Evaluating Fast Algorithms for Convolutional Neural Networks on FPGAs

Conference Paper · April 2017

DOI: 10.1109/FCCM.2017.64

4 authors, including:

Liqiang Lu
Peking University
2 PUBLICATIONS **3** CITATIONS

SEE PROFILE

2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines

Evaluating Fast Algorithms for Convolutional Neural Networks on FPGAs

Liqiang Lu*[1,3], Yun Liang†[1], Qingcheng Xiao[1], Shengen Yan[2,3]
[1]*Center for Energy-efficient Computing and Applications, Peking University, Beijing, China*
[2]*Department of Information Engineering, The Chinese University of Hong Kong.*
[3]*SenseTime Group Limited.*
Email: {liqianglu, ericlyun, walkershaw}@pku.edu.cn, yanshengen@sensetime.com

***Abstract*—In recent years, Convolutional Neural Networks (CNNs) have become widely adopted for computer vision tasks. FPGAs have been adequately explored as a promising hardware accelerator for CNNs due to its high performance, energy efficiency, and reconfigurability. However, prior FPGA solutions based on the conventional convolutional algorithm is often bounded by the computational capability of FPGAs (e.g., the number of DSPs). In this paper, we demonstrate that fast Winograd algorithm can dramatically reduce the arithmetic complexity, and improve the performance of CNNs on FPGAs. We first propose a novel architecture for implementing Winograd algorithm on FPGAs. Our design employs line buffer structure to effectively reuse the feature map data among different tiles. We also effectively pipeline the Winograd PE engine and initiate multiple PEs through parallelization. Meanwhile, there exists a complex design space to explore. We propose an analytical model to predict the resource usage and reason about the performance. Then, we use the model to guide a fast design space exploration. Experiments using the state-of-the-art CNNs demonstrate the best performance and energy efficiency on FPGAs. We achieve an average 1006.4 GOP/s for the convolutional layers and 854.6 GOP/s for the overall AlexNet and an average 3044.7 GOP/s for the convolutional layers and 2940.7 GOP/s for the overall VGG16 on Xilinx ZCU102 platform.**

I. INTRODUNCTION

Deep convolutional neural networks (CNNs) have achieved remarkable performance for various computer vision tasks including image classification, object detection, and semantic segmentation [1, 2]. 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 [3, 4]. Towards such overwhelming computation pressure, hardware accelerators such as GPUs, FPGAs, and ASICs have been employed to accelerate CNNs [5–17]. Among the accelerators, FPGAs have emerged as a promising solution due to its high performance, energy efficiency, and reprogramability. More importantly, High Level Synthesis (HLS) using C or C++ has greatly lowered the programming hurdle of FPGAs and improve the productivity [18–20].

A CNN typically involves multiple layers, where the output feature maps of one layer are the input feature maps of the following layer. Prior studies have shown that the computation of the state-of-the-art CNNs are dominated by the convolutional layers [6, 7]. Using the conventional convolution algorithm, each element in the output feature map is computed individually by using multiple multiply-accumulate operations. While the prior FPGA solutions of CNNs using this algorithm have demonstrated preliminary success [5–9, 11], greater efficiency is possible when the algorithm itself can be more efficient. In this paper, we show how convolution using Winograd algorithm [21] can dramatically reduce the arithmetic complexity, and improve the performance of CNNs on FPGAs. Using Winograd algorithm, a tile of elements in the output feature map are generated together by exploiting the structural similarity among them. This helps to cut down the arithmetic complexity by reducing the required number of multiplications. It has been demonstrated that fast Winograd algorithm can be used to derive efficient algorithms for CNNs with small filters [16].

More importantly, the current trend of CNNs is towards deeper topologies with small filters. For example, all convolutional layers of Alexnet employ 3×3 and 5×5 filters except the first layer [3]; VGG16 only uses 3×3 filters [22]. This opens up the opportunities of using Winograd algorithm for efficient implementation of CNNs. However, although using Winograd algorithm on FPGAs is appealing, several problems remain. First, it is crucial that the design can not only minimize the memory bandwidth requirement but also match the memory throughput with the computation engines. Second, there exists a large design space when mapping the Winograd algorithm onto FPGAs. It is very difficult to reason about which designs will improve or harm the performance.

In this paper, we design a line-buffer structure to cache the feature maps for Winograd algorithm. This allows different tiles to reuse the data when the convolution operations progress. The computation of Winograd algorithm involves a mixed matrix transformation of general purpose matrix multiplication (GEMM) and element-wise multiplication (EWMM). Then, we design an efficient Winograd PE and

*Work done while the author interned at Sensetime.
†Corresponding Author.

978-1-5386-4037-1/17 $31.00 © 2017 IEEE
DOI 10.1109/FCCM.2017.64

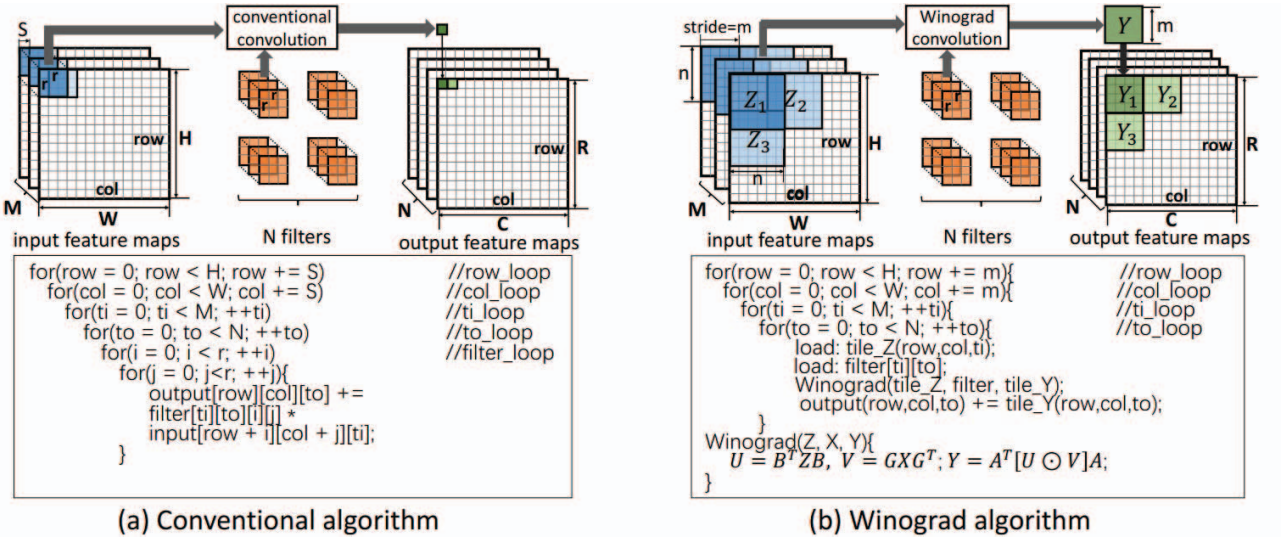

Figure 1: Comparison of conventional and Winograd convolution algorithms. We assume the stride S is 1 for Winograd algorithm .

initiate multiple PEs through parallelization. Finally, we develop analytical models to estimate the resource usage and predict the performance. We use the models to explore the design space and identify the optimal design parameters.

This paper makes the following contributions.

- We propose an architecture for efficient implementation of CNNs using Winograd algorithm on FPGAs. The architecture employs line-buffer structure, general purpose and element-wise matrix multiplication for Winograd PE, and PE parallelization.
- We develop analytical resource and performance models and use the models to explore the design space to identify the optimal parameters.
- We perform rigorous validation of our techniques using the state-of-the-art CNNs including AlexNet and VGG16.

Experiments using the state-of-the-art CNNs demonstrate the best performance and energy efficiency of CNNs on FPGAs. We achieve an average 1006.4 GOP/s for the convolutional layers and 854.6 GOP/s for the overall AlexNet and an average 3044.7 GOP/s for the convolutional layers and 2940.7 GOP/s for the overall VGG16 on Xilinx ZCU102 platform. This comes to 36.2 GOP/s/W energy efficiency for AlexNet and 124.6 GOP/s/W energy efficiency for VGG16.

II. BACKGROUND

A. CNN Basics

In general, CNNs is composed of a series of layers and each layer in turn is composed of input feature maps, filters and output feature maps. Among these layers, convolutional layers account for the major computation. CNNs are trained off-line and FPGAs are mainly used for accelerating the inference phase [5, 7, 8, 23]. Figure 1(a) presents a typical convolutional layer and its implementation using conventional algorithm. With the conventional convolution algorithm, each element in the output feature map is computed individually by multiplying and accumulating the corresponding input feature data with filters.

B. 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 [21].

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:

$$In = [z_0\ z_1\ z_2\ z_3]^{\mathrm{T}}\ F = [x_0\ x_1\ x_2]^{\mathrm{T}}\ Out = [y_0\ y_1]^{\mathrm{T}}$$

$$\begin{bmatrix} z_0 & z_1 & z_2 \\ z_1 & z_2 & z_3 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} m_1 + m_2 + m_3 \\ m_2 - m_3 + m_4 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \end{bmatrix} \quad (1)$$

m_1,m_2,m_3,m_4 are:

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

Only 4 multiplications are necessary for computing $m_1 - m_4$. The one-dimensional convolution using Winograd algorithm can be formulated using the transformation matrices A, B and G as follows,

$$Out = A^{\mathrm{T}}[(GF) \odot (B^{\mathrm{T}} In)] \quad (3)$$

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

where $\odot$ is element-wise multiplication (EWMM). In this paper, 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 $n \times n$ ($n = m + r - 1$). The output tile can be derived as follows,

$$\begin{aligned} Out &= A^{\mathrm{T}} \left[U \odot V \right] A \\ U &= GFG^{\mathrm{T}} \quad V = B^{\mathrm{T}} InB \end{aligned} \tag{4}$$

By defining the transformation matrices A, B, and G, we can formulate the 2-D Winograd algorithm as a mixed general purpose and element-wise matrix multiplication as shown in Figure 1(b). The transformation matrices are generated offline once the n and r are determined. In our implementation, the multiplication with the constants in the transformation matrices are converted to shift operations (like $\times\frac{1}{2}$), which is more efficient and uses only LUT and Flip Flops on FPGAs.

As shown in Figure 1(b), each time Winograd algorithm is called, it generates a tile of size $m \times m$ together. The number of multiplications is determined by $\odot$ in Equation 4. To compute the $m \times m$ tile in the output feature map, Wingograd algorithm requires n^2 multiplications while the conventional algorithm requires $m^2 r^2$ multiplications. For example, for a 4×4 output tile generated by convolving a 6×6 input tile with a 3×3 filter, conventional convolution needs $4^2 \times 3^2 = 144$ 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.

III. Architecture Design

In this paper, we propose a FPGA accelerator design for CNNs based on two-dimensional Winograd algorithm. Defying conventional convolution algorithm where each element in the output feature map is computed individually, Winograd algorithm can generate a tile of output feature maps together by exploiting the structural similarity among the elements in the same tile of the input feature map. More clearly, given a size $n \times n$ input tile and $r \times r$ filter, we employ Winograd algorithm to generate a size $m \times m$ ($n = m + r - 1$) output feature map. To derive the next $m \times m$ tile of output feature map, we just need to slide the input tile by m and perform the same Winograd computation as shown in the Figure 1 (b).

Several challenges arise when designing and implementing the Winograd algorithm based CNN accelerator on FPGAs. First, the convolution layers have high memory bandwidth demand. We observe that the neighboring tiles share input feature map data both horizontally and vertically. We leverage on this observation to design line buffers to maximize the data reuse (Section III-B). Second, different from the conventional convolution algorithm, Winograd algorithm generates a tile of output feature maps at a time. This requires all the elements in the input tiles and filters are ready at the same time before the Winograd transformation starts. We design an efficient PE engine for the Winograd algorithm (Section III-C) and instantiate multiple PEs through parallelization (Section III-D). Third, different implementation parameters (tile size, parallelization degree) form a large design space with multiple dimension resource and bandwidth constraints. We propose an analytical model for performance prediction and leverage it to explore the space efficiently (Section III-E).

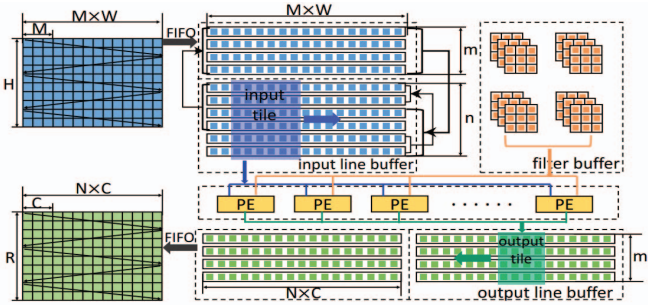

Figure 2: Architecture overview

A. Architecture Overview

Figure 2 presents the architecture overview of convolutional layer based on Winograd algorithm on FPGAs. 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 PEs fetch data from line buffers. Concretely, given an $n \times n$ input tile, a Winograd PE will generate an $m \times m$ output tile. We initiate an array of PEs by parallelizing the processing of the multiple channels. Finally, we use double 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. However, the size of the filters increases significantly as the network goes deeper. It is unpractical to load all the filters to on-chip memory. In our design, we split the input and output channels into several groups. Each group only contains a portion of filters. We load the filters group by group when they are needed. In the following, we assume there is only one group for easy illustration.

B. Line Buffer Design

There exist data reuse opportunities both horizontally and vertically. Clearly, two neighboring tiles share $(r - 1) \times n$ elements for each input feature map as shown in Figure 1(b). To exploit the data reuse opportunities, we store a few lines in the on-chip memory. Each input line buffer contains $M \times W$ elements, where M is the number of input channels and W is the width of the input feature maps as shown in Figure 2. Each output line buffer contains $N \times C$ elements, where N is the number of output channels and C is the width of the output feature maps as shown in Figure 1 (b). However, different layers may have different feature map

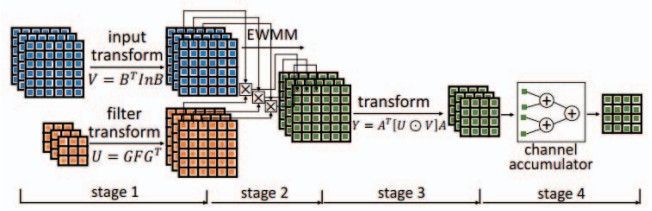

Figure 3: Winograd PE design

width and channels. In practice, We set W as the maximal width of all the feature maps.

To reuse the data, we store $n+m$ input lines in on-chip memory in total and rotate the lines as a circular buffer. More clearly, initially, Winograd engines will read the first n lines from the line buffer directly, meanwhile the next m lines of the line buffer will load data from external memory. The computation of the n lines and the transfer of m lines are done in parallel by employing the double buffer design. Note that the stride between two neighboring tiles in Winograd algorithm is m. Therefore, Winograd PE engines will skip the next m lines and process the following n lines from the line buffer and the skipped m lines will be overwritten by the new load data from the external memory. During this process, if it reaches the bottom of the line buffer, it will rotate to the beginning of the line buffer.

C. Winograd PE Design

Figure 3 gives the dataflow of our Winograd PEs. We divide the Winograd algorithm in Figure 1 (b) into 4 stages so that different tiles can be effectively overlapped through pipelining. The transformation matrices (A, B, G) are computed offline once the Winograd tile size is determined. In stage 1, the input tiles and filters are transformed. Note that the filter transformation can be done offline. The reason we choose to transform it online is to save the on-chip BRAM resources. Moreover, this will not cause extra delay as the transformation of input and filter can be done in parallel as they are independent. In stage 2, we use an array of DSPs to perform the EWMM computation. In stage 3, we perform additional transformation. Finally, in stage 4, we accumulate the output tiles from different input channels.

The Winograd algorithm in Figure 1 (b) is implemented using local buffer to store the transformation matrices. In our implementation, we completely partition the transformation and intermediate matrices to registers. This helps to improve the memory bandwidth as it alleviates the memory bank conflicts. Note that when we multiply the constants in the transformation matrix with the input and filters, we do not use the DSPs. Instead, we implement the multiplication with constants using shift operations, which are implemented as Look Up Table (LUT) arrays on FPGAs.

D. PE Parallelization

To initiate an array of PEs, we can parallelize the row and column of the input feature maps, and the input and output channels. This corresponds to parallelizing/unrolling the four loops (row, col, ti, to) surrounding the Winograd engine in Figure 1 (b). We choose not to parallelize the *row_loop* as it will significantly increases the size of line buffers. Different parallelization strategies of the other three loops can lead to different data sharing and throughput [7]. Similar to [7], we choose to parallelize the *ti_loop* and *to_loop* loops as the parallelization of *col_loop* can lead to serious memory bank conflicts. We define the unroll factors of *ti_loop* and *to_loop* are P_m and P_n, respectively. Therefore, there are a total of $P_m \times P_n$ Winograd PEs in parallel. We implement the parallelization through loop unrolling.

Together with loop unrolling, we also partition the input, output and filter buffers to sustain efficient memory bandwidth. Clearly, we implement 4 dimension filters which include dimension row, column, input and output channels. We partition each dimension. We implement 2 dimension input and output buffers and partition each dimension. Table I gives the partition factors for various buffers.

Table I: Memory partition factors.

buffers	Column	Row	Input channels	Out channels
filter	r	r	P_m	P_n
input	n	-	P_m	-
output	m	-	-	P_n

E. Design Space Exploration

Our Winograd implementation involves a few design parameters, input tile size (n), and parallelization degree (P_m and P_n). Given an input tile size n, since the filter size is fixed for a neural network layer (e.g., 3×3, 5×5), the output tile size m can be determined ($m = n - r + 1$). These design parameters affect both the performance and accuracy. Here, we develop an analytical model that can predict the performance of Winograd algorithm on FPGAs. Then, we rely on it to explore the design space.

As mentioned in Section II-B, the multiplication saving increases as the input tile size n increases. However, the range of the constants in the transformation matrices will increase as n increases, which may cause precision loss. In this work, we use fixed-point 16 bits to represent both data and filter. We set the precision to 2^{-10} for filters to maintain a high accuracy as prior work [23]. Under this precision constraint, we set the maximum value for n to 8 as beyond this we can not precisely represent the constants in the transformation matrix.

In the following, we model the resource consumption and predict the performance for different input tile size n and parallelization degree P_m and P_n. As mentioned in Section II-B, only the EWMM operation will consume DSP. Therefore, the number of DSPs only depends on the size of input tile and parallelization degree.

$$DSP = n^2 \times P_n \times P_m \tag{5}$$

LUT is difficult to predict. Here, we approximate its consumption using linear regression models,

$$LUT = \alpha_n^r \times P_m \times P_n \tag{6}$$

where α_n^r is the LUT consumption of a single Winograd PE with the input tile size n and filter size r. α_n^r is pre-trained on different platforms.

The number of BRAM banks is computed by adding the banks for filter, input and output buffers based on the memory partition factors in Table I.

$$\begin{aligned} Banks = & r^2 \times P_m \times P_n \\ & + (n+m) \times n \times P_m \\ & + 2 \times m^2 \times P_n \end{aligned} \tag{7}$$

We also model the memory bandwidth between the on-chip and off-chip memory. To efficiently utilize the resource, the data transfer speed must be greater than or equal to the computation speed. The time to process n rows of input data in the line buffer is,

$$T_{compute} = (\lceil \frac{W}{m} \rceil \times \lceil \frac{M}{P_m} \rceil \times \lceil \frac{N}{P_n} \rceil \times II + P_{depth}) \times \frac{1}{Freq} \tag{8}$$

where $Freq$ is the operating frequency of the FPGAs. II denotes the iteration interval of the pipeline. In our implementation, loops in Figure 1 (b) are perfectly pipelined, so the $II = 1$. P_{depth} is the pipeline depth, which can be ignored when the loop trip count is large enough.

The computation is in parallel with the transfer of m rows of input and output data.

$$T_{transfer} = \frac{m \times W \times max(N, M) \times 16}{Bandwidth} \tag{9}$$

We require that $T_{transfer} \leqslant T_{compute}$. Therefore, we can get the bandwidth requirement as,

$$Bandwidth \geqslant m^2 \times \lceil \frac{P_m \times P_n}{min(N, M)} \rceil \times 16 \times freq \tag{10}$$

We define the T_{init} as the time to load the first n rows of the input image into on-chip memory and filters,

$$T_{init} = \frac{M \times N \times r \times r + n \times W \times M}{Bandwidth/16} \tag{11}$$

The total operations and processing time of the convolution are,

$$OPs = H \times W \times M \times N \times r^2 \times 2 \tag{12}$$

$$T_{total} = \lceil \frac{H}{m} \rceil \times T_{compute} + T_{init} \tag{13}$$

We define the effective performance of convolution based on Winograd algorithm as,

$$Perf_{eff} = \frac{OPs}{T_{total}} \tag{14}$$

Now, given a convolutional layer represented by $\{H, W, M, R, C, N, r\}$, our goal is to find the optimal solution $\{n,\ P_m,\ P_n\}$ to maximize the performance (Equation 14) with resources and bandwidth constraints. To solve this problem, we rely on our performance models to explore the design space and identify the optimal solution.

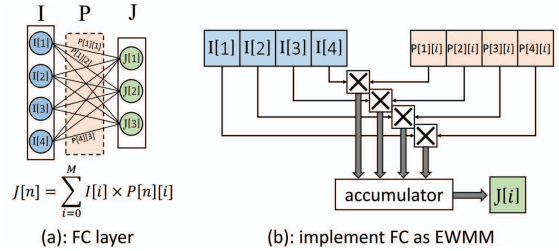

Figure 4: FC layer implementation

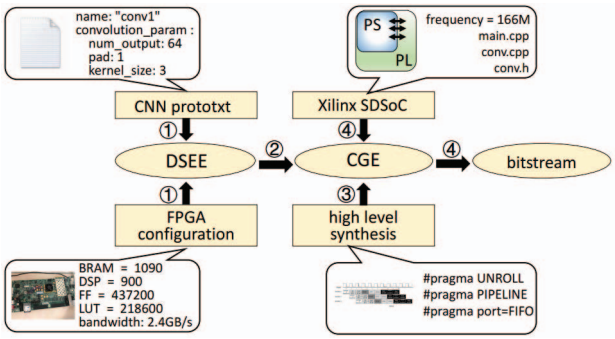

Figure 5: Automatic tool flow

F. Implementation of Other Layers

In addition to convolution layers, there are also other layers in CNNs such as Fully Connected (FC) layers, Pooling and Rectified Linear Unit (ReLU) layers. Here, we describe how to implement these layers.

FC layers connect all the neurons in the previous layer to every single neuron in the weight matrix as shown in Figure 4(a). The computation is a matrix-vector product. The operations in FC layers can be treated as EWMM by filling the input neurons and its corresponding weights into a matrix. To reuse the Winograd PE, FC layers only need to bypass the transformation stages (stage 1, 3 in Figure 3). The weights in FC layers are significantly larger than the input neurons. Therefore, similar to [8], we load the entire input neurons of FC layer into on-chip memory but stream the weights using FIFO interface. In addition, the FC computation contains no data reuse opportunities. To improve the memory bandwidth, an effective approach is to increase the batch size N_{batch} (the number of input images). Specifically, we assemble a batch of images from the previous layer, these images are processed together.

Max Pooling layers are widely used in CNNs, which output the maximum values in subregions of input feature maps. ReLU layers sets any input value less than zero to zero. ReLU and Pooling are implemented by introducing comparison operators to the output buffers.

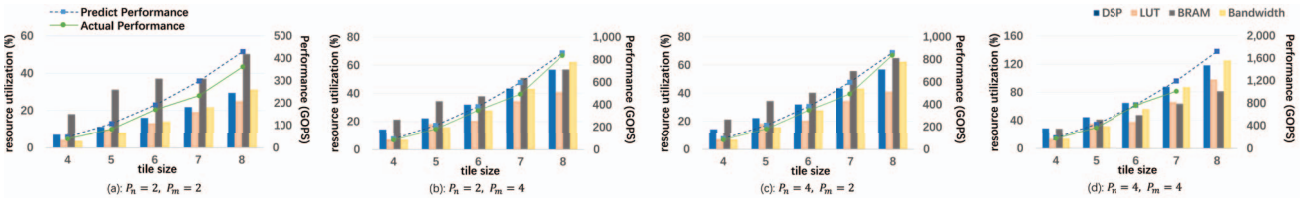

Figure 6: Resource utilization and performance results for 3×3 filter

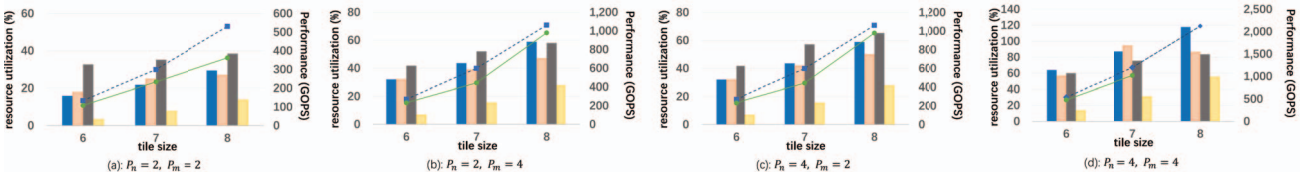

Figure 7: Resource utilization and performance results for 5×5 filter

IV. Automatic Tool Flow

We design an automatic tool flow to automate the mapping of CNNs onto FPGAs as shown in Figure 5. The flow consists of four steps. In **step 1**, CNN architecture and FPGA configuration are fed into the design space exploration engine (DSEE). Clearly, we use Caffe prototxt to describe the structure of CNNs [24]. The FPGA configuration parameters include the memory bandwidth, number of DSPs, logic cells and on-chip memory capacity. Then, the output of DSEE is the optimal solution $\{n,\ T_m,\ T_n\}$ as described in Section III-E. In **step 2**, based on the optimal solution, we develop a Code Generate Engine (CGE) which can generate the Winograd convolution functions automatically. The functions describe the whole accelerator architecture including line buffers, buffer management, and Winograd PEs. The generated implementation is HLS compatible C code. Pragmas such as memory partition factors, loop unroll factors T_n T_m and FIFO interfaces are inserted into the functions. In **step 3**, we use Xilinx HLS tool to synthesize the code into register transfer level. Finally, we use Xilinx SDSoC (software-defined system-on-chip) tool-chain to generate the bitstream.

V. Experiment Evaluation

A. Experiments Setup

We evaluate our techniques on two FPGA platforms: Xilinx ZC706 and ZCU102. Xilinx ZC706 platform consists of a Kintex-7 FPGA and dual ARM Cortex-A9 processors. The external memory is 1 GB DDR3. Our FPGA implementation is operated at 166MHz frequency on this platform. Xilinx ZCU102 consists of an UltraScale FPGA, quad ARM Cortex-A53 processors, 500 MB DDR3. Our FPGA implementations is operated at 200MHz frequency on this platform. To measure the runtime power, we plugged a power meter in the FPGA platform.

In the following, we first present the model and resource analysis results for a typical convolution layer (Section V-B). Then, we perform case studies using the state-of-the-art CNNs including AlexNet and VGG16 (Section V-C). It should be noted that the performance we report in the following is the effective performance. It is computed by dividing the total operations by the total processing time (Equation 14). For conventional algorithm, the effective performance is always bounded by $MaxF$, the maximum computational capability of the FPGA platform. $MaxF = DSP \times Freq \times 2$, where 2 means multiply and add operations. However, for Winograd algorithm, the effective performance can exceed the $MaxF$ as Winograd algorithm can increase the effective DSP efficiency by reducing the number of multiplications required by convolution.

B. Model and Resource Analysis

In this subsection, we evaluate our analytical models and analyze the resource usage of Winograd algorithm using a single convolutional layer. We use a typical input feature map size: $224(H) \times 224(W)$ and try two different filter sizes: 3×3 and 5×5. Figure 6 and Figure 7 compare the predict and actual performance for different input tile size and parallelization degree, and give the corresponding resource utilization. The experiments are performed on Xilinx ZC706. We can see that our performance prediction is very accurate. On average, the prediction error is 15.4% and 13.7% for filters 3×3 and 5×5, respectively. The sources of the inaccuracy may come from the discrepancy of actual and peak bandwidth and DDR access latency.

Thanks to the Winograd algorithm, DSP is no longer the limiting resource for most cases as shown by Figure 6 and 7. Instead, BRAMs and memory bandwidth can be the limiting resources. The BRAMs consumption comes from a few aspects. First, unlike the conventional convolution, Winograd convolution requires more buffers because of the line buffer structure. Second, paralleling Winograd PEs requires memory partition to sustain the on-chip memory bandwidth. Finally, when the computation efficiency improves, the off-chip bandwidth might become the bottleneck. Overall, Winograd algorithm saves the DSPs and improves the overall resource utilization.

C. Case Study

Here, we evaluate our Winograd implemention using AlexNet and VGGNet. Table II gives the parameters for each network in our implementation.

Table II: Design parameters

ZC706 (ZCU102)	n	P_n	P_m	N_{batch}
Alexnet(3×3)	6(6)	2(4)	8(8)	32(128)
VGG16(3×3)	7(6)	4(4)	4(16)	32(128)

1) AlexNet: AlexNet consists of five convolution and three FC layers [3]. The input image is 224×224. All the convolution layers use small filters (5×5 and 3×3) except the first convolution layer (11×11). For the first layer, We choose to use the conventional convolution algorithm for implementation. For the rest layers, we use a uniform 3×3 filter for Winograd algorithm. For the 5×5 filter, we implement it using four 3×3 filters with zero padding.

Table III gives the results. [7] only gives the convolution implementation without FC layers and [5] only gives the overall CNN performance without the detailed results for each convolutional layer. Compared to prior work [7], we improve the average convolution performance from 61.6 GOP/s to 1006.4 GOP/s[1]. For the overall CNN, we improve the performance from 72.4 GOP/s to 854.6 GOP/s compared to [5].

Table III: Performance comparison for Alexnet

	[7]	[5]	Our Impl	Our Impl
Precision	32bits float	16bits fixed	16bits fixed	16bits fixed
Device	VX485T	GSD8	ZC706	ZCU102
Freq (MHz)	100	120	167	200
Logic cell (K)	485.7	695	350	600
DSP[2]	2800	1963	900	2520
BRAM (Kb)	2060×18	2567×20	1090×18	1824×18
conv1 (GOP/s)	27.5	-	83.1	409.6
conv2 (GOP/s)	83.8	-	501.7	1355.6
conv3 (GOP/s)	78.8	-	610.2	1535.7
conv4 (GOP/s)	77.9	-	401.2	1361.7
conv5 (GOP/s)	77.6	-	355.6	1285.7
conv average (GOP/s)	61.6	-	271.8	1006.4
CNN average (GOP/s)	-	72.4	201.4	854.6
Power (W)	18.6	19.1	9.4	23.6
DSP Efficiency (GOP/s/DSPs)	0.022	0.037	0.224	0.339
Logic cell Efficiency (GOP/s/cells/K	0.127	0.104	0.575	1.424
Energy Efficiency (GOP/s/W)	3.31	3.79	21.4	36.2

To make a fair comparison across different platforms. We also present the total resource efficiency and energy efficiency on each platform. In Table III, we can see that our implementation achieves better resource efficiency, which comes from the reduction of arithmetic complexity and novel architecture. Our implementation also improves the energy efficiency from 3.79 GOP/s/W to 36.2 GOP/s/W.

2) VGGNet: In VGG16 [22], all convolutional layers are with 3×3 filters, which fit well for Winograd algorithm. VGG16 consists of 5 convolution groups with different input size (224, 112, 56, 28, 14). Table IV compares our techniques with prior works. For the convolutional layers, we improve the average performance from 136.5 - 488 GOP/s to 3044.7 GOP/s compared to [5, 8, 23]. For the overall CNN, we improve the performance from 117.8 - 354 GOP/s to 2940.7 GOP/s.

Similar to AlexNet experiments, we also measure the resource efficiency and energy efficiency. Similar findings hold for VGG16. We notice that we achieve higher performance for VGG16 than AlexNet. This is because VGG16 uses uniform convolution structure, while AlexNet uses two different convolution structures. We also find that the performance of convolutional layer decreases as the network goes deeper. This is due to the fact that the initial time (T_{init}) accounts for more total time (T_{total}) and the initial time only involves data transfer without actual computation.

Table IV: Performance comparison for VGG

	[23]	[5]	[8]	Our Impl
Precision	16bits fixed	16bits fixed	16bits fixed	16bits fixed
Device	ZC706	GSD8	XC7VX690T	ZCU102
Freq (MHz)	150	120	150	200
Logic cell (K)	350K	695K	693K	600K
DSP	900	1963	3600	2520
BRAM (Kb)	1090×18	2567×20	2940×18	1824×18
conv1 (GOP/s)	123.8	-	320	2734.7
conv2 (GOP/s)	235.3	-	635	3212.4
conv3 (GOP/s)	235.3	-	600	3111.1
conv4 (GOP/s)	254.8	-	585	3069.3
conv5 (GOP/s)	70.2	-	400	2431.4
conv average (GOP/s)	187.8	136.5	488	3044.7
CNN average (GOP/s)	137.0	117.8	354	2940.7
Power (W)	9.6	-	25	23.6
DSP Efficiency (GOP/s/DSPs)	0.152	0.06	0.10	1.16
Logic cell Efficiency (GOP/s/cells)	0.391	0.196	0.511	4.901
Energy Efficiency (GOP/s/W)	14.3	-	14.2	124.6

D. Comparison with GPU

In this subsection, we conduct a comparison between GPU and FPGA platforms. For GPUs, we measure the performance of VGG16 using Caffe framework [24] on NVIDIA TitanX platform. To make a fair comparison, we test the performance of TitanX with the latest CuDNN 5.1 [25] as Winograd algorithm is also included in CuDNN 5.1. Power on GPU is obtained using NVIDIA profiling tools. Table V shows the comparison results. As shown, TitanX

[1]In [7], FC layer is not implemented. So the efficiency value of [7] is calculated based on the average performance of convolution.

[2]In Xilinx ZC706 (Kintex-7) Platform, a single DSP(DSP48E1) slice can be implemented as one 18×25 fixed-point multiplier. In Altera GSD8 (Stratix-V) Platform, a single DSP slice can be implemented as two 18×18 fixed-point multipliers

Table V: Comparison with GPU platform

Device	TitanX[1]	TitanX[2]	ZC706	ZCU102
Technology	28 nm	28 nm	28 nm	16 nm
Precision	32bits float	32bits float	16bits fixed	16bits fixed
CNN average (TOP/s)	4.98	5.60	0.67	2.94
Power (W)	130	134	9.4	23.6
Energy efficiency (GOP/s/W)	38.3	41.8	72.3	124.6

[1] We use the default implementation in Cudnn5.1, selected layers will call Winograd algorithm.
[2] We force every layer to use the Winograd algorithm.

gives better better performance, but our implementation on Xilinx ZCU102 FPGA achieves much better (2.98X) energy efficiency.

VI. Related Work

Recently, FPGAs are gaining popularity for use as accelerators for deep learning tasks due to its high performance, low power and reconfigurability. Most FPGA accelerators focus on the implementations of convolutional layers using the conventional algorithms[5, 7, 10, 11, 14, 23]. Zhang [7] et al. propose a design space exploration technique to optimize the throughput from computation resources and bandwidth aspects. Qiu et al. [23] propose a dynamic-precision data quantization to increase DSP efficiency. Several other studies target a uniform implementation for convolutional layer and FC layer [5, 8, 15]. In [5], 3D convolution operations is flattened as 2D general purpose matrix multiplication, which is widely adopted on GPU platforms. But on FPGAs, it can result in massive memory usage. Zhang [8] et al. present an uniform representation for convolutional layers and FC layers, which can share the same computing resources. Song [15] et al. propose a general purpose accelerator using kernel-partition method.

A few studies also focus on reducing the arithmetic complexity of convolution [10, 16, 26, 27] using non-conventional algorithms. Zhang et al. [10] reduce the computation using low-rank approximation which is based on minimizing the reconstruction error of nonlinear response. Lavin [16] evaluates Fast Fourier Algorithm (FFT) and Winograd algorithm on GPU platforms. But FFT shows less efficiency for convolutions with small filters. [26] implements FFT on FPGA platform for CNN. But it shows little reduction of computation complexity with small filters like 3×3. Aydonat et al. [27] apply Winograd algorithm on Arria 10 FPGA platform. But they only use 1-D Winograd to reduce arithmetic complexity. In our work, we evaluate 2-D Winograd algorithm on FPGA platforms and use line-buffer structure to enable data reuses and performance models to guide design space exploration.

VII. Conclusion

FPGAs have been widely used to accelerate CNN-based applications. However, prior implementations based on the conventional convolutional algorithms are mainly limited by the computational capability on FPGAs. In this work, we propose a CNN architecture on FPGAs based on Winograd algorithm, which can effectively reduce the arithmetic complexity. We also develop analytical models to estimate the resource usage and performance. Our implementations of Alexnet and VGG16 achieve the overall performance of 854.6 GOP/s and 2940.7 GOP/s, respectively on ZCU102 FPGA platform, which outperforms all previous work.

VIII. Acknowledgement

We thank Qian Li for her help in GPU experiment.

References

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *ICCV*, 2015.
[2] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *CVPR*, 2014.
[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *NIPS*, 2012.
[4] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *arXiv preprint arXiv:1512.03385*, 2015.
[5] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-s. Seo, and Y. Cao, "Throughput-Optimized OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks," in *FPGA*, 2016.
[6] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *CVPR*, 2015.
[7] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *FPGA*, 2015.
[8] C. Zhang, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: towards uniformed representation and acceleration for deep convolutional neural networks," in *ICCAD*, 2016.
[9] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun, "Cnp: An FPGA-based processor for convolutional networks," in *FPL*, 2009.
[10] X. Zhang, J. Zou, X. Ming, K. He, and J. Sun, "Efficient and accurate approximations of nonlinear convolutional networks," in *CVPR*, 2015.
[11] M. Sankaradas, V. Jakkula, S. Cadambi, S. Chakradhar, I. Durdanovic, E. Cosatto, and H. P. Graf, "A massively parallel coprocessor for convolutional neural networks," in *ASAP*, 2009.
[12] Y. H. Chen, J. Emer, and V. Sze, "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks," in *ISCA*, 2016.
[13] S. Liu, Z. Du, J. Tao, D. Han, T. Luo, Y. Xie, Y. Chen, and T. Chen, "Cambricon: an instruction set architecture for neural networks," in *ISCA*, 2016.
[14] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, "A dynamically configurable coprocessor for convolutional neural networks," in *ACM SIGARCH Computer Architecture News*, 2010.
[15] L. Song, Y. Wang, Y. Han, X. Zhao, B. Liu, and X. Li, "C-brain: a deep learning accelerator that tames the diversity of CNNs through adaptive data-level parallelization," in *DAC*, 2016.
[16] A. Lavin, "Fast algorithms for convolutional neural networks," *arXiv preprint arXiv:1509.09308*, 2015.
[17] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "DianNao:a small-footprint high-throughput accelerator for ubiquitous machine-learning," *Acm Sigplan Notices*, 2014.
[18] Y. Liang, K. Rupnow, Y. Li, D. Min, M. N. Do, and D. Chen, "High-level synthesis: productivity, performance, and software constraints," *ECE*, 2012.
[19] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for FPGAs: From prototyping to deployment," *TCAD*, 2011.
[20] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "LegUp: high-level synthesis for FPGA-based processor/accelerator systems," in *FPGA*, 2011.
[21] S. Winograd, "Arithmetic complexity of computations," 1980.
[22] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.
[23] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song *et al.*, "Going deeper with embedded FPGA platform for convolutional neural network," in *FPGA*, 2016.
[24] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *MM*, 2014.
[25] "NVIDIA cuDNN, https://developer.nvidia.com/cudnn."
[26] C. Zhang and V. Prasanna, "Frequency Domain Acceleration of Convolutional Neural Networks on CPU-FPGA Shared Memory System," in *FPGA*, 2017.
[27] U. Aydonat, S. O'Connell, D. Capalija, A. C. Ling, and G. R. Chiu, "An OpenCL™ Deep Learning Accelerator on Arria 10," in *FPGA*, 2017.