 1 presents the architecture overview of our implementation of Winograd and cross-layer supporting on FPGA. To store input data and output data in the same buffer, the hardware parallelism of input channels and output channels should be the same. An input register file is designed to shuffle data in case of zero-padding. Weights are fetched from Weight Buffer in 3×3 size. PEs compute Winograd algorithm with the input tile and weights. We also design an output register buffer data in case of pooling. In our design, there are several Buffer Pools and Weights Buffer for featuremaps and weights in different channels. For easy illustration, we assume there is only one Buffer Pool and Weights Buffer, And the hardware parallelism is 1×1.

B. Data Buffer Design

Previous work adapts Line Buffer structure for data reuse [3]. Initially, the line buffer will read the first M lines from input data. PE will stay idle until the M lines of input data are fully read. Considering the scheduling of cross-layer strategy mentioned in section III, the basic operation is to calculate a line of the output featuremap. The PE will be idle waiting for the M lines of this featuremap for a single line. In cross-layer scheduling, the redundance of reading first M lines will keep PE idle in most of the time, resulting in low utilization of PE.

In order to support cross-layer scheduling, the input data buffer and the output data buffer should be merged into the same buffer, called **Buffer Pool** in our implementation. Each buffer pool consists of 4 RAMs, and Each ram provides four input data a clock cycle. The addresses of RAMs is given in instruction, and instruction also provides the write back addresses of RAMs to store the result of convolution layer.

C. Winograd PE Design

To compute a convolution for which the image tile is 4×4 and the kernel tile is 3×3, the Winograd algorithm uses 16 multiplications. However, the standard algorithm of the same size uses 36 multiplications. Winograd algorithm can be written in matrix form as:

$$Y = A^T \left[\left(G g G^T \right) \otimes \left(B^T d B \right) \right] A \tag{5}$$

Where the $\otimes$ indicates elementwise multiplication. The matrices are:

$$B^T = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} G = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix} A^T = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix} \tag{6}$$

g is the 4×4 input tile, d is the 3×3 convolution kernel. The multiplication with these transformation matrix elements can all be converted to shift operations(like$\times \frac{1}{2}$), which is very friendly to hardware.

The larger the Winograd tile is, the acceleration rate of Winograd is higher. However, transformation matrices of a tile greater than 4×4 contain elements that cannot be simply converted to shift operations(like $\times \frac{1}{5}$ in a 5×5 Winograd tile). For this reason, some approximation should be done ($\frac{1}{5} \approx \frac{3}{16} = \frac{1}{8} + \frac{1}{16}$). The approximation will result in precision degradation in CNN, especially in detection tasks. We set our Winograd tile size 4×4 to keep the precision of CNN, with a $2.25\times$ acceleration rate to the original standard convolution.

Figure 2 shows the structure of PEs. The calculation of convolution layer can be divided into four stages. The first stage is to do the transformation of input tile and weights. Note that the transformation of input data and weights can be divided into two constant matrix multiplication, the second multiplication of input tile is arranged in DSPs. The second stage is the element-wise multiplication of transformed input tile and weights. We use DSPs to perform this operation. The third stage is the transformation of result. The forth stage is to accumulate the output tiles from different input channels.

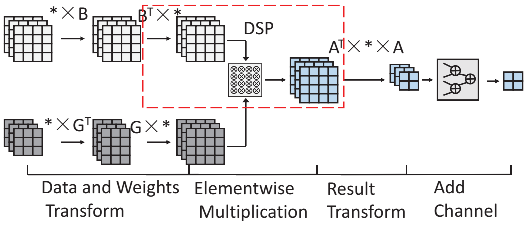

Fig. 2. PE design

We notice that DSPs can do more than element-wise multiplication. In our design, we set DSPs in the mode of $(A + B) \times D$, which can calculate the $B^T \times *$ and the $* \otimes *$ to make full use of DSPs.

The computation on hardware also parallelizes the loops of input channel and output channel. We define the unroll factors of input channel and output channel are P_i and P_o. Therefore, there is a total of $P_i \times P_o$ PEs in the hardware design. Moreover, in order to store input data and output data in the same buffer, P_i is the same with P_o. Each PE consists of 16 DSPs. Therefore, P_o and P_i are constrained by the number of DSPs on FPGA platform as eq. (7). The number of DSPs is described as DSP (3600 in our design). As there are 2

TABLE I
PERFORMANCE COMPARISON WITH STATE-OF-THE-ART FPGA ACCELERATORS

	[3]	[6]	[5]	[8]	Ours	
Platform	Zynq XC7Z045	Virtex 7 VX690T	Virtex 7 VX690T	MPSOC ZCU102	Virtex 7 VX690T	
Clock(MHz)	150	156	200[1]	200	200	
Data Format	16-bit fixed	16-bit fixed	32-bit float	16-bit fixed	8-bit fixed	
Used DSPs	780	2144	2987	2304	2048	
Batch	1	1	1	1	2	
Network	VGG-D	AlexNet	VGG-E layer 1-5	VGG-D	VGG-D	VGG-A
Complexity(GOP)	39	2.66	11.2	30.6	30.6	16.8
Intermediate Data Transfer (MB)	-	-	-	-	0.2(6.6)[2]	0.2(6.6)
Performance(GOP/s)	137	565.9	194.72	3044	1467.6(1366)[3]	1500(1317)

[1] The frequency is estimated to calculate performance

[2] The intermediate data transfer of cross-layer scheduling and without cross-layer (in brackets).

[3] The performance of cross-layer scheduling and without cross-layer (in brackets).

DDRs on the FPGA board, we set 2 computation kernels in our design.

$$P_i \times P_o \times 16 \times 2 \leq DSP \tag{7}$$

In the experiments, we set $P_i = P_o = 8$ in each computation kernel.

V. EXPERIMENTS

The performance comparison with other FPGA work is shown in Table I. Our work achieves state-of-the-art performance on FPGA. Moreover, our design is an instruction driven accelerator, providing flexibility for different CNN models. Two different networks, VGG-D and VGG-A, can run on the same FPGA with different instructions.

With the help of our cross-layer scheduling, the intermediate data transfer is reduced from 6.6MB to 0.2MB, with a 97% reduction. As is shown in the results, around 12% of acceleration is achieved by adapting cross-layer scheduling on a small model VGG-A, and 7% on a larger model VGG-D. It should be noticed that when the network goes large, the weights of Convolutions overwhelms the intermediate data. So the cross-layer speed-up ratio of a larger model (VGG-D) is lower than the speed-up ratio of a smaller model(VGG-A).

It should be noticed the Winograd kernel of our design is 4×4 and accelerates convolution $2.25\times$. The Winograd kernel of [8] is 6×6 and accelerates convolution $4\times$. As described in section IV, we choose 4×4 Winograd kernel to guarantee the precision of CNN models. We can also use a larger Winograd kernel size like 6×6 for tasks requiring less precision.If we use 6×6 Winograd kernel in our design, we can achieve 78% more performance with the same number of DSPs and achieve the same DSP efficiency with the state-of-the-art work [8].

VI. CONCLUSION

We propose an instruction driven CNN accelerator based on FPGA with the optimization of cross-layer scheduling and Winograd computing unit. We optimize cross-layer strategy for instruction support and propose a network dividing method to minimize intermediate data transfer. Then we propose a hardware architecture to support the Winograd computation and instruction set on a Xilinx Virtex-7 FPGA.

VII. ACKNOWLEDGEMENT

This work was supported by National Natural Science Foundation of China (61373026, 61622403, 61621091), National Key R&D Program of China (2017YFA0207600), Huawei Technologies Co. Ltd, and Joint fund of Equipment pre-Research and Ministry of Education (No. 6141A02022608).

REFERENCES

[1] S. Zhang, C. Liu, H. Jiang, S. Wei, L. Dai, and Y. Hu, "Feedforward Sequential Memory Networks: A New Structure to Learn Long-term Dependency," *arXiv:1512.08301 [cs]*, 3 2015.

[2] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 3 2014.

[3] 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, "Going Deeper with Embedded FPGA Platform for Convolutional Neural Network," FPGA '16, pp. 26–35, ACM, 3 2016.

[4] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," pp. 161–170, ACM, 3 2015.

[5] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer CNN accelerators," in *MICRO*, pp. 1–12, 2016.

[6] H. Li, X. Fan, L. Jiao, W. Cao, X. Zhou, and L. Wang, "A high performance FPGA-based accelerator for large-scale convolutional neural networks," pp. 1–9, IEEE, 3 2016.

[7] S. Winograd, *Arithmetic complexity of computations*, vol. 33. Siam, 4 1980.

[8] L. Lu, Y. Liang, Q. Xiao, and S. Yan, "Evaluating fast algorithms for convolutional neural networks on fpgas," in *FCCM*, pp. 101–108, IEEE, 2017.

[9] Y. Ma, Y. Cao, S. Vrudhula, and J.-s. Seo, "Optimizing Loop Operation and Dataflow in FPGA Acceleration of Deep Convolutional Neural Networks," FPGA '17, pp. 45–54, ACM, 4 2017.

[10] Y. H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2017.