A High-efficiency FPGA-based Accelerator for Convolutional Neural Networks using Winograd Algorithm

Y Huang1,2, J Shen1,2, Z Wang1, M Wen1,2 and C Zhang1,2

1 Science and Technology on Parallel and Distributed Laboratory, National University of Defence Technology, Changsha 410073, China
2 College of Computer, National University of Defence Technology, Changsha 410073, China

Abstract. Convolutional neural networks (CNNs) are widely used in many computer vision applications. Previous FPGA implementations of CNNs are mainly based on the conventional convolutional algorithm. However, the high arithmetic complexity of conventional convolution algorithm for CNNs restricts the performance of accelerators and significantly increases the challenges of design. It has been proved that the Winograd algorithm for CNNs can effectively reduce the computational complexity. Although a few FPGA approaches based on the Winograd algorithm have been implemented, their works are lack of evaluation on the performance for different tile sizes of the Winograd algorithm. In this work, we focus on exploring the possibility of using the Winograd algorithm to accelerate CNNs on FPGA. First, we propose an accelerator architecture applying to both convolutional layers and fully connected layers. Second, we use high level synthesis tool to expediently implement our design. Finally, we evaluate our accelerator with different tile sizes in terms of resource utilization, performance and efficiency. On VUS440 platform, we achieve an average 943 GOPS for overall VGG16 under low resource utilization, which reaches higher efficiency than the state-of-the-art works on FPGAs.

1. Introduction
Convolutional neural networks (CNNs) have achieved great success in many computer vision applications [1], such as image recognition, speech recognition. Recently, the computational complexity of CNN has significantly increased with the continuous improvement of its recognition accuracy. Since CPUs can hardly provide the massive parallelism required by CNNs, many hardware accelerators, such as GPU, FPGA [2-12], and ASIC [13], have been developed to improve the performance of CNNs. Of these designs, FPGA-based accelerators have become a particularly option due to their high energy efficiency and reconfigurability. The existing FPGA implementations of CNNs are mainly based on conventional convolution algorithm [4, 7]. However, due to its high arithmetic complexity, more intensive design efforts are required to design an efficient accelerator for 2D CNNs on FPGA. As the current trend of CNN topologies is towards deeper, the increasing computational complexity of CNNs makes it difficult to design an efficient accelerator. Therefore, it is necessary to reduce arithmetic complexity of convolution operation by using fast convolution algorithm. In this work, we propose a FPGA accelerator design based on the Winograd algorithm. It is evident that using the Winograd algorithm can effectively reduce the arithmetic complexity [14], thereby achieving better acceleration performance on the FPGA platform compared to the conventional convolution algorithm.
A few prior works [2, 11] have focused on implementing an accelerator with the fixed tile sizes of the Winograd algorithm, which are lack of evaluation on the performance of different tile sizes. However, the Winograd algorithm contains different tile sizes and the arithmetic complexity reduction of convolution operation for each tile size is different. In this work, we aim to evaluate our accelerator with different tile sizes.

In this paper, we make the following contributions:

- An accelerator design using the Winograd algorithm on FPGA is proposed.
- An analytical model is developed to explore the design space.
- As a case study, we implement the accelerator for state-of-the-art CNNs: VGG16. Our accelerator is evaluated with different tile sizes. Moreover, the implementation on VUS440 achieves a peak performance of 1412 GOPS and an average performance of 943 GOPS.

The rest of the paper is organized as follows: Section 2 introduces the background of CNNs and the Winograd algorithm. Section 3 presents the details of accelerator design. Section 4 describes the design space exploration. Section 5 shows our experimental results. Section 6 discusses related works and Section 7 makes conclusion.

2. Background

2.1. Convolutional Neural Networks

CNNs extended from artificial neural network has been used to solve many complex problems [2]. CNNs are composed of series of layers, including convolution layer, ReLU layer, pooling layer and fully connected layer. Of these layers, convolution layer makes major contributions to CNNs, where the convolution operations occupy most of computation time [15]. Therefore, in this work our design focus on accelerating convolutional layers. In addition, our design can be reused in fully connected layers but in a suboptimal performance. A typically convolution layer contains input feature maps, filters and output feature maps. Each layer outputs \( M \) feature maps by the convolution of a set of \( N \) input feature maps of size \( H \times W \) and \( M \) sets of \( N \times K \times K \) filters. For each of \( M \) sets, \( N \) filters of size \( K \times K \) slide across the corresponding \( N \) input feature maps with a stride of \( S \). At each location, \( N \) filters multiply-accumulate with the overlapping elements of input feature maps to generate one output value. Each of output feature maps has the size of \( R \times C \), where \( R = \frac{H-K}{S} + 1 \), \( C = \frac{W-K}{S} + 1 \). The computation pattern of convolution layers can be expressed as follows:

\[
Out[m][r][c] = \sum_{n=0}^{N} \sum_{i=0}^{K} \sum_{j=0}^{K} In[n][S \times r + i][S \times c + j] \times W[m][n][i][j]
\] (1)

Where \( In \) and \( Out \) denotes the input and output feature maps of 3-dimensional arrays and \( W \) indicates the filters.

2.2. Winograd Convolution Algorithm

Winograd Convolution is a new class of fast algorithm for CNN which derived from Winograd’s minimal filtering algorithms [16]. The Winograd algorithm reduces the number of multiplications through a series of transformations on input tile and filters. To simply discussion, we describe computing \( m \) outputs with \( r \)-tap FIR filter as \( F(m, r) \). For \( F(2,3) \), We define the input vector \( \mathbf{x} = (x_0, x_1, x_2, x_3) \), filter \( \mathbf{w} = (w_0, w_1, w_2) \) and output vector \( \mathbf{y} = (y_0, y_1) \).

Firstly, by the Winograd algorithm, the input vector and filter are transformed into the following vectors:
\( x' = (x_0 - x_2, x_1 + x_2 - x_1, x_1, x_2) \), \( w' = \left( \frac{w_0 + w_1 + w_2}{2}, \frac{w_0 - w_1 + w_2}{2}, w_2 \right) \)

Secondly, by calculating the dot product of \( x' \) and \( w' \), we can get the following results:

\[
\begin{align*}
fi = (x_0 - x_2)w_0, & \quad f_2 = (x_1 + x_2)\frac{w_0 + w_1 + w_2}{2} \\
fi = (x_1 - x_1)w_2, & \quad f_3 = (x_2 - x_1)\frac{w_0 - w_1 + w_2}{2}
\end{align*}
\]

Finally, the Winograd algorithm for \( F(2,3) \) give the following equation:

\[
y = \begin{bmatrix} y_0 \\ y_1 \end{bmatrix} = \begin{bmatrix} x_0 & x_1 & x_2 \\ x_1 & x_2 & x_1 \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix} = \begin{bmatrix} f_1 + f_2 + f_1 \\ f_2 - f_3 - f_4 \end{bmatrix}
\]

(2)

It can be seen from Equation (2) that the Winograd algorithm only requires 4 multiplications to compute the output \( y \), while the standard convolution algorithm uses \( 2 \times 3 = 6 \) multiplications. Winograd algorithm can also be written in matrix form:

\[
y = A^T[(Gw)e \circ (B^T x)]
\]

(3)

Where \( e \) denotes element-wise multiplication. \( A, G \) and \( B \) are constant matrices, which are determined by the sizes of \( x \) and \( w \). For \( F(2,3) \), the matrices are

\[
B^T = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}, \quad G = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix}, \quad A^T = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix}
\]

(4)

Similarly, 2D Winograd algorithm for \( F(m \times m, r \times r) \) can be expressed as follows:

\[
y = A^T[(GwG')e \circ (B^T xB)]A
\]

(5)

Where now \( w \) is an \( r \times r \) filter and \( x \) is an \( n \times n \) \( (n = m + r - 1) \) input tile. In general, the Winograd algorithm for \( F(m \times m, r \times r) \) requires \( (m + r - 1) \times (m + r - 1) \) multiplications [17], and the standard algorithm uses \( m \times m \times r \times r \) multiplications. The 2D Winograd algorithm for \( F(m \times m, r \times r) \) can be used to compute convolution layers in CNNs, which can effectively reduce the arithmetic complexity. Table 1 shows the arithmetic complexity reduction for different tile sizes. It can be seen from table 1 that different tile sizes lead to different arithmetic complexity reduction.

**Table 1.** The arithmetic complexity reduction for different tile sizes.

| \( F(m \times m, r \times r) \) | Multiplications |
|-------------------------------|-----------------|
| \( m=2, r=3 \)                | 36              |
| \( m=4, r=3 \)                | 144             |
| \( m=6, r=3 \)                | 324             |
| Standard algorithm            |                 |
| Winograd algorithm            |                 |
| Arithmetic complexity reduction| 2.25x           |

3. Accelerator design

In this section, we bring forward a FPGA accelerator design based on 2D Winograd algorithm.
3.1. Accelerator Overview

Figure 1 shows an overview of our accelerator architecture. It can be divided into three principal modules, which are the receiving module, the computation module and the sending module. The receiving module is responsible for fetching the input feature maps from the off-chip memory. The computation module contains several processing elements and the accumulator. The Winograd PEs fetch the input data from the on-chip buffers and send the processed data to the accumulator buffer. Finally, the cached output results are written back to off-chip memory through the sending module.

3.2. PE

Our PE is designed based on Winograd convolution algorithm. As shown in figure 2, the Winograd PE is separated into three parts. PE1 is the transformation of input tile; PE2 is the transformation of filter; PE3 contains both dot-product units and transformation unit. More clearly, for computing $F(m \times m, r \times r)$, each time we need fetch $n \times n$ input tile from input feature maps, where current input tile is obtained by sliding the last input tile by m stride. In section 2, we mentioned the regular expression of Winograd convolution algorithm. As shown in equation (5), $A$, $G$ and $B$ are constant matrix already given. In fact, matrix multiplication in equation (5) can be designed as processing the rows and columns of matrix with multiplication and addition operation. Owing to different transformation for different tile size, we design several PEs for each of tile size.

As we can see from the figure 2, PE1 generate $x_i$ by calculating input tile column by column, and then processing $x_i$ row by row to acquire $U$. The transformation in PE2 and PE3 are the similar method. Two matrices $U$ and $V$ with the same scale $(n \times n)$ are fed to PE3. PE3 multiply $U$ and $V$ by dot-product operation and transform $M$ to $Out$. The overall structure of PEs is better for dataflow.

3.3. Parallel Design of PE

![Figure 2. Winograd PE](image-url)
The parallel design of PE is the key point to improve throughput of computation module. As we can see from the figure 2, PE1 and PE2 are two separate processes which can be executed in parallel. The code of computation module designed by HLS is shown in figure 3. Both Loop3 and Loop4 are unrolled, where parallelism degree of PE1 is $T_N$ and parallelism degree of PE2 and PE3 is $T_N \times T_M$. In fact, PE1 and PE2 executed in parallel consume lots of DSP and LUT, which causes Loop3 and Loop4 cannot be completely unrolled. Considering the short latency of the transformation of filters, we put PE2 out of the whole loop and let it to be executed first. The transformed filters are stored in the caches before executing PE3.

Figure 4 shows the completely parallel structure of computation module. $T_N$ input tiles are transformed by PE1 in parallel. Then $T_N$ transformed input tiles and $T_M$ sets of $T_N$ transformed filters fetched from on-chip buffers are fed to PE3 in parallel. PE3 generates $T_M$ sets of $T_N$ output tiles in a cycle. For each group, we accumulate $T_N$ output tiles and store the result in output buffers. It is notable that the current results are generated from $T_N$ input feature maps while we have $N$ input feature maps. Therefore, the computation module should be executed $\frac{N}{T_N}$ times. The complete output results are given by accumulating these $\frac{N}{T_N}$ sets of results.

```
Loop1: for (trr = 0; trr < TR; trr += m) {
    Loop2:    for (tcc = 0; tcc < TC; tcc += m) {
        #pragma HLS PIPELINE
        Loop3:        ...
        #pragma HLS UNROLL
        Load input_tile(trr, tcc, tn);
        Winograd_PE1(input_tile, input_tile1);
        #pragma HLS UNROLL
        Load filter(tm, tn)
        Winograd_PE2(filter, filter1);
        Winograd_PE3(input_tile1, filter1, tile_temp);
        Output_temp(trr, tcc, tm) += tile_temp; }}}}
```

**Figure 3.** Pseudo code of computation module

**Figure 4.** Computation engine
3.4. Memory Access
Communication is the main constraint in the optimization of accelerator design [2]. All the input feature maps and filters are stored in external memory. Owing to the large amount of input data of CNNs and limited on-chip resource, it is difficult for the accelerator to fetch all the data from the external memory at the same time. In this work, the input data are divided into several groups and the accelerator processing the data group by group. More clearly, for a set of $N$ input feature maps of size $H \times W$ convoluted with $M$ sets of $N \times K \times K$ filters, each time accessing external memory the accelerator fetch a set of $T_N$ input feature maps of size $T_N \times H$ and $T_M$ sets of $T_N \times K \times K$ filters. However, due to the limited memory bandwidth, there is a memory access delay when writing the input feature maps and filters to the on-chip buffers. As shown in figure 1, all the data processed by computation module comes from on-chip buffers. Therefore, the computation module could not be executed until the receiving module finish writing the input data to on-chip buffers. Obviously, the memory access delay has an impact on the optimization of the whole system. To overlap data communication and computation, double buffers should be applied in the receiving module. Double buffers used by ping-pong operation contains four sets: two buffers for input feature maps and two buffers for filters (here collectively referred to as input buffer 0 and input buffer 1). Computation module fetch the data from input buffer 0 while receiving module writes the off-chip data to input buffer 1, and the next time computation module fetch the data from input buffer 1 while receiving module writes the data to input buffer 0.

3.5. Fully Connected Layers
The designed PEs can also be used to execute fully connected layers. In the fully connected layers, each node of the layer is connected to all nodes of the upper layer, as shown in figure 5(a). In this work, we use the dot-product units to achieve the computation of fully connected layers. Figure 5(b) describes the implementation of fully connected layers. The batch size is determined by the size of dot-product array. Moreover, for $F(m \times m, r \times r)$, the batch size is $n \times n$. $T_N$ input tiles and $T_M$ weights are fed to $T_N$ dot-product arrays in a cycle. The input tile consists of $n \times n$ elements fetched from $n \times n$ different input features. For each dot-product array, $n \times n$ elements of input tile are multiplied by the same weight. Dot-product arrays generates $T_N$ sets of $n \times n$ outputs which are sent to accumulator. These $T_N$ sets of $n \times n$ outputs will be added point-to-point to acquire the final $n \times n$ outputs.

![FC layer](image)

(a) FC layer

![Process of FC layer mapping](image)

(b) The process of FC layer mapping

**Figure 5.** FC layer implementation

4. Design Space Exploration
In this section, we build an analytical model to theoretically estimate the performance of our accelerator. For $F(m \times m, r \times r)$, the design of accelerator adopts several parameters including input
output parameters \( \langle N, M, W, H, R, C, K, S \rangle \) and tile parameters \( \langle T_N, T_M, T_R, T_C \rangle \). Among these, \( N \) is the number of input feature maps, \( M \) is the number of output feature maps, \( W \) and \( H \) are the width and height of input feature maps, \( R \) and \( C \) are the width and height of output feature maps, \( K \times K \) (short for \( \text{Ksize} \)) is the size of filters and \( S \) is the stride of convolution. \( T_N, T_M \) are tiling factors for \( N \) and \( M \), and \( T_R, T_C \) are tiling factors for \( R \) and \( C \).

In the following, we model the execution time of a convolution layer. For convenient to analysis, we first model the individual execution time of receiving module, computation module and sending module. Execution time of these three modules are calculated by the following equations.

\[
T_{\text{recv}} = \frac{Q_{\text{recv}} \times DW}{BW} \tag{6}
\]

\[
Q_{\text{recv}} = Q_{\text{in}} + Q_{\text{s}} = T_M \times T_N \times \text{Ksize} \tag{7}
\]

\[
Q_{\text{in}} = T_N \times (S \times T_R + K - S) \times (S \times T_C + K - S) \tag{8}
\]

\[
T_{\text{send}} = \frac{Q_{\text{send}} \times DW}{BW} \tag{9}
\]

\[
Q_{\text{send}} = T_M \times T_R \times T_C \tag{10}
\]

\[
T_{\text{com}} = \left( \frac{T_R \times T_C \times II}{m \times m} + I_{\text{depth}} \right) \times \frac{1}{FREQ} \approx \frac{T_R \times T_C \times II}{m \times m \times FREQ} \tag{11}
\]

Where \( DW \) is the data width of input feature maps, weights and output feature maps which are 16-bit fixed point precision. \( BW \) is the bandwidth of the off-chip memory. \( II \) denotes the iteration interval of the pipeline of computation. \( I_{\text{depth}} \) is the depth of pipeline, which can be ignored due to its little impact on total execution cycle. \( FREQ \) is the clock frequency of FPGA. Considering the ping-pong operation (section 3) between two modules, we can model the total execution time as:

\[
T_{\text{total}} = R \times T_C \times \left[ \frac{M}{T_M} \right] \times \left[ \frac{N}{T_N} \right] \times \text{max} \left\{ T_{\text{recv}}, T_{\text{com}} \right\} + T_{\text{send}} \tag{12}
\]

\[
\approx \frac{R \times C}{T_R \times T_C} \times \left[ \frac{M}{T_M} \right] \times \left[ \frac{N}{T_N} \right] \times \text{max} \left\{ T_{\text{recv}}, T_{\text{com}} \right\}
\]

\( T_{\text{send}} \) can be ignored, because it has little impact on the total time. Due to limited bandwidth, both communication and computation are probably the constraint of optimization. To make the accelerator compute-bound [3], there should be a \( T_{\text{recv}} < T_{\text{com}} \). Thus, we can get a bandwidth expression with lower bound:

\[
BW_{\text{req}} \geq \frac{Q_{\text{recv}} \times DW \times m \times m \times FREQ}{T_R \times T_C \times II} \tag{13}
\]

If the actual bandwidth is out of this range, the accelerator would be memory-bound. The total operations of a convolution layer are:

\[
OP_{\text{total}} = 2 \times \text{Ksize} \times N \times R \times C \times M \tag{14}
\]

Assuming the memory bandwidth meets the minimum requirement of compute-bound, the calculation of the accelerator performance can be defined as:
\[ Per = \frac{OP_{\text{total}}}{T_{\text{total}}} = \frac{2 \times K_{\text{size}} \times N \times M \times m \times m \times Freq}{\left(\frac{N}{T_N} \times \frac{M}{T_M}\right) \times H} \]  

From the above expression of performance, we can modify the parameters \( \{T_N, T_M\} \) to maximize the performance. However, it is necessary to consider the on-chip resource when improving performance. Next, we will focus on modelling the utilization of DSP. Since we design the PEs based on Winograd algorithm, the dot-product operation of PE3 mostly consume DSP. Moreover, when the input tile \( n > 4 \), the transformation in PEs consumes DSP due to its high complexity. It is notable that a multiplication of 16-bit fixed point only consumes one DSP. Assuming the number of DSP required by the transformation is \( DSP_{\text{trans}} \), we can calculate the consumption of DSP by the following equation.

\[ DSP = T_N \times T_M \times \alpha(H) \times (n \times n + DSP_{\text{trans}}) \]  

As shown in the above equation, parameter \( \{T_N, T_M\} \) directly affects the consumption of DSP. Since the actual iteration interval of the pipeline of computation is more than one cycle, the parallelism degree of PEs cannot reach \( T_N \times T_M \). Therefore, we add a parameter \( \alpha \) to the equation, and \( \alpha \) is related to the iteration interval \( \alpha = \frac{1}{H} \).

5. Experimental Results

5.1. Experimental Setup
In this work, we implement our accelerator design by using Vivado HLS 2016.4, which can turn C program into hardware description language required by FPGA. Two FPGA platforms used to evaluate our design are Xilinx VC709 and VUS440. Xilinx VC709 integrates a Virtex-7 690t FPGA and two 4GB DDR3 DRAMS. The working frequency of this work is 150MHz. VUS440 consists of a Xilinx VCU440 FPGA and an 8GB DDR4, which evaluate our accelerator with 200MHz working frequency. The CNN model used for case studies is VGG16. All the convolution layers of VGG16 use \( 3 \times 3 \) filters, which is suitable for Winograd algorithm.

5.2. Result Analysis and Comparison
In our experiment, we evaluate the Winograd algorithm for both \( F(2 \times 2, 3 \times 3) \) and \( F(4 \times 4, 3 \times 3) \). Our designed accelerator contains several tile parameters \( \{T_N, T_M, T_R, T_C\} \) for CNNs. The uniform tile parameters and off-chip bandwidth for all layers of VGG16 are shown in table 2. The other tile parameters are assigned according to the size of each layer. It is worth noting that \( T_N \) in Conv1 should be changed to 3 due to its only 3 channels of input features.

| Parameter | \( T_N \) | \( T_M \) | \( BW_{\text{off}} \) |
|-----------|-----------|-----------|----------------|
| Design Value | 4         | 64        | 256bit         |
Moreover, the performance of all the same work at higher consumption of DSP is not only related to input tile size but also percentage. As we can see, the consumption of DSP is about 39% of DSP while $F(4 \times 4 \times 3)$ uses 21% of DSP. For further analysis, in the case of the same $T_x$ and $T_y$, the consumption of DSP is not only related to input tile size but also associated to the iteration interval of the pipeline of computation according to equation (16). In our implementation, the actual iteration interval for $F(2 \times 2 \times 3 \times 3)$ and $F(4 \times 4 \times 3 \times 3)$ are 3 and 11. Thus the theoretical results given by equation (16) are mostly same with the actual results. As we can see from the table 3, VUS440 contains less DSP but much more LUT (5.8X) than VC709. In general, VUS440 platform present the less overall resource utilization, which can be allowed to work at higher clock frequency.

Comparison with prior FPGA works are shown in table 4. We mainly compare DSP, overall performance and DSP efficiency. As we can see, DSP efficiency of our work is 0.86 on VC709 and 1.2 on VUS440, which significantly outperform other prior works. In this work, 16-bit fixed point data are used to evaluate the designed accelerator. The advantage of 16-bit fixed point is that addition operation does not require DSP and multiplication operation consumes only one DSP, while the Winograd algorithm reduce the number of multiplications and requires more adders to deal with the additional transformation. That’s why our implementation can reach such a high DSP efficiency. For the overall performance, our implementation achieves better results than other works except [2]. However, the work in [2] uses Intel FPGA SDK for OpenCL while we use Vivado HLS which can hardly reach a high frequency with over 90% utilization of DSP. Therefore, there is no comparability between our works and [2].

**Figure 6.** Results on multiple FPGA platform for different tile sizes

Figure 6 shows the results of all convolution layers on different FPGA platform for two sizes of accelerator. First, on both two platforms, Winograd algorithm for $F(4 \times 4 \times 3 \times 3)$ achieve higher peak performance (995 GOPs for VC709 and 1412 GOPs for VUS440) and higher overall performance (646 GOPs for VC709 and 943 GOPs for VUS440) than $F(2 \times 2 \times 3 \times 3)$. Moreover, the performance of $F(4 \times 4 \times 3 \times 3)$ is higher than $F(2 \times 2 \times 3 \times 3)$ in each layer of VGG16 except Conv5. As shown in table1, the Winograd algorithm for $F(4 \times 4 \times 3 \times 3)$ achieves a nearly 2x arithmetic complexity reduction over $F(2 \times 2 \times 3 \times 3)$. Therefore, it is reasonable that $F(4 \times 4 \times 3 \times 3)$ achieves better performance. Second, under the same size, the results on VUS440 are better than VC709’s. However, it can be seen from figure 6 that layer1-1 performs the worst result in convolutional layers for both two sizes. It is because layer1-1 has only 3 input channels, which even cannot perform ping-pong operation. Third, as the convolutional layer goes deeper, there is a downward trend on performance for both $F(4 \times 4 \times 3 \times 3)$ and $F(2 \times 2 \times 3 \times 3)$. The main reason is that the size of input feature maps are getting smaller from conv1 to conv5, which causes inefficient memory access.

Table 3 gives the resource utilization of designed accelerator with two sizes on different FPGA platform. Thanks to the Winograd algorithm, DSP are no longer the mainly consumption of on-chip resource. Instead, the utilization of LUT increases a lot. On the VC709 platform, $F(2 \times 2 \times 3 \times 3)$ consumes about 39% of DSP while $F(4 \times 4 \times 3 \times 3)$ uses 21% of DSP. For further analysis, in the case of the same $T_x$ and $T_y$, the consumption of DSP is not only related to input tile size but also associated to the iteration interval of the pipeline of computation according to equation (16). In our implementation, the actual iteration interval for $F(2 \times 2 \times 3 \times 3)$ and $F(4 \times 4 \times 3 \times 3)$ are 3 and 11. Thus the theoretical results given by equation (16) are mostly same with the actual results. As we can see from the table 3, VUS440 contains less DSP but much more LUT (5.8X) than VC709. In general, VUS440 platform present the less overall resource utilization, which can be allowed to work at higher clock frequency.
Table 3. FPGA resource utilization.

| Device | Available | DSP | BRAM | LUT | FF |
|--------|-----------|-----|------|-----|----|
| VC709  | Available | 3600| 2940 | 433K| 866K|
|        | Utilization (m=2) | 1402(39%) | 1352(46%) | 199K(46%) | 225K(26%) |
|        | Utilization (m=4) | 756(21%) | 1376(47%) | 326K(75%) | 438K(51%) |
| VUS440 | Available | 2880| 5040 | 2518K | 5037K|
|        | Utilization (m=2) | 1402(49%) | 1352(27%) | 199K(8%) | 225K(4%) |
|        | Utilization (m=4) | 756(26%) | 1376(27%) | 326K(13%) | 438K(9%) |

Table 4. Comparison with prior FPGA implementations.

|                      | [2]       | [7]       | [8]       | [4]       | Ours |
|----------------------|-----------|-----------|-----------|-----------|------|
| FPGA                 | Arria10  | Zynq XCV7045 | Stratix-V GSD8 | Ultrascale KU060 | Virtex 690T | Virtex690T | VCU440 |
| CNN                  | AlexNet  | VGG       | VGG       | VGG       | VGG   | VGG   | VGG   |
| Freq(MHz)            | 303       | 150       | 120       | 200       | 150   | 150   | 200   |
| Precision            | 16-bit float | 16-bit fixed | 16-bit fixed | 16-bit fixed | 16-bit fixed | 16-bit fixed | 16-bit fixed |
| DSP Utilization      | 1476      | 780       | 1963      | 1058      | 2833  | 756   | 756   |
| Throughput (GOPS)    | 1382      | 137       | 117       | 266       | 354   | 646   | 943   |
| Efficiency (GOPS/DSPs)| 0.93     | 0.18      | 0.06      | 0.25      | 0.12  | 0.86  | 1.2   |

5.3. Comparison with CPU and GPU
In this section, we compare our FPGA design with CPU and GPU solution. Table 5 shows the comparison between our accelerator for \(F(4 \times 4,3 \times 3)\) and CPU and GPU. CPU and GPU solution are optimized by using OpenBLAS and CuDNN libraries. On VC709 platform, our accelerator achieves an average performance of 11x over CPU solution and 5.7x energy efficiency over GPU solution. VUS440 platform presents better results, where our implementation achieves a 16x speedup over CPU and 8x energy efficiency over GPU.

Table 5. Comparison with CPU/GPU platform.

|                      | CPU       | GPU       | FPGA     |
|----------------------|-----------|-----------|----------|
| Device               | E5-2680   | K40       | VC709    | VUS440   |
| Technology           | 22nm      | 28nm      | 28nm     | 20nm     |
| Freq                 | 2.8GHz    | 1GHz      | 150MHz   | 200MHz   |
| Power(W)             | 115       | 250       | 25       | 25       |
| VGG average(GOPS)    | 58.7      | 1174.0    | 646      | 943      |
| Speed up             | 1x        | 20.0x     | 11.0x    | 16.0x    |
| Energy efficiency (GOPS/W) | 1x        | 9.2x      | 51.0x    | 74.5x    |

6. Related Works
Recently, there have been several FPGA-based accelerators for CNN. Among these, many accelerators are implemented with conventional convolution algorithm. Communication and computation are the main constraint of accelerator design. The work in [3] build a roofline model to maximize the compute resource. Many works focus on implementing CNNs layer by layer, which lead to bottleneck of memory access. The work in [9] propose a design of fused-layer CNN accelerator to minimize the off-chip data transfer. In [7], the author proposes a dynamic-precision data quantization to improve the bandwidth and resource utilization. Some works also use fast algorithm to reduce the computational complexity. The work in [10] designs an accelerator based on FFT. The work in [2] and [11] use fixed tile size of Winograd algorithm to design the accelerators. In our work, we use Vivado HLS to implement our accelerator design based on Winograd algorithm and evaluate the performance of different sizes.

7. Conclusion
In this work, we present a FPGA-based accelerator design based on Winograd algorithm. Moreover, our designed architecture applies to both convolutional layers and fully connected layers. We also build an analytical model to estimate the performance and resource utilization. In our experiment, we use VGG16 to evaluate our accelerator with different tile size. Synthesizing the overall performance and resource utilization, the design for $F(4 \times 4, 3 \times 3)$ achieves better performance.

8. Acknowledgment
The authors gratefully acknowledge supports from National Key Research and Development program under No.2016YFB1000401 and No.2016YFB1000403; National Nature Science Foundation of China under NSFC No.61502509 and 61402504.

9. References
[1] Yang K, Dou Y, Lv S, Zhang F and Lv Q 2016 Relative distance features for gait recognition with Kinect J. Vis. Commun. Image. R 39 p 209-217
[2] Aydonat U, O’Connell S, Capalija D, Ling AC and Chiu GR 2017 An OpenCL(TM) Deep Learning Accelerator on Arria 10 Field-Programmable Gate Arrays
[3] Zhang C, Li P, Sun G, Guan Y, Xiao B and Cong J 2015 Optimizing fpga-based accelerator design for deep convolutional neural networks Proc. of the 2015 ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays (New York: ACM) p 161-170
[4] Zhang C, Fang Z, Zhou P, Pan P and Cong J 2016 Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks Int. Conf. on Computer-Aided Design (Austin) p 1-8
[5] Farabet C, Poulet C, Han J Y and LeCun Y 2009 Cnp: An fpga-based processor for convolutional networks Int. Conf. on Field Programmable Logic and Applications (Prague: IEEE) p 32-37
[6] Ma Y, Cao Y, Vrudhula S and Seo J S 2017 Optimizing Loop Operation and Dataflow in FPGA Acceleration of Deep Convolutional Neural Networks Proc. of the 2017 ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays (New York: ACM) p 45-54
[7] Qiu J, et al 2016 Going deeper with embedded fpga platform for convolutional neural network Proc. of the 2016 ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays (New York: ACM) p 26-35
[8] Suda N, Chandra V, Dasika G, Mohanty A, Ma Y, Vrudhula S, Seo J and Cao Y 2016 Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks Proc. of the 2016 ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays (New York: ACM) p 16-25
[9] Alwani M, Chen H, Ferdman M and Milder P 2016 Fused-layer CNN accelerators 49th Annual IEEE/ACM Int. Symp. on Microarchitecture (Taipei) p 1-12
[10] Zhang C and Prasanna V K 2017 Frequency Domain Acceleration of Convolutional Neural Networks on CPU-FPGA Shared Memory System FPGA p 35-44
[11] DiCecco R, Lacey G, Vasiljevic J, Chow P, Taylor G and Areibi S 2016 Caffeinated FPGAs: FPGA Framework For Convolutional Neural Networks Int. Conf. on Field-Programmable Technology (Xi'an) p 265-268
[12] Liu Z, Dou Y, Jiang J, Xu J, Li S, Zhou Y and Xu Y 2017 Throughput-Optimized FPGA Accelerator for Deep Convolutional Neural Networks ACM. T. Reconfig. Techn 10 p 17
[13] Chen T, Du Z, Sun N, Wang J, Wu C, Chen Y and Temam O 2014 Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning Proc. of the 19th Int. conf. on Architectural support for programming languages and operating systems (New York: ACM) p 269-284
[14] Lavin A and Gray S 2016 Fast algorithms for convolutional neural networks Proc. of the IEEE Conf. on Computer Vision and Pattern Recognition p 4013-4021
[15] Cong J and Xiao B 2014 Minimizing computation in convolutional neural networks Int. Conf. on artificial neural networks (Switzerland) p 281-290
[16] Winograd S 1980 Arithmetic complexity of computations vol 33, ed J W Arrowsmith L and Bristol E (Philadelphia: Siam)
[17] Winograd S 1980 On multiplication of polynomials modulo a polynomial SIAM J. Comput, (Philadelphia: Siam) p 225-229