Evaluating fast algorithms for convolution neural network

1 Introduction

Deep convolutional neural networks (CNNs) have achieved remarkable performance for various computer vision tasks including image classification, object detection, and semantic segmentation. The significant accuracy improvement of CNNs comes at the cost of huge computational complexity as it requires a comprehensive assessment of all the regions across the feature maps. Towards such overwhelming computation pressure, hardware accelerators such as GPUs, FPGAs, and ASICs have been employed to accelerate CNNs. In this project we are required to accelerate the convolution operation basing on the code given above on zcu104. In order to reduce the complexity of the project, all the hyper parameters are fixed value, that is, M=16, OR=56, OC=56, N=16, IR=56, IC=56, K=3, S=1, P=1.

1.1 Contribution

In this project, we consider two directions to accelerate the CNN inference. In the first direction, we design a new structure that can maximise the parallelism of CNN inference. A **line buffer** for input feature map is used to reuse two rows of the input feature map, moreover, by implementing this line buffer as a **Ping-Pong buffer** one can write data to input line buffer and executing CNN algorithm simultaneously. A Ping-Pong buffer is also used to implement output buffer. Still, this gives us the ability to write and read data at the same time. By such a hardware structure, the parallelism of this program is maximised.

The second direction is using some new algorithm with less computation complexity. The **winograd algorithm** is used to replace the general CNN algorithm. The basic idea behind the Winograd algorithm is using more addition, shifting operation to replace float multiplication, since addition and shift operation is cheaper than multiplication. Notably, the Winograd algorithm is faster than general CNN algorithm only at the fix-point scenario. Since this project is not suitable for fixpoint data, so we only implement it on the computer. However, as we will see, this algorithm can quickly assemble to our hardware framework (replace the normal CNN function with algorithm function). After using the method mentioned above our approach can achieve **117.77x** speedup of CNN inference (from **45344us** to **385us**) actually, this the speedup can be tremendously increased by utilising more PEs and using fixpoint data combine with Winograd algorithm on FPGA.

The input line buffer is acutually implemented by rotate buffer, but we divide the first dimension manually to let SDx

1. A line buffer is used for input feature map buffer with Ping-Pong buffer protocol to reused and parallel process data.
2. A Ping-Pong buffer is used for output buffer to maximise parallelism of program.
3. The data pack method is used to increase the bandwidth of the whole system.
4. Winograd algorithm is implemented on a computer to calculate convolution faster.
5. The final speedup of this algorithm is 117.77x.

This rest of this report is organised as follows, in the second part, a new hardware structure of CNN inference is proposed. In the third part, we will introduce the Winograd algorithm and show you how to assemble it to the previous framework. Part four shows you result in a different method. Finally, we conclude this report.

2 Hardware struture

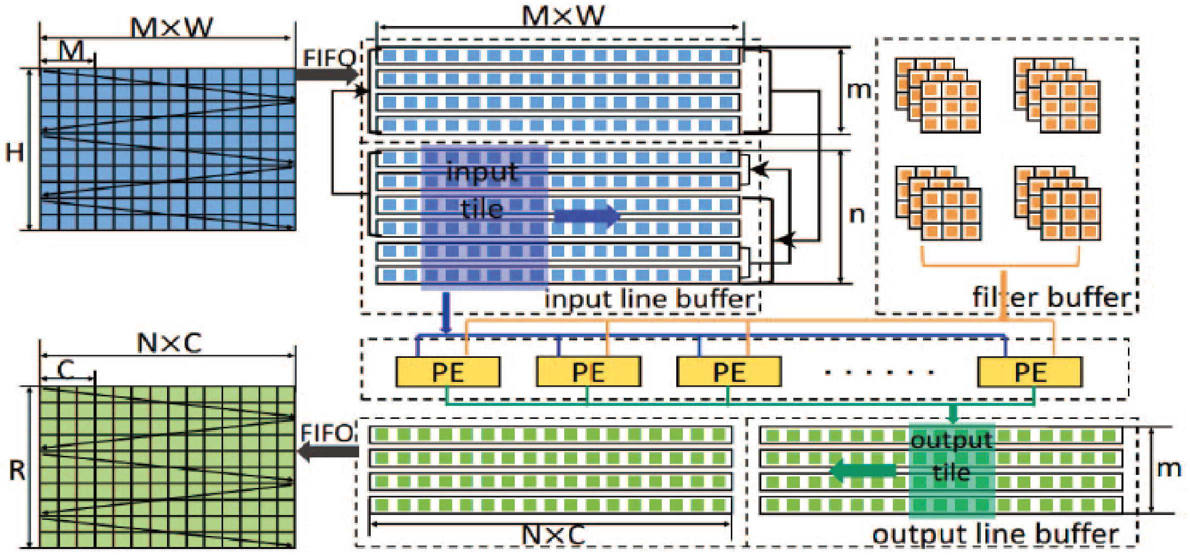


Figure 1: Hardware structure overview

We identify data reuse opportunities in the feature maps of neighboring tiles. To this end, we naturally implement line buffers. There are multiple channels of input feature maps (M) as shown in Figure 1. Each line of the line buffers stores the same rows across all the channels. Winograd/Elemente-Wise-Multiplication PEs fetch data from line buffers. Concretely, given an nn input tile, a Winograd PE will generate an mm output tile a Element-wise-multiplication PEs will give $(n - 2p) \times (n - 2p)$ output tile. We initiate an array of PEs by parallelizing the processing of the multiple channels. Finally, we use Ping-Pong buffers to overlap the data transfer and computation. All the input data (e.g. input feature maps, filters) are stored in the external memory initially. The input and output feature maps are transferred to FPGAs via a FIFO.

The first dimention of input line buffer is manually partioned to assist SDx recognize it as a rotate buffer, moreover, it will help SDx to implement it as a Ping-Pong buffer. Thus we can write one rows to input line buffer as the same time process three other input buffer rows. We process the input feature maps row by row, this is implemented by a for loop (line1-4, line 20-21). Two output feature map buffers are used to implement Ping-Pang buffer, for instance, we will first write data to buffer1 as the same time read data from buffer2 in the next turn we will first write data to buffer2 as the same tiem read data from buffer1 (line 21-22).

At the beginning of program first three rows and filter weight will be load to line buffer and filter buffer (line13-14), then we process input data row by row (line15). The write_row_ifm will write one row to line buffer at each iteration except the last iteration. The conv_write function will process the input line buffer (convolution operation) and write it to output buffer. The third function conv_read will write the output buffer to PS (PS is reading data from PL that is why it is called conv_read). By carefully write the code this three function can run simutaneously (line 21-22).

```
1 FIXDATA ifm_buff0 [N] [IC+2*P];
2 FIXDATA ifm_buff1 [N] [IC+2*P];
3 FIXDATA ifm_buff2 [N] [IC+2*P];
4 FIXDATA ifm_buff3 [N] [IC+2*P];
5
6 FIXDATA filter_buff [M] [N] [ROW_G] [ROW_G];
7
8 FIXDATA ofm_buff0 [M] [OC];
9 FIXDATA ofm_buff1 [M] [OC];
```

```c
int cifm_counter = 0;
int cofm_counter = 0;

load_cifm_data(cifm, ifm_buff1, ifm_buff2, ifm_buff3, &cifm_counter);
load_filter_buffer(tran_wgt, filter_buff);
for (int row = 0; row < IR; row += w_m)             //Row loop with stride w_m
{

        if (rotate_counter == 0)
        {
                write_row_ifm(cifm, ifm_buff0, &cifm_counter, 1);
                conv_write(filter_buff, ifm_buff1, ifm_buff2, ifm_buff3, ofm_buff0);
                conv_read(cofm, ofm_buff1, &cofm_counter, row!=0);

        } else if (rotate_counter == 1)
        {
                write_row_ifm(cifm, ifm_buff1, &cifm_counter, 1);
                conv_write(filter_buff, ifm_buff2, ifm_buff3, ifm_buff0, ofm_buff1);
                conv_read(cofm, ofm_buff0, &cofm_counter, 1);
        } else if (rotate_counter == 2)
        {
                write_row_ifm(cifm, ifm_buff2, &cifm_counter, 1);
                conv_write(filter_buff, ifm_buff3, ifm_buff0, ifm_buff1, ofm_buff0);
                conv_read(cofm, ofm_buff1, &cofm_counter, 1);
        } else if (rotate_counter == 3)
        {
                write_row_ifm(cifm, ifm_buff3, &cifm_counter, row!=IR-1);
                conv_write(filter_buff, ifm_buff0, ifm_buff1, ifm_buff2, ofm_buff1);
                conv_read(cofm, ofm_buff0, &cofm_counter, 1);
        }
        rotate_counter += 1;
        if (rotate_counter == 4)
        {
                rotate_counter = 0;
        }
}
conv_read(cofm, ofm_buff1, &cofm_counter, 1);
```

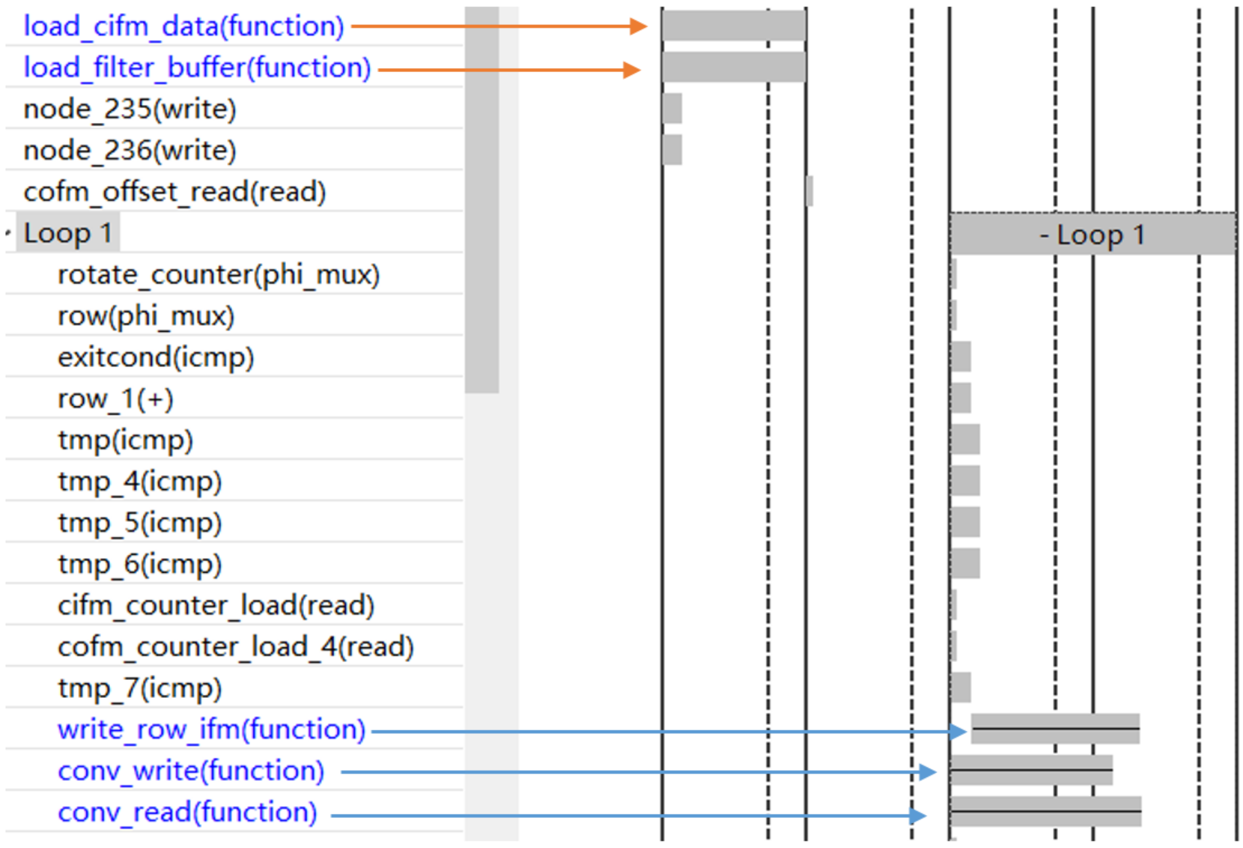


Figure 2: Data flow from HLS

Figure 2 shows the data flow of CNN accelerator, we can find the load input data and load filter data is processed simutaneously. And the three most import functions are excuted at the time. Thus by using such structure one can acheive best paralleism of CNN inference.

3 Winograd algorithm

The trends of CNNs are moving towards deeper topologies with small filters. The conventional convolution algorithm is general, but less efficient. As an alternative, convolution can be implemented more efficiently using Winograd minimal filtering algorithm. Let us denote the result of computing m outputs with the r-tap FIR filter as $F(m, r)$. Conventional algorithm for $F(2, 3)$ requires $2 \times 3 = 6$ multiplications. Winograd algorithm computes $F(2, 3)$ in the following way:

$$In = \begin{bmatrix} z_0 & z_1 & z_2 & z_3 \end{bmatrix}^{\mathrm{T}}$$

$$F = \begin{bmatrix} x_0 & x_1 & x_2 \end{bmatrix}^{\mathrm{T}}$$

$$Out = \begin{bmatrix} y_0 & y_1 \end{bmatrix}^{\mathrm{T}}$$

$$\begin{bmatrix} z_0 & z_1 & z_2 \\ z_1 & z_2 & z_3 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} m_1 + m_2 + m_3 \\ m_2 - m_3 + m_4 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \end{bmatrix}$$

Here m_1, m_2, m_3, m_4 are

$$m_1 = (z_0 - z_2)\, x_0 \quad m_2 = (z_1 + z_2)\, \frac{x_0 + x_1 + x_2}{2}$$
$$m_4 = (z_1 - z_3)\, x_2 \quad m_3 = (z_2 - z_1)\, \frac{x_0 - x_1 + x_2}{2}$$

Only 4 multiplications are necessary for computing m1 m4. The one-dimensional convolution using Winograd algorithm can be formulated using the transformation matrices A, B and G as follows,

$$Out = A^{\mathrm{T}} \left[(GF) \odot \left(B^{\mathrm{T}} In \right) \right]$$

In this project, we use two-dimensional Winograd algorithm $F(m \times m, r \times r)$ where the output tile size is $m \times m$, the filter size is $r \times r$ and the input tile size is $n \times n (n = m + r1)$. The output tile can be derived as follows,

$$\begin{aligned} \mathrm{Out} &= A^{\mathrm{T}}[U \odot V]A \\ U &= GFG^{\mathrm{T}} \quad V = B^{\mathrm{T}} \, \mathrm{In} \, B \end{aligned}$$

In this project we use $F(4 \times 4, 3 \times 3)$, and the A, B, G is:

$$B^T = \begin{bmatrix} 4 & 0 & -5 & 0 & 1 & 0 \\ 0 & -4 & -4 & 1 & 1 & 0 \\ 0 & 4 & -4 & -1 & 1 & 0 \\ 0 & -2 & -1 & 2 & 1 & 0 \\ 0 & 2 & -1 & -2 & 1 & 0 \\ 0 & 4 & 0 & -5 & 0 & 1 \end{bmatrix}$$

$$G = \begin{bmatrix} \frac{1}{4} & 0 & 0 \\ -\frac{1}{6} & -\frac{1}{6} & -\frac{1}{6} \\ -\frac{1}{6} & \frac{1}{6} & -\frac{1}{6} \\ \frac{1}{\frac{1}{24}} & -\frac{1}{12} & \frac{1}{6} \\ 0 & 0 & 1 \end{bmatrix}$$

$$A^T = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & 2 & -2 & 0 \\ 0 & 1 & 1 & 4 & 4 & 0 \\ 0 & 1 & -1 & 8 & -8 & 1 \end{bmatrix}$$

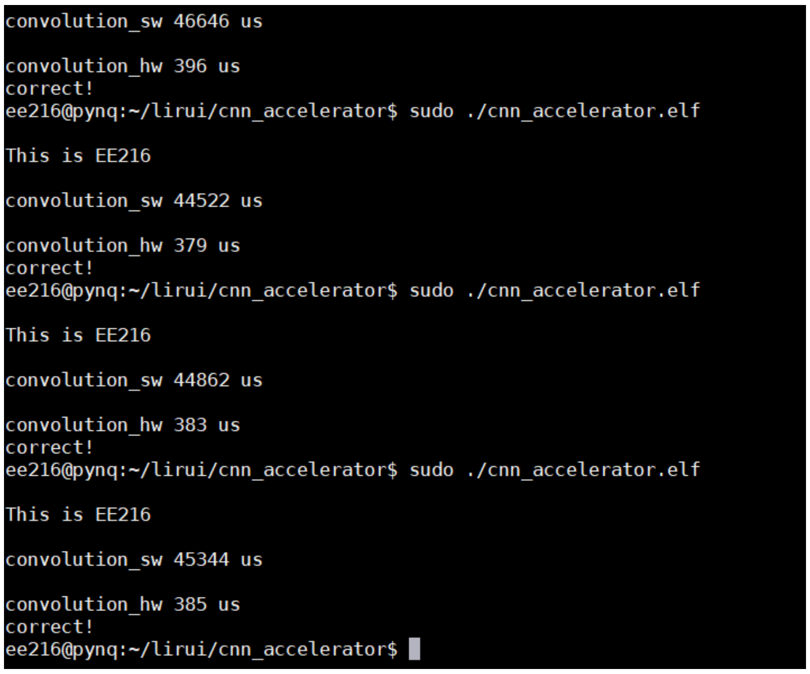
for a 4×4 output tile generated by convolving a 6×6 input tile with a 3×3 filter, conventional convolution needs $42 \times 32 = 144$ multiplications, while Winograd algorithm only needs $6 \times 6 = 36$ multiplications. However, Winograd algorithm requires more additions than conventional algorithm as it needs to add the intermediate results together. Notice, currently, only PS version is implemented since the forbidden of using fix point data in this project.

4 Result

We have implemented four methods. In method one we using 16 Element-wise-multipliction PEs to calculate convolution without Ping-Pong buffer. The method two still utilized 16 Element-wise-multipliction PEs but using Ping-Pong of input and output buffer to acheive best parallelism. In method two although we can write, process convolution, write back data at the same time but the time of wrting one row is longer than process convolution, in another word our system is limited by the bandwith of PS-PL communication. Thus in method 3 we using data pack method to improve the system bandwidth by using 16 PEs. In method four we using the same method as method three except there are 32 PEs, in this method we improve the system computation ability to match the high bandwidth of current system and this acheive the best performance. The following compares the performance and resource utilization of these four methods.

	DSPs	BRAM	LUTs	FFs	Estimated cycles	CPU time	FPGA time	Speed up
Method 1	360	69	50978	65262	1904137	44522us	4406us	**10.10x**
Method 2	360	69	57551	66627	1307137	44522us	2310us	**19.27x**
Method 3	360	125	60720	71015	1287615	44522us	648us	**68.71x**
Method 4	720	125	95339	119075	986559	44522us	379us	**117.47x**

Table 1: CNN accelerator speed up



```
convolution_sw 46646 us

convolution_hw 396 us
correct!
ee216@pynq:~/lirui/cnn_accelerator$ sudo ./cnn_accelerator.elf

This is EE216

convolution_sw 44522 us

convolution_hw 379 us
correct!
ee216@pynq:~/lirui/cnn_accelerator$ sudo ./cnn_accelerator.elf

This is EE216

convolution_sw 44862 us

convolution_hw 383 us
correct!
ee216@pynq:~/lirui/cnn_accelerator$ sudo ./cnn_accelerator.elf

This is EE216

convolution_sw 45344 us

convolution_hw 385 us
correct!
ee216@pynq:~/lirui/cnn_accelerator$ 
```

Figure 3: Result of best CNN accelerator

References

[1] L. Lu, Y. Liang, Q. Xiao and S. Yan, ”Evaluating Fast Algorithms for Convolutional Neural Networks on FPGAs,” 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), Napa, CA, 2017, pp. 101-108.

[2] Fast Algorithms for Convolutional Neural Networks

[3] Shen, Junzhong & Huang, You & Wang, Zelong & Qiao, Yuran & Wen, Mei & Zhang, Chunyuan. (2018). Towards a Uniform Template-based Architecture for Accelerating 2D and 3D CNNs on FPGA. 97-106. 10.1145/3174243.3174257.

[4] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA ’15). ACM, New York, NY, USA, 161-170. DOI: https://doi.org/10.1145/2684746.2689060

[5] Guo, Kaiyuan & Zeng, Shulin & Yu, Jincheng & Wang, Yu & Yang, Huazhong. (2017). A Survey of FPGA Based Neural Network Accelerator.