Evaluating Fast Algorithms for Convolutional Neural Networks on FPGAs

Yun Liang, *Member, IEEE*, Liqiang Lu, Qingcheng Xiao, and Shengen Yan

***Abstract*—In recent years, convolutional neural networks (CNNs) have become widely adopted for computer vision tasks. Field-programmable gate arrays (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). To address this problem, the feature maps are transformed to a special domain using fast algorithms to reduce the arithmetic complexity. Winograd and fast Fourier transformation (FFT), as fast algorithm representatives, first transform input data and filter to Winograd or frequency domain, then perform element-wise multiplication, and apply inverse transformation to get the final output. In this paper, we propose a novel architecture for implementing fast algorithms on FPGAs. Our design employs line buffer structure to effectively reuse the feature map data among different tiles. We also effectively pipeline the Winograd/FFT processing element (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 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 854.6 and 2479.6 GOP/s for AlexNet and VGG16 on Xilinx ZCU102 platform using Winograd. We achieve 130.4 GOP/s for Resnet using Winograd and 201.1 GOP/s for YOLO using FFT on Xilinx ZC706 platform.**

***Index Terms*—Convolutional neural network (CNN), fast algorithm, fast Fourier transformation (FFT), field-programmable gate array (FPGA), Winograd.**

I. Introduction

DEEP convolutional neural networks (CNNs) have achieved remarkable performance for various computer vision tasks, including image classification, object detection, and semantic segmentation [8], [9]. 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 [10], [11]. Toward such overwhelming computation pressure, hardware accelerators, such as graphics processing unit (GPU), field-programmable gate arrays (FPGAs), and application specific integrated circuit (ASIC) have been employed to accelerate CNNs [2], [12]–[24]. 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 [25]–[27].

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 is dominated by the convolutional layers [12], [13]. Using the spatial convolution algorithm, each element in the output feature map is computed individually by using multiple multiply accumulate (MAC) operations. While the prior FPGA solutions of CNNs using this algorithm have demonstrated preliminary success [1], [2], [4]–[7], [12], [13], [28]–[33]. Table I shows the resource utilization of recent FPGA accelerators. In these designs, it can be concluded that DSP is the most consumed resource since the operations in a typical CNN mostly consists of MAC units and the multiplier is usually implemented by DSP on FPGAs.

Besides the spatial convolution algorithm, some designs choose to flatten convolution to general purpose matrix multiplication (GEMM). However, this approach does not reduce the number of multiplications. With the insight that higher DSP efficiency is possible if the number of multiplications can be reduced, fast algorithms are widely used to reduce the arithmetic complexity of convolutional operation. It has been demonstrated that the Winograd Fast algorithm and classic FFT algorithm can dramatically reduce the arithmetic complexity. When applying FFT and Winograd algorithms, the input feature map and filter are transformed to the corresponding domain, then perform element-wise matrix-multiplication (EWMM). The reduction degree depends on the parameters of fast algorithms. For example, using the Winograd algorithm with 6×6 input tile size can bring $4\times$ multiplication reduction for 3×3 filters and using the FFT algorithm with 8×8 input tile size can bring $3.45\times$ multiplication reduction for 3×3 filters.

More importantly, the current trend of CNNs is toward deeper topologies with small filters. For example, VGG16 and YOLO only use 3×3 filters [34], [35], and 3×3 and 5×5 filters are widely used in Resnet and Googlenet [11], [12]. And it has been demonstrated that the fast Winograd algorithm and

Manuscript received September 23, 2018; revised December 16, 2018; accepted January 14, 2019. Date of publication February 4, 2019; date of current version March 18, 2020. This work was supported in part by the Beijing Natural Science Foundation under Grant L172004, and in part by the Municipal Science and Technology Program under Grant Z181100008918015. This paper was recommended by Associate Editor W. Zhang. *(Corresponding author: Yun Liang.)*

Y. Liang is with the School of EECS, Peking University, Beijing 100871, China, and also with the Peng Cheng Laboratory, Shenzhen 518055, China (e-mail: ericlyun@pku.edu.cn).

L. Lu and Q. Xiao are with the Center for Energy-Efficient Computing and Applications, Peking University, Beijing 100871, China (e-mail: liqianglu@pku.edu.cn; walkershaw@pku.edu.cn).

S. Yan is with the Algorithm Platform Department, SenseTime, Hong Kong (e-mail: yanshengen@sensetime.com).

Digital Object Identifier 10.1109/TCAD.2019.2897701

TABLE I
RESOURCES UTILIZATION IN PREVIOUS FPGA IMPLEMENTATIONS

	[1]	[2]	[3]	[4]	[5]	[6]	[7]
Device	Virtex-7 VX690t	Stratix-V GSD8	Zynq XC7Z045	Stratix-V GXA7	Virtex-7 VC709	Arria-10 GX 1150	Arria-10 GX 1150
Frequency (MHz)	150	120	150	100	156	150	370
CNN Type	VGG	VGG	VGG	Alexnet	Alexnet	VGG	VGG
Precision	16 bit	8-16 bit	16 bit	8-16 bit	16 bit	8-16bit	32 float
DSP Utilization (%)	-	-	89	100	60	100	86
Logic Utilization	-	-	84	52	63	38	43
On-chip RAM Utilization	-	-	87	61	65	70	46

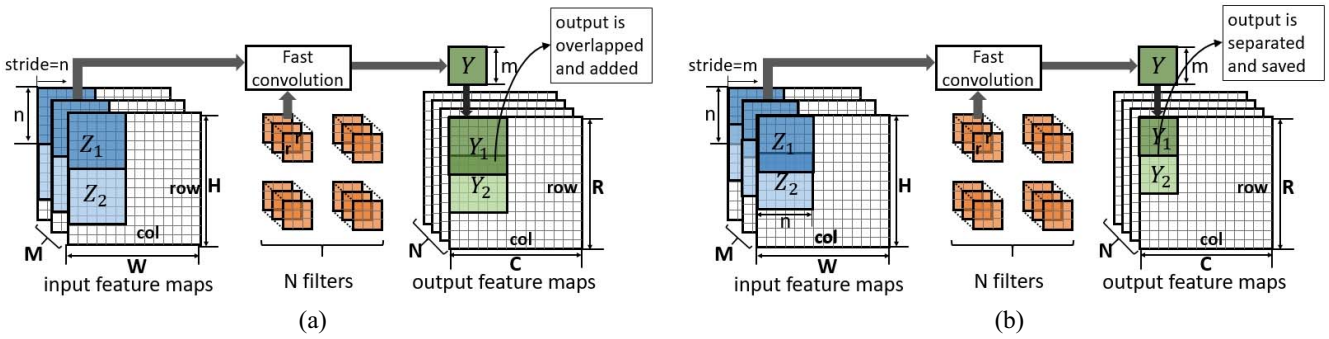


Fig. 1. Comparison of OaA and OaS dataflow. We assume the stride S is 1. (a) OaA-based fast algorithm. (b) OaS-based fast algorithm.

FFT can be used to derive efficient algorithms for CNNs with small filters [19]. This opens up the opportunities of using fast algorithms for efficient implementation of CNNs. However, using fast algorithms 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 fast algorithm onto FPGAs. It is very difficult to reason about which designs will improve or harm the performance.

In this paper, we comprehensively evaluate the Winograd fast algorithm [36] and fast Fourier transformation (FFT) algorithm which can dramatically reduce the arithmetic complexity, and improve the performance of CNNs on FPGAs. Using these two fast algorithms, 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. Then a universal framework of the FPGA accelerator is proposed. We design a line-buffer structure to cache the feature maps for the fast algorithm. Each line represents one row in the input feature maps, and we rotate the line buffers so that the input data can be reused when the filter sliding. In our design, we apply the Cooley–Turkey FFT algorithm and Winograd algorithm. We design two efficient PE architectures for both algorithms and initiate multiple PEs through parallelization. Finally, we develop analytic models to estimate resource usage and predict the performance. We use the models to explore the design space and identify the optimal design parameters.

A preliminary version of this paper was reported in [37]. In this paper, we propose an accelerator framework for both the Winograd algorithm and FFT algorithm. We design an efficient PE architecture with highly optimized FFT implementation. Besides, we generalize the performance model and resource model for both Winograd algorithm and FFT algorithm.

This paper makes the following contributions.

1) We propose a framework for efficient implementation of CNNs using Winograd and FFT algorithm on FPGAs.
2) We propose an architecture that employs line-buffer structure, efficient and fully pipelined PE, and PE parallelization.
3) We develop analytical resources and performance models and use the models to explore the design space to identify the optimal parameters. The model is integrated with an automatic tool-chain which can automatically generate the implementation of fast algorithms.

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 and 2601.3 GOP/s for the convolutional layers and 854.6 and 2479.6 GOP/s for the overall AlexNet and VGG on ZCU102 platform, respectively. This comes to 36.2 GOP/s/W energy efficiency for AlexNet and 105.4 GOP/s/W energy efficiency for VGG16. We achieve an average 163.1 and 201.1 GOP/s for the convolutional layers and 130.4 and 201.1 GOP/s for the overall Resnet and YOLO on ZC706 platform, respectively. This comes to 13.8 GOP/s/W energy efficiency for Resnet and 21.4 GOP/s/W energy efficiency for YOLO.

II. BACKGROUND

A. CNN Basics

CNN is a class of deep, feed-forward artificial neural networks in machine learning. 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. Filters which have learned features in the training process are applied to extract some certain features from input images as shown in Fig. 1(a). Among all layers in a typical CNN, convolutional layers account for the major computation. CNNs are trained offline and FPGAs are mainly used for accelerating the inference phase [2], [3], [13]. The state-of-the-art

CNN models are very large and deep which involve extensive calculation.

The spatial convolution algorithm is composed of six *for* loops as shown in the following equation:

$$\mathrm{Out}(k,i,j)=\sum_{t=1}^{M}\sum_{p=1}^{r}\sum_{q=1}^{r}F(k,t,p,q)\times \mathrm{In}(t,i*S+p,j*S+q). \tag{1}$$

Each element in the output feature map is computed individually by multiplying and accumulating the corresponding input feature data with filters. By flattening the input feature maps and rearranging the filters, the spatial convolution can be mapped to GEMM. This method will increase the local memory requirement since the pixels in the input feature maps will be copied for multiple times in flattening process. GEMM is widely used in GPU implementation because of its versatility for different layer types. However, GEMM contributes no arithmetic reduction in CNN implementation.

B. Fast Algorithms for CNNs

In addition to spatial algorithm and GEMM, fast algorithms like Winograd and FFT have been applied to fasten convolutions. Defying spatial convolution algorithm where each element in the output feature map is computed individually, the fast algorithms 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 the fast algorithms to generate a size $m\times m$ ($n=m+r-1$) output feature map. Hence, convolution based on these fast algorithms can be described by a common formula

$$\mathrm{Out}=\boldsymbol{Inverse\_Transform}\big[\boldsymbol{Transform}(\mathrm{In})\odot\boldsymbol{Transform}(F)\big] \tag{2}$$

where $\odot$ represents EWMM, In, Out, and F are input tiles, output feature maps, and filters, respectively. According to the formula, these fast algorithms are composed of the following three stages.

1) *Input and Filter Transformation:* The first stage is to transform an input tile and filter to the same shape. Winograd and FFT convolution employ different transform functions which would be introduced later.
2) *Element-Wise Multiplication:* Then in both algorithms, EWMM are performed with the obtained intermediate matrices. The tiny difference lies in that FFT convolution uses complex data so that it requires more compute resources.
3) *Inverse Transformation:* The final stage is to transform the EWMM result to the original convolution result. The inverse transform functions also differ according to functions used in the first stage.

Then we give more details about Winograd-based and FFT-based convolution separately. Note that though 1-D-Winograd and 1-D-FFT could be applied to convolution, we refer convolutions using 2-D-Winograd and 2-D-FFT as Winograd-based and FFT-based convolution, respectively.

1) Winograd-Based Convolution: Winograd documented a technique for computing polynomial multiplication which is equivalent to convolution operation. And Winograd applied the Chinese remainder theorem to produce a minimal algorithm for it.

Let us denote the result of computing m outputs by convolving the r-tap filter with inputs as $F(m,r)$. For example, the spatial algorithm for $F(2,3)$ requires $m\times r$ ($2\times 3=6$) multiplications, while the Winograd algorithm computes $F(2,3)$ in the following way, which only needs $n=m+r-1$ (4) multiplications:

$$\mathrm{In}=\begin{bmatrix}z_0 & z_1 & z_2 & z_3\end{bmatrix}^T\ F=\begin{bmatrix}x_0 & x_1 & x_2\end{bmatrix}^T\ \mathrm{Out}=\begin{bmatrix}y_0 & y_1\end{bmatrix}^T$$
$$\begin{bmatrix}z_0 & z_1 & z_2\\ z_1 & z_2 & z_3\end{bmatrix}\begin{bmatrix}x_0\\ x_1\\ x_2\end{bmatrix}=\begin{bmatrix}m_1+m_2+m_3\\ m_2-m_3+m_4\end{bmatrix}=\begin{bmatrix}y_0\\ y_1\end{bmatrix} \tag{3}$$

m_1–m_4 are

$$m_1=(z_0-z_2)x_0\quad m_2=(z_1+z_2)\frac{x_0+x_1+x_2}{2}$$
$$m_4=(z_1-z_3)x_2\quad m_3=(z_2-z_1)\frac{x_0-x_1+x_2}{2}. \tag{4}$$

Accordingly, 2-D convolution using Winograd algorithm $F(m\times m, r\times r)$ can be derived from nested $F(m,r)$ as follows:

$$\mathrm{Out}=A^T\big[\big(GFG^T\big)\odot\big(B^T\mathrm{In}B\big)\big]A \tag{5}$$

where transformation matrices A, B, and G can be derived offline once m and r are determined. Therefore, the transform functions for input and filter are

$$\mathrm{Transform}(\mathrm{In})=B^T\mathrm{In}B$$
$$\mathrm{Transform}(F)=GFG^T \tag{6}$$

and the inverse transform function is

$$\mathrm{Inverse\_Transform}(E)=A^TEA. \tag{7}$$

The constants in transformation matrices are computed using polynomial interpolation.

2) FFT-Based Convolution: FFT was proposed by Cooley–Turkey in 1995 to accelerate machine's computing speed, which can reduce constant complex multiplications from $O(N^2)$ to $O(N\log N)$ with no accuracy loss. The Cooley–Turkey 1-D-FFT is computed as follows:

$$\begin{cases}X(k)=X_1(k)+W_N^kX_2(k)\\ X\big(k+\frac{N}{2}\big)=X_1(k)-W_N^kX_2(k)\end{cases}\quad k=0,1,2,\ldots,N/2-1 \tag{8}$$

where $X_1(k)$, $X_2(k)$, W_N^k is defined as follows:

$$\begin{cases}X_1(k)=\sum_{r=0}^{N/2-1}x(2r)W_N^{2rk}\\ X_2(k)=\sum_{r=0}^{N/2-1}x(2r+1)W_N^{2rk}\end{cases}\quad W_N^k=e^{-2i\pi k/N}. \tag{9}$$

In (8) and (9), FFT is inducted from small-scale discrete Fourier transformation divided according to the odevity, which is also named butterfly computation. It should be noted that the Cooley–Turkey algorithm is only valid when FFT size is a power of 2. Fig. 2 shows a brief example of FFT algorithm with FFT size = 4. In this example, it takes two stages to complete FFT. In these stages, several butterfly computations are performed with two points (Radix-2 FFT). 2-D-FFT algorithm should be conducted from row and column dimension with 1-D-FFT respectively.

Similar to the Winograd algorithm, 2-D-FFT can be integrated in convolution. The transform functions for input and filter are

$$\mathrm{Transform}(\mathrm{In})=\mathrm{FFT}(\mathrm{In})$$
$$\mathrm{Transform}(F)=\mathrm{FFT}(\mathrm{pad}(F)) \tag{10}$$

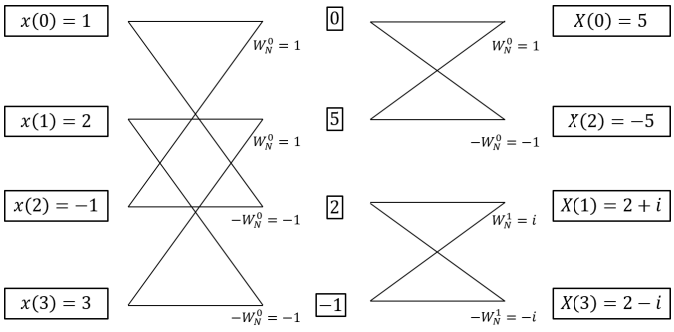


Fig. 2. Radix-2 FFT algorithm with $N = 4$.

TABLE II
DIFFERENT IMPLEMENTATION ALGORITHM FOR CONVOLUTION

Algorithms	Arithmetic Reduction	Transformation Overhead	Inner Computation
Spatial	×	-	Dot product
GEMM	×	△	Dot product
Winograd	√	△△△	EWMM
FFT	√	△△	EWMM

where pad means to pad the filter with zeros to the same size as input. The inverse transform function is

$$\text{Inverse\_Transform}(E) = \text{crop}(\text{IFFT}(E)) \tag{11}$$

where crop represents cropping the intermeidate result to the output size.

3) Comparison: Table II compares the algorithms for convolution which shows that the Winograd and FFT algorithms can effectively reduce the arithmetic complexity with transformation overhead.

Though the Winograd algorithm and FFT algorithm have a similar flow, they differ in the following aspects.

1) *Multiplication Savings:* In fast algorithms, only stage 2 needs multiplications which depend on the input size. For Winograd algorithms, the number of multiplications is n^2 which means one multiplication per input. For example, for a 4×4 output tile generated by convolving a 6×6 input tile with a 3×3 filter, spatial convolution needs $4^2 \times 3^2 = 144$ multiplications, while the Winograd algorithm only needs $6 \times 6 = 36$ multiplications. However, FFT algorithm involves complex numbers. Generally, one complex multiplication needs four real multiplications. In fact, by applying the technique in [38], a complex multiplication only needs three real multipliers as shown in the following equation:

$$\begin{aligned}(a+ib) \times (c+id) &= (ac-bd) + i(ad+bc)\\ (ac-bd) &= b(c-d) + c(a-b)\\ (ad+bc) &= b(c-d) + d(a+b).\end{aligned} \tag{12}$$

Besides, in deep learning scenario, the input is usually real value which makes the transformed matrix of FFT has Hermitian symmetry [39] as shown in the following equation:

$$\overline{X(i,j)} = X(-i \ \textbf{mod}\ n, -j \ \textbf{mod}\ n). \tag{13}$$

In the example of Fig. 2, we can observe that $X(0)$ and $X(2)$, and $X(1)$ and $X(3)$ show Hermitian symmetry. Hermitian symmetry can reduces the number of complex multiplications from $n \times n$ to $n \times (\lfloor (n/2) \rfloor + 1)$. To conclude, the number of multiplications reduce to $3n(\lfloor (n/2) \rfloor + 1) \approx 1.5$ per input pixel in EWMM stage. Table III shows that the FFT algorithm must use tile size at least 16 × 16 to equal the multiplication stage complexity of Winograd 6×6 tile.

2) *Constants Ranging:* The constant multiplications in the transformation stage can be replaced with shift operations which can be implemented using look up tables (LUTs) on FPGAs. Therefore, the range of constants determines the precision requirement. In other words, the larger or smaller constants are, the more LUTs are required. So we compare the constants in the Winograd and FFT algorithms. In Winograd, the constants are determined by polynomial interpolation. In FFT, the constants are twiddle factors W_N^k. In Table III, we list the constants with some commonly used tile size from which we observe that the Winograd algorithm has a higher precision requirement with the same input tile size.

TABLE III
PARAMETERS OF DIFFERENT TILE SIZE IN FAST ALGORITHMS

Winograd algorithm					
n	r	m	max	min	multiplication saving convention/Winograd
4	3	2	1	1/2	$36/16 = 2.25$
5	3	3	4	1/6	$81/25 = 3.24$
6	3	4	8	1/24	$144/36 = 4$
7	3	5	81	1/120	$225/49 = 4.59$
8	3	6	243	1/720	$324/64 = 5.06$
9	3	7	4096	1/5040	$441/81 = 5.44$
6	5	2	5	1/24	$100/36 = 2.78$
7	5	3	16	1/120	$225/49 = 4.59$
8	5	4	49	1/720	$400/64 = 6.25$
9	5	5	256	1/5040	$625/81 = 7.71$
FFT algorithm					
4	3	2	1	0.25	$36/22 = 1.64$
8	3	6	1	0.088	$324/94 = 3.45$
16	3	14	1	0.024	$1764/382 = 4.62$
32	3	30	1	0.0061	$8100/1534 = 5.28$

III. DATAFLOW

Both Winograd-based and FFT-based convolutions employ tiles as their basic units. There are two widely used methods in signal processing to split feature maps into tiles.

1) *Overlap-and-Add (OaA):* In this method, the input is split into several tiles without overlap. Then the output tiles are overlapped by $r-1$ and added together to create the final result, as shown in Fig. 1(a).
2) *Overlap-and-Save (OaS):* In this method, as shown in Fig. 1(b), output tiles which come from the overlapped input tiles are concatenated into resulting convolution.

Though these two methods have the same arithmetical complexity, OaA method can lead to severe memory conflict in FPGA implementation. As shown in Fig. 1(a), the overlapped parts share the same address and multiply-and-accumulate (MAC) operation requires at least two ports in block random access memory (BRAM) when the task is fully pipelined. Therefore, we choose the OaS method as our dataflow to solve data dependency issues.

```
L1:for row = 0 to R, row+=m{
L2:  for col = 0 to C, col+=m{
L3:    for to = 0 to N, ++to{
L4:      for ti = 0 to M, ++ti{
           tile_Z = load(row,col,ti);
           fast_algo(tile_Z, filter, tile_Y);
           output(row,col,to) += tile_Y;
         }
         output[row][col][to] += bias[to];
       }
     }
   }
```

Listing 1. Dataflow of the convolution with OaS.

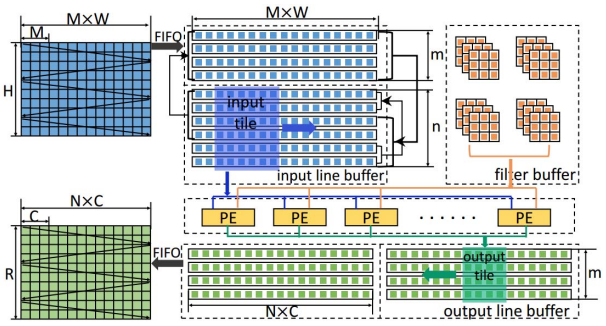


Fig. 3. Architecture overview.

Together with the OaS method, we apply fast algorithms in the generation of the output tiles as shown in Fig. 1(b). The pseudo code of the fast convolution algorithm can be written as that in Listing 1.

IV. Architecture Design

Based on OaS, we propose an FPGA accelerator design for CNNs based on Winograd-based and FFT-based convolutions. However, there exist several challenges. 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 IV-B). Second, different from the spatial convolution algorithm, the fast algorithms generate 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 transformation starts. We design an efficient PE engine for the Winograd algorithm (Section V-A) and FFT algorithm (Section V-B). Then we instantiate multiple PEs through parallelization (Section V-C). Third, different implementation parameters (tile size and 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 VI).

A. Architecture Overview

Fig. 3 presents the architecture overview of convolutional layer based on the fast algorithms 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 Fig. 1. Each line of the line buffers stores the same rows across all the channels. PEs (Winograd PE or FFT PE) fetch data from line buffers. Concretely, given a $n \times n$ input tile, a PE will generate a $m \times m$ 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 and filters) are stored in the external memory initially. The input and output feature maps are transferred to FPGAs via a first-input, first-output (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

On-chip memory is not always sufficient enough to store the entire feature maps and weights. Therefore, many previous works applied loop tiling strategy [3], [6], [13], [28], [37]. Different tiling strategies can lead to different data reuse opportunities. In our design, we choose to tile the loops $L1$–$L3$ in Listing 1 as shown in Fig. 4(a). Because when the filter sliding across feature maps, the relationship between the data of different channels is irrelevant or independent.

There exist data reuse opportunities both horizontally and vertically when tiling loop $L1$. Clearly, two neighboring tiles share $(r-1) \times n$ elements for each input feature map as shown in Fig. 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 Fig. 3. 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 Fig. 1(b). However, different layers may have different feature map width and channels. In practice, we set W as the maximal width of all the feature maps.

To reuse the data, we set the tiling factor of $L1$ to m and store $n+m$ input lines in on-chip memory in total and rotate the lines as a circular buffer. More clearly, initially, fast algorithm 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 the fast algorithm is m. Therefore, fast algorithm 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. Loop Tiling

Generally, the on-chip memory of FPGAs is not large enough to hold the entire feature maps because the number of channels increases exponentially as the network goes deeper. Hence, we tile the feature maps so that it can be accommodated in the line buffers. Specifically, we tile $L3$ and $L4$ loops by the factor Tm and Tn, which means that the accelerator handles Tm channels of the input feature maps and Tn channels of the output feature maps as shown in Fig. 4. We do not

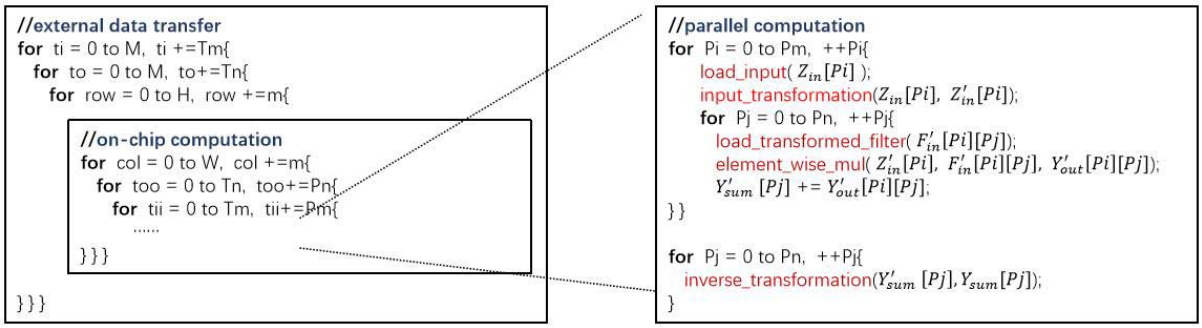


Fig. 4. (a) Loop tiling for limited on-chip memory. (b) Local memory promotion for fast algorithms.

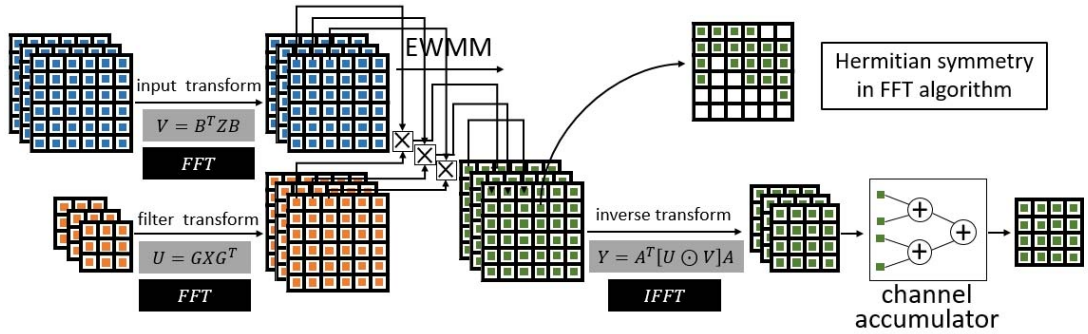


Fig. 5. General PE design.

tile $L1$ and $L2$ loops. This is because it will lead the complex data dependency in the boundary.

V. PE Design

Once input data are ready, we feed them to PEs to perform convolution. Since both Winograd-based and FFT-based convolutions have three stages as described in Section II-B, we propose a general PE design for them.

Fig. 5 illustrates the PE details. Three stages (input and filter transform, EWMM, and output inverse transform) are pipelined so that different tiles can be effectively overlapped. We also use an additional stage to accumulate the output tiles from different input channels. The PE has multiple modules to perform these stages. However, due to the algorithm difference, the PE has to be specialized as below.

A. Winograd PE Design

In Winograd PE design, we choose to transform the filters online. In this way, not only on-chip BRAM resources are saved, but also cause no extra delay because the transformation of input and filter can be done in parallel. Observing that transformation matrices (B, G, and A) can be determined as long as the input tile size and the filter size are given, we replace the multiplications in the transformation stage with constant multiplications which are computed using shift operations. The shift operation can be easily implemented using LUT arrays, therefore the DSP utilization can be reduced. The multiplications in stage 2 are performed in parallel, so we store the intermediate data matrices in registers to improve the memory bandwidth as it alleviates the memory bank conflicts.

B. FFT PE Design

Traditional FFT implementations focus on data with the large size which may contain thousands of signals [40], [41]. In these scenarios, FFT computation cannot be fully paralleled because of resources constraints. However, in CNN computation, the filter size is relatively small and the feature map size also decreases as the networks go deeper. Therefore, it is reasonable to apply a small-scale FFT algorithm.

Based on (10) and (11), we specialize the PE for FFT-based convolution. In the transformation stage, to perform 2-D-FFT, we first conduct 1-D-FFT for each row of the input tile, then the intermediate matrix is transposed, preparing for next row-wise 1-D-FFT. Then another transposition is required to obtain the right permutation of the FFT result. Note that the multiplications in stages 1 and 3 are also constant multiplications, therefore it does not cost any DSPs. In stage 2, we apply the technique in (12) to conduct the complex multiplication. Moreover, we leverage Hermitian symmetry of both input and filter to reduce the number of multiplications and memory requirement, as shown in Fig. 5. Similar to Winograd PE, intermediate matrices are stored in registers and constant multiplications are replaced by shift operations.

C. PE Parallelization and Local Memory Promotion

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 ($L1$–$L4$) surrounding the fast algorithm engine in Listing 1. We choose not to parallelize the $L1$ as it will

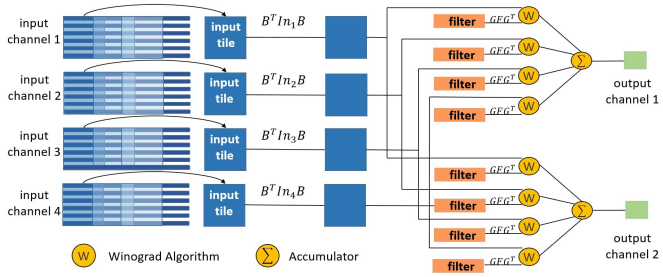


Fig. 6. PE parallelization design with $P_m = 4$ and $P_n = 2$.

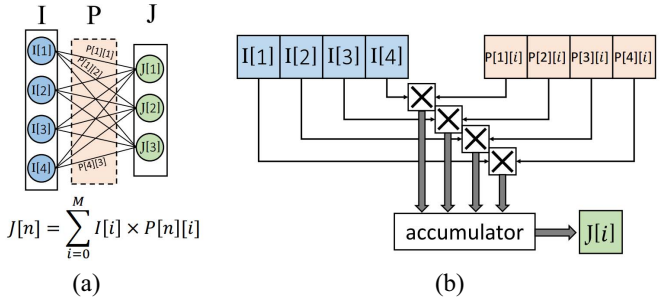


Fig. 7. FC layer implementation. (a) FC layer. (b) Implement FC as EWMM.

TABLE IV
MEMORY PARTITION FACTORS

buffers	Column	Row	Input channels	Out channels
$filter_W$	r	r	P_m	P_n
$filter_F$	n^2		P_m	P_n
input	n	-	P_m	-
output	m	-	-	P_n

significantly increase the size of line buffers. Different parallelization strategies of the other three loops can lead to different data sharing and throughput. Similar to [13], we do not choose to parallelize the $L1$ and $L2$ loops as the parallelization can lead to serious memory bank conflicts. We define the unroll factors of $L4$ and $L3$ are P_m and P_n, respectively. Therefore, there are a total of $P_m \times P_n$ fast algorithm PEs in parallel. We implement the parallelization through loop unrolling as shown in Fig. 4(a). Loops that are fully unrolled are set be the innermost loops. Fig. 6 shows the implementation of parallelization. The input tiles are transformed first then broadcasted to $P_m \times P_n$ PE array, which can reduce the resource consumption.

Together with loop unrolling, we also partition the input, output and filter buffers to sustain efficient memory bandwidth. Clearly, for Winograd, we implement 4-D filters which include dimension row, column, input, and output channels. For FFT-based implementation, we store the filters in multiple buffers resulting from the irregularity after Hermitian symmetry. Each buffer is 3-D which includes dimension column, input, and output channels. In addition, we store the real part and the imaginary part separately. Like Winograd implementation, the row and column dimension are partitioned completely. Therefore, there are $2 \times n \times (\lfloor (n/2) \rfloor + 1) \approx n^2$ buffers for the filters. We implement 2-D input and output buffers and partition each dimension. Table IV gives the partition factors for various buffers.

In parallel computation, we interchange the order of loops as shown in Fig. 4(b) to avoid data duplication. Note that $P_m \times P_n$ PEs only require P_m times of input transformation, therefore we set P_m as the outmost loop in computation. Similarly, the inverse transformation only needs to be performed P_n times. So we do not perform inverse transformation immediately. The results of EWMM operation are accumulated across all input channels, after that, the accumulated results are transformed to the spatial domain.

D. 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 Fig. 7(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. The weights in FC layers are significantly larger than the input neurons. Therefore, we load the entire input neurons of FC layer into on-chip memory but stream the weights using the FIFO interface. In addition, the FC computation contains no data reuse opportunities. To improve 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 set any input value less than zero to zero. ReLU and pooling are implemented by introducing comparison operators to the output buffers.

VI. RESOURCE AND PERFORMANCE MODEL

Our fast algorithm implementations involve several design parameters: input tile size (n) and parallelization degrees (P_m and P_n). Given an input tile size n, since the filter size is fixed for a convolutional layer (e.g., 3×3 and 5×5), the output tile size m can be determined ($m = n - r + 1$). These design parameters affect both performance and accuracy. Hence, we develop an analytical model that can predict the performance of Winograd and FFT algorithm on FPGAs. Then, we employ 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 also increase as n increases, leading to precision loss. In this paper, we use fixed-point 16 bits to represent both data and filter. We set the precision to 2^{-15} for filters to maintain a high accuracy as prior work [3]. Under this precision constraint, we set the maximum value for n as 8 in the Winograd algorithm so that we can precisely represent the constants in the transformation matrix. For FFT algorithm, n is restricted to 4 and 8.

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.

1) Modeling Resource Usage: As mentioned in Section II-B3, only the EWMM operations consume DSP resource. Therefore, the number of DSPs can be formulated

as follows:

$$\text{DSP} = \begin{cases} n^2 \times P_n \times P_m & \text{if Winograd} \\ 3n\left(\lfloor \frac{n}{2} \rfloor + 1\right) \times P_n \times P_m & \text{if FFT.} \end{cases}$$

Modeling LUT resource is more complex. According to Fig. 6, we approximate its consumption using linear regression models

$$\text{LUT} = \alpha_n^r \times P_m + \beta_n^r \times P_n \tag{14}$$

where α_n^r is the LUT consumption for the transformation of a $n \times n$ input tile in Winograd PE or FFT PE with the filter size r. Note that the results of the EWMM operation are first accumulated then inversely transformed. Therefore, there are nine inverse transformations in total. Here, β_n^r is the LUT consumption for the inverse transformation of a $n \times n$ intermediate tile.

α_n^r and β_n^r can be obtained on different platforms in advance. In our design, α_n^r and β_n^r is obtained from Vivado HLS tool.

The BRAM usage is composed of the banks for filter, input, and output buffers. The bank number is decided by the memory partition factors in Table IV

$$\text{Banks} = \begin{cases} r^2 P_m P_n + (n+m) n P_m + 2m^2 P_n \\ \quad \text{if Winograd} \\ n^2 P_m P_n + (n+m) n P_m + 2m^2 P_n \\ \quad \text{if FFT.} \end{cases}$$

2) Modeling Performance: For both algorithms, their performance is bounded by either computation or data transfer. To efficiently utilize the resource, the data transfer speed must be greater than or equal to the computation speed. According to Fig. 4, we model the time to process n rows of input data in the line buffers as follows:

$$T_{\text{compute}} = \left(\left\lceil \frac{C}{m} \right\rceil \times \left\lceil \frac{T_m}{P_m} \right\rceil \times \left\lceil \frac{T_n}{P_n} \right\rceil \times \text{II} + P_{\text{depth}} \right) \times \frac{1}{\text{Freq}} \tag{15}$$

where Freq is the operating frequency of the FPGAs. II denotes the iteration interval of the pipeline. In our implementation, loops in Fig. 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.

On the other hand, the transfer time for the corresponding input and output data is as follows:

$$T_{\text{transfer}} = \frac{m \times W \times \max(T_n, T_m) \times 16}{\text{Bandwidth}}. \tag{16}$$

Since we require that $T_{\text{transfer}} \leq T_{\text{compute}}$, bandwidth requirement can be formulated as follows:

$$\text{Bandwidth} \geq m^2 \times \left\lceil \frac{P_m \times P_n}{\min(T_n, T_m)} \right\rceil \times 16 \times \text{Freq}. \tag{17}$$

In addition, we also consider the time T_{init} to load the first n rows of the input image into on-chip memory and filters as follows:

$$T_{\text{init}} = \frac{T_m \times T_n \times r \times r + n \times W \times T_m}{\text{Bandwidth}/16}. \tag{18}$$

Putting it all together, the total operations and processing time of the convolution are

$$\text{OPs} = H \times W \times M \times N \times r^2 \times 2 \tag{19}$$

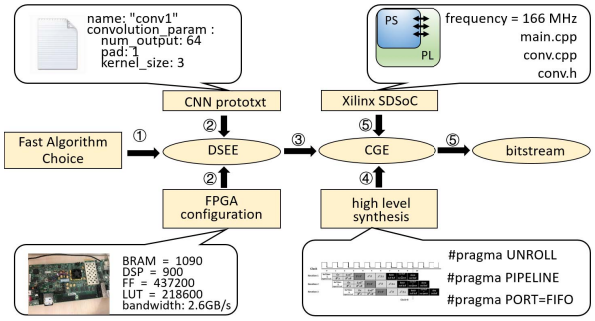


Fig. 8. Automatic tool chain.

$$T_{\text{total}} = \left\lceil \frac{M}{T_m} \right\rceil \times \left\lceil \frac{N}{T_n} \right\rceil \times \left(\left\lceil \frac{H}{m} \right\rceil \times T_{\text{compute}} + T_{\text{init}} \right). \tag{20}$$

Accordingly, the effective performance of convolution based on fast algorithm is as follows:

$$\text{Perf}_{\text{eff}} = \frac{\text{OPs}}{T_{\text{total}}}. \tag{21}$$

Now, given a convolutional layer represented by $\{H, W, M, R, C, N, r\}$ and tiling factors $\{T_n, T_m\}$ determined by the size of on-chip memory, our goal is to find the optimal solution $\{n, P_m, P_n\}$ which maximizes the performance (21) 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.

VII. Automatic Tool Chain

We propose an automatic tool chain that maps CNN model to FPGA implementation. The observation that OaS method is suitable for both Winograd algorithm and FFT algorithms motivates a uniform data locality-aware architecture design. Clearly, both Winograd PE and FFT PE fetch data from line buffer and generate output tile by tile. So shifting one algorithm to another only requires reconfiguration of the PE without any other change to the architecture. To optimize our design, we formulate the performance and resource utilization with design parameters. Then we rely on this model to guide our design space exploration. With a given fast algorithm (Winograd or FFT) our automatic tool can map CNNs onto FPGAs automatically as shown in Fig. 8. The flow consists of four steps. In step 1, CNN architecture and FPGA configuration are fed into the design space exploration engine (DSEE) to get the optimal solution. In step 2, based on the optimal solution, we develop a code generate engine (CGE) which can generate fast convolution functions automatically. In step 3, we use the Xilinx HLS tool to synthesize the code into register transfer level. Finally, we use the Xilinx SDSoC (software-defined system-on-chip) tool-chain to generate the bitstream.

A. Design Space Exploration Engine

First, given a certain fast algorithm, a CNN model and FPGA configuration are fed into DSEE. The CNN model can be described like Caffe prototxt [42]. The FPGA configuration parameters include the memory bandwidth, number of DSPs, logic cells, and on-chip memory capacity. Then, the DSEE will output the optimal solution based on our resource and

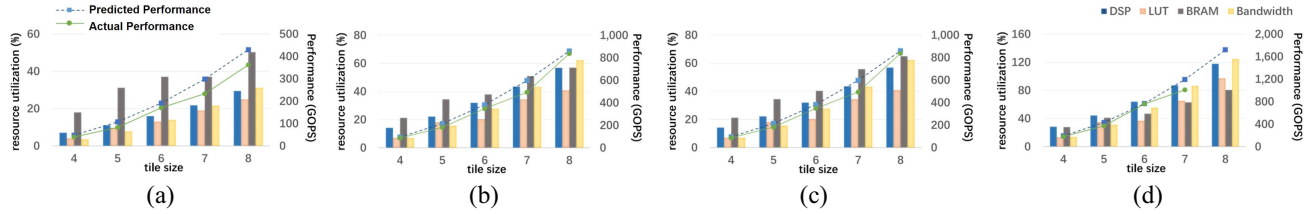


Fig. 9. Resource utilization and performance results in Winograd for 3×3 filter. (a) $P_n = 2, P_m = 2$. (b) $P_n = 2, P_m = 4$. (c) $P_n = 4, P_m = 2$. (d) $P_n = 4, P_m = 4$.

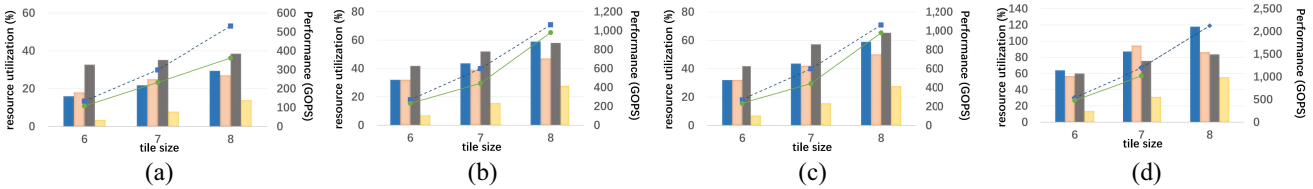


Fig. 10. Resource utilization and performance results in Winograd for 5×5 filter. (a) $P_n = 2, P_m = 2$. (b) $P_n = 2, P_m = 4$. (c) $P_n = 4, P_m = 2$. (d) $P_n = 4, P_m = 4$.

performance model in Section VI. The solution includes the workload of a single PE n and the number of PEs P_n and P_m.

B. Code Generate Engine

Based on the optimal solution, CGE can generate the fast convolution functions using C++ template. First, the algorithm parameters (the tile size in fast algorithms) determine the memory-aware functions. Then, according to the certain algorithm and parameters, CGE will generate corresponding PE functions. These functions describe the whole accelerator architecture, including line buffers, buffer management, and PE configurations. Precisely, the generated implementation is HLS compatible C++ code. Pragmas, which are used to describe hardware structure, are inserted into the functions. For example, optimal parallelism parameters P_n and P_m from DSEE set the loop unrolling factors and memory partition factors.

VIII. Experimental Evaluation

A. Experiments Setup

We evaluate our techniques on two FPGA platforms: 1) Xilinx ZC706 and 2) 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 166-MHz frequency on this platform. Xilinx ZCU102 consists of an UltraScale FPGA, quad ARM Cortex-A53 processors, 500-MB DDR3. Our FPGA implementations are operated at 200-MHz 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 (Sections VIII-B and VIII-C). Then, we perform case studies using the state-of-the-art CNNs, including AlexNet, VGG16, Resnet, and YOLO (Section VIII-E). 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 (21). For the spatial algorithm, the effective performance is always bounded by Max$F$, the maximum computational capability of the FPGA platform. Max$F$ = DSP $\times$ Freq $\times$ 2, where 2 means multiply and add operations. However, for the fast algorithm, the effective performance can exceed the Max$F$ as fast algorithm can increase the effective DSP efficiency by reducing the number of multiplications required by convolution.

B. Winograd Model and Performance Analysis

In this section, 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: $224(H) \times 224(W)$ with $\{M = N = T_n = T_m = 64\}$ and try two different filter sizes: 3×3 and 5×5. Figs. 9 and 10 compare the predicted 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 Figs. 9 and 10. Instead, BRAMs and memory bandwidth can be the limiting resources. The BRAMs consumption comes from a few aspects. First, unlike the spatial 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, the Winograd algorithm saves the DSPs and improves the overall resource utilization.

C. FFT Model and Performance Analysis

Fig. 11 shows the experimental results of the FFT model. As aforementioned, in the FFT algorithm, the FFT size is equal to the input tile size and the filters are padded to the same size as the input tile. In this section, we only evaluate FFT with two sizes ($n = 4$ and $n = 8$) in a single convolutional layer just. Because when FFT size is larger than 8, the on-chip memory is not large enough to store all buffers in our framework. On average, the prediction error of our performance model is 10.1%.

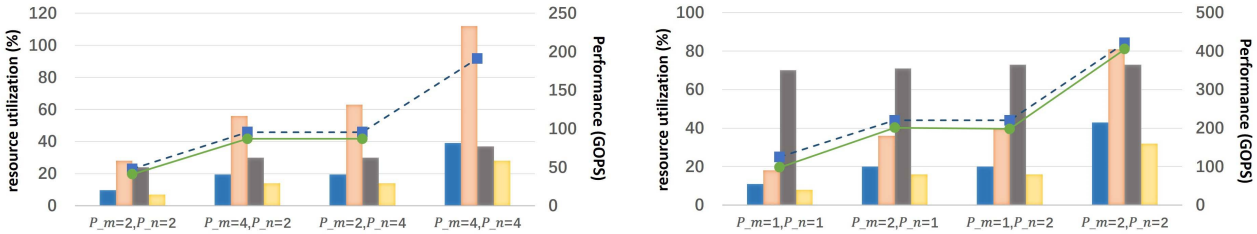


Fig. 11. Resource utilization and performance results in FFT. (a) FFT size = 4. (b) FFT size = 8.

TABLE V
PARAMETERS IN RESOURCE MODEL

(n,r)	(4,3)	(5,3)	(6,3)	(7,3)	(8,3)	(6,5)	(7,5)	(8,5)
Winograd								
α_n^r	985	2137	3006	5779	8375	7789	10021	13121
β_n^r	1231	4067	5317	13245	20213	14211	22150	25079
FFT								
α_n^r	20600	-	-	-	41280	-	-	-
β_n^r	21130	-	-	-	42340	-	-	-

TABLE VI
RESOURCE UTILIZATION OF WINOGRAD AND FFT

Resource utilization of Winograd				
		BRAM	LUT	DSP
$P_m = 4, P_n = 4$ $n = 6, r = 3$	predicted	512(47%)	118044(54%)	576(64%)
	real	532(49%)	126788(58%)	643(71%)
$P_m = 2, P_n = 2$ $n = 8, r = 5$	predicted	392(36%)	76400(35%)	256(28%)
	real	425(39%)	808821(37%)	297(33%)
Resource utilization of FFT				
		BRAM	LUT	DSP
$P_m = 4, P_n = 4$ $n = 4$	predicted	391(36%)	166920(76%)	320(35%)
	real	414(38%)	238246(109%)	351(39%)
$P_m = 2, P_n = 2$ $n = 8$	predicted	368(41%)	167240(76%)	576(64%)
	real	397(44%)	179252(82%)	643(71%)

TABLE VII
DESIGN PARAMETERS

Winograd design parameters				
ZC706 (ZCU102)	n	P_n	P_m	N_{batch}
Alexnet(3×3)	6(6)	2(4)	8(8)	16(64)
VGG16(3×3)	6(6)	4(8)	4(8)	8(32)
Resnet(3×3)	6	4	4	8
FFT design parameters				
ZC706	n	P_n	P_m	N_{batch}
VGG16(3×3)	8	2	2	8

In Fig. 11, we can see that DSP becomes the abundant resources because of the improvement of computation efficiency. Compared to Winograd-based implementation, FFT-based implementation requires more BRAMs, since the filters are complex numbers which have transformed offline. Moreover, the FFT algorithm consumes more logic resources (LUT). Precisely, there are more additions and constant multiplications in the FFT algorithm comparing the Winograd algorithm. Note that 2-D-FFT transformation takes $2n$ times more operations than 1-D-FFT transformation. However, in Winograd (6) and (7), the transformation of the input tile only requires two constant matrix multiplications.

D. Resource Validation

In previous FPGA implementation which applied the spatial algorithm, the performance is usually bounded by the number DSPs. However, when applying fast algorithms which can result in less DSP utilization, the other resources like LUT and BRAM should be taken into account. In this section, we evaluate our resource model in Section VI. Table V lists the parameters α_n^r and β_n^r in (14) which comes from Xilinx Vivado HLS tool to guide our resource exploration. For the FFT algorithm, the filters are padded to the same size as the input tile and transformed offline, parameters for the filter with $r = 5$ are not necessary. For both algorithms, α_n^r and β_n^r increase dramatically with n, because the matrix size and constants increase exponentially. For FFT algorithm, α_n^r and β_n^r are approximately equal, because FFT and IFFT share the same arithmetic complexity.

Figs. 9–11 show the resource utilization obtained from the Xilinx Vivado tool. It shows that our resource model can accurately predict resource utilization. Table VI shows the real resource utilization and predicted utilization for four cases in Figs. 9–11. It is reasonable that the real resource utilization is slightly higher than the predicted. The extra BRAMs are used for FIFO logic and buffers in padding process. And a small part of LUT is used for multiplexer and FIFO logic. Besides the multiplications in EWMM stage, the calculation for the boundary condition and array index also need DSPs to perform multiplications. Besides, we find that the real LUT utilization is higher than the predicted utilization in the FFT algorithm. This is because the transposition operation in 2-D-FFT algorithm needs the logic resource to rearrange the data layout.

E. Case Study

Here, we evaluate our Winograd implementation using AlexNet, VGGNet, and Resnet. FFT is tested on VGGNet and YOLO. Table VII gives the parameters for each network in our implementation. For ZC706 platform, we choose tiling parameter as $T_m = T_n = 64$ and for ZCU102 platform, $T_m = T_n = 128$.

1) AlexNet: AlexNet consists of five convolution and three FC layers [10]. 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 spatial 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. Due to the layer diversity of AlexNet, we set (2/3) on-chip resource as our constraints in the design spaces exploration and the rest resources are implemented for the spatial convolution algorithm.

TABLE VIII
PERFORMANCE COMPARISON FOR ALEXNET

	[13]	[43]	Our Impl	Our Impl
Precision	32bits float	16bits fixed	16bits fixed	16bits fixed
Device	VX485T	Arria 10	ZC706	ZCU102
Freq (MHz)	100	303	167	200
Logic cell (K)	485.7	427	350	600
DSP[2]	2800	1518	900	2520
BRAM (Kb)	2060×18	2713×20	1090×18	1824×18
Algorithm	Convention	Winograd	Winograd	Winograd
conv1 (GOP/s)	27.5	2308	83.1	309.6
conv2 (GOP/s)	83.8	1740	501.7	1355.6
conv3 (GOP/s)	78.8	1960	548.9	1535.7
conv4 (GOP/s)	77.9	1960	360.2	1361.7
conv5 (GOP/s)	77.6	1743	360.2	1285.7
conv average (GOP/s)	61.6	-	265.1	1006.4
CNN average (GOP/s)	-	1382	202.8	854.6
Power (W)	18.6	45	9.4	23.6
DSP Efficiency (GOP/s/DSPs)	0.022	0.91	0.223	0.339
Logic cell Efficiency (GOP/s/cells/K	0.127	3.24	0.574	1.424
Energy Efficiency (GOP/s/W)	3.31	30.7	21.4	36.2

TABLE IX
PERFORMANCE COMPARISON FOR VGG

	[7]	[44]	Our Impl	Our Impl
Precision	16bits fixed	32bits float	16bits fixed	16bits fixed
Device	Arria 10	Intel HARP	ZC706	ZCU102
Freq (MHz)	370	200	166	200
Logic cell (K)	427	-	350K	600K
DSP	1518	224	900	2520
BRAM (Kb)	2713×20	32768	1090×18	1824×18
Algorithm	Convention	FFT	FFT	Winograd
conv1 (GOP/s)	1098.0	118.2	241.5	1908.2
conv2 (GOP/s)	1232.0	119.4	389.1	3312.4
conv3 (GOP/s)	1329.6	111.8	342.7	3111.1
conv4 (GOP/s)	1347.9	111.6	289.7	2527.3
conv5 (GOP/s)	1223.3	125.5	197.1	2021.1
conv average (GOP/s)	1283.9	123.5	296.8	2601.3
CNN average (GOP/s)	866.0	-	277.8	2479.6
Power (W)	41.7	13.2	9.4	23.6
DSP Efficiency (GOP/s/DSPs)	0.57	0.54	0.31	0.98
Logic cell Efficiency (GOP/s/cells)	2.03	-	0.79	4.13
Energy Efficiency (GOP/s/W)	20.8	9.3	29.4	105.4

Table VIII gives the results. Zhang *et al.* [13] only given the convolution implementation without FC layers. Compared to prior work [13], we improve the average convolution performance from 61.6 to 1006.4 GOP/s.[1] For the overall CNN, we improve the performance from 72.4 to 854.6 GOP/s compared to [2]. Our performance is less than [43], this mainly comes from three reasons.

1) The irregular structure of AlexNet. The stride of the first layer is 4, which makes the first layer inefficient if using Winograd.
2) The frequency in [43] is much higher.
3) The DSP in Arria 10 can be implemented as two FP16 multipliers.

However, for AlexNet, our implementation shows a better energy-efficiency. When implementing a more regular CNN model (like VGGNet), our design shows a better performance result.

2) VGGNet: VGG16 [34] consists of five convolution groups with different input size (224, 112, 56, 28, and 14). In VGG16, all convolutional layers are with 3×3 filters, which fit well for Winograd algorithm. Therefore, we set 95% on-chip resource as our constraints in the design spaces exploration. Table IX compares our techniques with prior works. For the convolutional layers, we improve the average performance from 1283 to 2601.4 GOP/s compared to [2], [3], and [7]. For the overall CNN, we improve the performance from 866 (1790 if fixed point) [7] to 2479.6 GOP/s. Zhang and Li [7] did not apply fast algorithm but showed a comparable performance with ours. Because the frequency in [7] is much higher than ours and the different configuration of DSPs between Intel FPGA and Xilinx FPGA. To make a fair comparison across different platforms, we also present the total resource efficiency and energy efficiency on each platform. We can observe that our implementation achieves better resource efficiency, which comes from the reduction of arithmetic complexity and novel architecture.

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.

We also apply the FFT fast algorithm to accelerate VGGNet on the ZC706 platform and compare it with [44] which also apply FFT algorithm. Zhang and Prasanna [44] did not implement FC layer, so the resource and energy efficiency is computed according to the convolution result. We improve the performance from 123.5 to 277.8 GOP/s and energy-efficiency from 9.3 to 29.4.

3) YOLO: You only look once (YOLO) is a state-of-the-art network for real-time object detection system [35]. We use Tiny-YOLO version to evaluate our design. Tiny-YOLO consists of nine convolutional layers and six max pooling layers. All convolutional layers use 3×3 filters. We apply the FFT algorithm to accelerate YOLO network, the results are shown in Table X. The detailed performance of each layer is shown in Fig. 12. We notice that the performance increases in the first few layers. This is because the number of channels in the first few layers is relatively small which means less data reuse opportunities, so the performance of these layers is bounded by the off-chip bandwidth.

4) Resnet: Resnet is a modern network for image recognition which consists of many residual blocks [11]. A residual block is composed of two 1×1 convolutional layers and one 3×3 convolutional layer. We set (2/3) on-chip resource as

[1]In [13], FC layer is not implemented. So the efficiency value of [13] 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 X
PERFORMANCE COMPARISON FOR RESNET AND YOLO

	[46]	[45]	Our Impl	Our Impl
CNN type	Resnet	Resnet	Resnet	YOLO
Precision	16bits fixed	16bits fixed	16bits fixed	16bits fixed
Device	Arria 10	Stratix-V GSMD5	ZC706	ZC706
Freq (MHz)	200	150	166	166
Logic cell (K)	427	172	350	350
DSP	1518	1590	900	900
BRAM (Kb)	2713×20	2014×20	1090×18	1090×18
Algorithm	Convention	GEMM	Winograd	FFT
conv average (GOP/s)	935.3	-	163.1	201.1
CNN average (GOP/s)	707.2	226.5	130.4	201.1
Power (W)	-	25	9.4	9.4
DSP Efficiency (GOP/s/DSPs)	0.47	0.14	0.15	0.23
Logic cell Efficiency (GOP/s/cells)	1.65	1.31	0.37	0.57
Energy Efficiency (GOP/s/W)	-	9.0	13.8	21.4

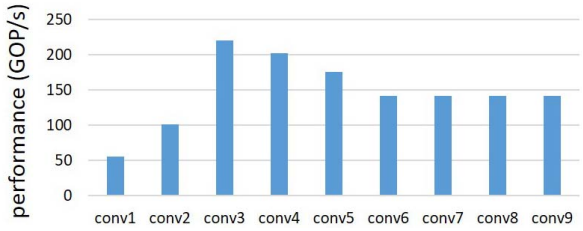


Fig. 12. Performance of each layer in YOLO.

our constraints for the fast algorithm. We apply the Winograd algorithm to accelerate the 3×3 convolutional layer and the spatial convolution algorithm for 1×1 convolutional layer. Table X gives the results. Our performance is 163.1 GOP/s for the convolutional layers and 130.4 GOP/s for the whole network. Our performance is less than [45] and [46], this mainly comes from two aspects.

1) A single DSP can be implemented as two 16×16 fixed-point multipliers in Intel Altera platform.
2) Part to on-chip resources is used for 1×1 filters and these filters account for 40% computation in a residual block.

For Resnet, our implementation shows a better energy-efficiency compared with [45].

F. Comparison With GPU

In this section, we conduct a comparison between GPU and FPGA platforms. For GPUs, we measure the performance of VGG16 using Caffe framework [42] on NVIDIA TitanX platform. To make a fair comparison, we test the performance of TitanX with the latest cuDNN [47] as Winograd and FFT algorithm is also included in cuDNN Power on GPU is obtained using NVIDIA profiling tools. Table XI shows the comparison results. In cuDNN, Winograd algorithm outperforms FFT algorithm, because A significant memory workspace is needed to store intermediate results in FFT algorithm which takes more time than Winograd. As shown, TitanX gives better performance, but our implementation on Xilinx ZCU102 FPGA achieves much better (2.5X) energy efficiency.

TABLE XI
COMPARISON WITH GPU PLATFORM

Device	TitanX[1]	TitanX[2]	TitanX[3]	ZCU102
Technology	28 nm	28 nm	28 nm	16 nm
Precision	32bits float	32bits float	32bits float	16bits fixed
CNN average (TOP/s)	4.98	5.60	1.98	2.48
Power (W)	130	134	118	23.6
Energy efficiency (GOP/s/W)	38.3	41.8	16.8	105.4

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

IX. RELATED WORK

A. Architecture for CNNs Using Spatial Convolution

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 spatial algorithms [1], [2], [5]–[7], [12], [13], [45]. In [2], 3-D convolution operations is flattened as 2-D GEMM, which is widely adopted on GPU platforms. But on FPGAs, it can result in massive memory usage. Song *et al.* [18] proposed a general purpose accelerator using the kernel-partition method. Ma *et al.* [6] made an in-depth analysis of loop optimization techniques in spatial convolution, which includes loop tiling, loop unrolling, and loop interchange. Zhang and Li [7] focused on reducing the on-chip memory bandwidth requirement. Wang *et al.* [48] proposed to employ a structured compression technique using block-circulant matrices to compress the LSTM model small enough to be fitted on BRAMs of FPGA. Zeng *et al.* [49] presented a framework for generating Verilog of high throughput CNN accelerators. Zeng *et al.* [49] proposed a novel concatenate-and-pad technique, which improves OaA significantly by reducing the wasted computation on the padded pixels. Wei *et al.* [50] implemented CNN on an FPGA using a systolic array architecture, which can achieve high clock frequency under high resource utilization. Wei *et al.* [31] proposed tile-grained pipeline architecture for low latency CNN inference, which supports pipelining execution of multiple tiles within a single input image on multiple heterogeneous accelerators. Lin *et al.* [51] proposed layer clusters paralleling mapping method to classify the layers into clusters based on their differences of parameters and data localities, and different clusters will be accelerated using different partitions of FPGA. Zhang *et al.* [52] proposed an AccDNN tool which included high-quality RTL network layer IPs, a fine-grained layer-based pipeline architecture, and an automatic design space exploration tool.

B. Architecture for CNNs Using Fast Algorithms

A few studies also focus on reducing the arithmetic complexity of convolution [14], [19], [43], [44] using non-conventional algorithms. Zhang *et al.* [14] reduced the computation using low-rank approximation which is based on minimizing the reconstruction error of nonlinear response. Lavin and Gray [19] evaluated the FFT and Winograd algorithms on GPU platforms. In general, FFT shows less

efficiency for convolutions with small filters. Zhang and Prasanna [44] implemented FFT with the size of 8 on FPGA platform for CNN. But it shows a little reduction of computation complexity for two reasons. First, the multiplications in FFT is constant multiplications which do not have to be implemented by DSPs. Second, Zhang and Prasanna [44] did not take advantage of Hermitian symmetry which can reduce the DSP consumption by half. Ko *et al.* [53] proposed an FFT-based architecture for CNN model, which can be used for training and inference process. In the design in [53], Hermitian symmetry and three-DSP-multiplications were applied. Aydonat *et al.* [43] applied the Winograd algorithm on the Arria 10 FPGA platform. But they only use 1-D Winograd to reduce arithmetic complexity in which the multiplication saving rate is only (n/mr). Shen *et al.* [54] proposed a uniform template-based architecture based on the Winograd algorithm. Shen *et al.* [54] extended the Winograd algorithm to 3-D for 3-D CNN models. Podili *et al.* [55] also implemented the Winograd algorithm and proposed a novel data layout to reduce the required memory bandwidth by half. To achieve generality and higher resource utilization, Xiao *et al.* [28] explored heterogeneous algorithms to maximize the throughput of a CNN based on a fusion architecture. The work in [29] is called SpWA accelerator which exploits sparsity in CNN models. Lu and Liang [29] rearranged the data access in the convolutional layer and transformed 2-D-Winograd algorithm to several vector-matrix multiplications.

In this paper, we deeply evaluate 2-D Winograd algorithm and FFT on FPGA platforms. We propose a uniform framework which uses a line-buffer structure to enable data reuses and performance models to guide design space exploration.

X. Conclusion

FPGAs have been widely used to accelerate CNN-based applications. However, prior implementations based on the spatial algorithms are mainly limited by the computational capability of FPGAs. In this paper, we propose a framework on FPGAs based on the fast algorithm, which can effectively reduce the number of multiplications in the convolutional layer. We design an efficient PE engine for both the Winograd algorithm and FFT algorithm. To guide a fast design space exploration, we also develop analytical models to estimate resource usage and performance. Our implementations can reach the peak performance 2479.6 GOP/s on ZCU102 FPGA platform, which outperforms all previous work.

References

[1] C. Zhang *et al.*, "Energy-efficient CNN implementation on a deeply pipelined FPGA cluster," in *Proc. Int. Symp. Low Power Electron. Design*, 2016, pp. 326–331.

[2] N. Suda *et al.*, "Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks," in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays*, 2016, pp. 16–25.

[3] J. Qiu *et al.*, "Going deeper with embedded FPGA platform for convolutional neural network," in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays*, 2016, pp. 26–35.

[4] Y. Ma, N. Suda, Y. Cao, J. S. Seo, and S. Vrudhula, "Scalable and modularized RTL compilation of convolutional neural networks onto FPGA," in *Proc. Int. Conf. Field Program. Logic Appl.*, 2016, pp. 1–8.

[5] H. Li *et al.*, "A high performance FPGA-based accelerator for large-scale convolutional neural networks," in *Proc. Int. Conf. Field Program. Logic Appl.*, 2016, pp. 1–9.

[6] Y. Ma, Y. Cao, S. Vrudhula, and J.-S. Seo, "Optimizing loop operation and dataflow in FPGA acceleration of deep convolutional neural networks," in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays*, 2017, pp. 45–54.

[7] J. Zhang and J. Li, "Improving the performance of OpenCL-based FPGA accelerator for convolutional neural network," in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays*, 2017, pp. 25–34.

[8] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification," in *Proc. IEEE Int. Conf. Comput. Vis.*, 2015, pp. 1026–1034.

[9] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2014, pp. 580–587.

[10] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2012, pp. 1097–1105.

[11] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.

[12] C. Szegedy *et al.*, "Going deeper with convolutions," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2015, pp. 1–9.

[13] C. Zhang *et al.*, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays*, 2015, pp. 161–170.

[14] X. Zhang, J. Zou, X. Ming, K. He, and J. Sun, "Efficient and accurate approximations of nonlinear convolutional networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2015, pp. 1984–1992.

[15] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *Proc. ACM SIGARCH Comput. Archit. News*, vol. 44, no. 3, pp. 367–379, 2016.

[16] S. Liu *et al.*, "Cambricon: An instruction set architecture for neural networks," in *Proc. ACM SIGARCH Comput. Archit. News*, vol. 44, no. 3, pp. 393–405, 2016.

[17] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, "A dynamically configurable coprocessor for convolutional neural networks," in *Proc. ACM SIGARCH Comput. Archit. News*, 2010, pp. 247–257.

[18] L. Song *et al.*, "C-brain: A deep learning accelerator that tames the diversity of CNNs through adaptive data-level parallelization," in *Proc. IEEE Design Autom. Conf.*, 2016, pp. 1–6.

[19] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 4013–4021.

[20] T. Chen *et al.*, "DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proc. ACM SIGPLAN Notices*, 2014, pp. 269–284.

[21] S. Yin *et al.*, "A high energy efficient reconfigurable hybrid neural network processor for deep learning applications," *IEEE J. Solid-State Circuits*, vol. 53, no. 4, pp. 968–982, Apr. 2018.

[22] F. Tu *et al.*, "Deep convolutional neural network architecture with reconfigurable computation patterns," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 25, no. 8, pp. 2220–2233, Aug. 2017.

[23] S. Yin *et al.*, "A high throughput acceleration for hybrid neural networks with efficient resource management on FPGA," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, to be published.

[24] S. Liang, S. Yin, L. Liu, W. Luk, and S. Wei, "FP-BNN: Binarized neural network on FPGA," *Neurocomputing*, vol. 275, pp. 1072–1086, Jan. 2018.

[25] Y. Liang *et al.*, "High-level synthesis: Productivity, performance, and software constraints," *J. Elect. Comput. Eng.*, vol. 2012, p. 1, Oct. 2012.

[26] J. Cong *et al.*, "High-level synthesis for FPGAs: From prototyping to deployment," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 30, no. 4, pp. 473–491, Apr. 2011.

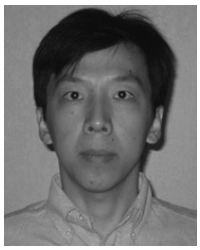
[27] A. Canis *et al.*, "LegUp: High-level synthesis for FPGA-based processor/accelerator systems," in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays*, 2011, pp. 33–36.

[28] Q. Xiao, Y. Liang, L. Lu, S. Yan, and Y.-W. Tai, "Exploring heterogeneous algorithms for accelerating deep convolutional neural networks on FPGAs," in *Proc. Design Autom. Conf.*, 2017, p. 62.

[29] L. Lu and Y. Liang, "SpWA: An efficient sparse winograd convolutional neural networks accelerator on FPGAs," in *Proc. Design Autom. Conf.*, 2018, p. 135.

[30] D. Caiwen *et al.*, "REQ-YOLO: A resource-aware, efficient quantization framework for object detection on FPGAs," in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays*, 2019, pp. 31–40.

[31] X. Wei *et al.*, "TGPA: Tile-grained pipeline architecture for low latency CNN inference," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design*, 2018, pp. 1–8.
[32] G. Zhong, A. Prakash, Y. Liang, T. Mitra, and S. Niar, "Lin-analyzer: A high-level performance analysis tool for FPGA-based accelerators," in *Proc. ACM Design Autom. Conf.*, 2016, p. 136.
[33] Y. Liang, S. Wang, and W. Zhang, "FlexCL: A model of performance and power for OpenCL workloads on FPGAs," *IEEE Trans. Comput.*, vol. 67, no. 12, pp. 1750–1764, Dec. 2018.
[34] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.
[35] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 779–788.
[36] S. Winograd, *Arithmetic Complexity of Computations*, vol. 33. Philadelphia, PA, USA: SIAM, 1980.
[37] L. Lu, Q. Xiao, S. Yan, and Y. Liang, "Evaluating fast algorithms for convolutional neural networks on FPGAs," in *Proc. IEEE Int. Symp. Field Program. Custom Comput. Mach.*, 2017, pp. 101–108.
[38] M. Mathieu, M. Henaff, and Y. Lecun, "Fast training of convolutional networks through FFTs," *arXiv preprint arXiv:1312.5851*, 2013.
[39] M. Hemnani, S. Palekar, P. Dixit, and P. Joshi, "Hardware optimization of complex multiplication scheme for DSP application," in *Proc. Int. Conf. Comput. Commun. Control*, 2015, pp. 1–4.
[40] R. Chen, H. Le, and V. K. Prasanna, "Energy efficient parameterized FFT architecture," in *Proc. Int. Conf. Field Program. Logic Appl.*, 2013, pp. 1–7.
[41] R. Chen, N. Park, and V. K. Prasanna, "High throughput energy efficient parallel FFT architecture on FPGAs," in *Proc. High Perform. Extreme Comput. Conf.*, 2013, pp. 1–6.
[42] Y. Jia *et al.*, "Caffe: Convolutional architecture for fast feature embedding," in *Proc. ACM Int. Conf. Multimedia*, 2014, pp. 675–678.
[43] U. Aydonat, S. O'Connell, D. Capalija, A. C. Ling, and G. R. Chiu, "An OpenCL^{TM} deep learning accelerator on Arria 10," in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays*, 2017, pp. 55–64.
[44] C. Zhang and V. Prasanna, "Frequency domain acceleration of convolutional neural networks on CPU-FPGA shared memory system," in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays*, 2017, pp. 35–44.
[45] Y. Guan *et al.*, "FP-DNN: An automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates," in *Proc. IEEE Int. Symp. Field Program. Custom Comput. Mach.*, 2017, pp. 152–159.
[46] Y. Ma, Y. Cao, S. Vrudhula, and J.-S. Seo, "Optimizing the convolution operation to accelerate deep neural networks on FPGA," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 26, no. 7, pp. 1354–1367, Jul. 2018.
[47] NVIDIA. (2018). *CUDNN*. [Online]. Available: https://developer.nvidia.com/cudnn
[48] S. Wang *et al.*, "C-LSTM: Enabling efficient LSTM using structured compression techniques on FPGAs," in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays*, 2018, pp. 11–20.
[49] H. Zeng, R. Chen, C. Zhang, and V. Prasanna, "A framework for generating high throughput CNN implementations on FPGAs," in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays*, 2018, pp. 117–126.
[50] X. Wei *et al.*, "Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs," in *Proc. Design Autom. Conf.*, 2017, p. 29.
[51] X. Lin *et al.*, "LCP: A layer clusters paralleling mapping method for accelerating inception and residual networks on FPGA," in *Proc. ACM Design Autom. Conf.*, 2018, p. 16.
[52] X. Zhang *et al.*, "AccDNN: An IP-based DNN generator for FPGAs," in *Proc. IEEE Int. Symp. Field Program. Custom Comput. Mach.*, 2018, p. 210.
[53] J. H. Ko, B. Mudassar, T. Na, and S. Mukhopadhyay, "Design of an energy-efficient accelerator for training of convolutional neural networks using frequency-domain computation," in *Proc. ACM Design Autom. Conf.*, 2017, p. 59.
[54] J. Shen *et al.*, "Towards a uniform template-based architecture for accelerating 2D and 3D CNNs on FPGA," in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays*, 2018, pp. 97–106.
[55] A. Podili, C. Zhang, and V. Prasanna, "Fast and efficient implementation of convolutional neural networks on FPGA," in *Proc. IEEE Appl. Specific Syst. Archit. Processors*, 2017, pp. 11–18.
[56] S. Liao *et al.*, "Energy-efficient, high-performance, highly-compressed deep neural network design using block-circulant matrices," in *Proc. IEEE Int. Conf. Comput.-Aided Design*, 2017, pp. 458–465.
[57] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding," *Fiber*, vol. 56, no. 4, pp. 3–7, 2015.



Yun Liang (M'10) received the Ph.D. degree in computer science from the National University of Singapore, Singapore, in 2010.

He is an Associate Professor (with tenure) with the School of EECS, Peking University, Beijing, China. He was a Research Scientist with UIUC, Champaign, IL, USA. His current research interests include heterogeneous computing (GPUs, field-programmable gate arrays, and ASICs) for emerging applications, such as AI and big data, computer architecture, compilation techniques, programming model and program analysis, and embedded system design. He has authored over 70 scientific publications in premier international journals and conferences in related domains.

Prof. Liang was a recipient of the Best Paper Award at Field-Programmable Custom Computing Machines 2011 and International Conference On Computer Aided Design (ICCAD) 2017 and best paper nominations at Design Automation Conference 2012 and 2017, Asia and South Pacific Design Automation Conference (ASPDAC) 2016, International Conference on Field Programmable Technology 2011, and International Conference on Hardware/Software Codesign and System Synthesis 2008. He serves as an Associate Editor for *ACM Transactions in Embedded Computing Systems* and serves on the program committees of premier conferences in the related domain, including, International Symposium on High-Performance Computer Architecture, International Conference on Parallel Architectures and Compilation Techniques, International Symposium on Code Generation and Optimization, ICCAD, International Conference on Supercomputing, International Conference on Compiler Construction, Design, Automation and Test in Europe, International Conference on Compilers, Architectures, and Synthesis for Embedded Systems, ASPDAC, and International Conference on Computer Design.

Liqiang Lu received the B.S. degree from the Institute of Microelectronics, Peking University, Beijing, China, in 2017, where he is currently pursuing the Ph.D. degree with the School of EECS.

His current research interests include algorithm-level and architecture-level optimizations of field-programmable gate array for machine learning applications.

Qingcheng Xiao received the B.S. degree from the School of Electronics Engineering and Computer Science, Peking University, Beijing, China, in 2016, where he is currently pursuing the Ph.D. degree with the Center for Energy-Efficient Computing and Applications.

His current research interests include accelerating deep neural networks and heterogeneous computing.

Shengen Yan received the B.S. degree from the Harbin Institute of Technology, Harbin, China, in 2009, and the Ph.D. degree from the Institute of Software, Chinese Academy of Sciences, Beijing, China, in 2014, respectively.

He was a visiting student with NC State University, Raleigh, NC, USA, from 2013 to 2014. He was a Post-Doctoral Researcher with the Multimedia Laboratory, Chinese University of Hong Kong, Hong Kong, from 2015 to 2017. He is currently the Research and Development Director of the Algorithm Platform Department, SenseTime, Hong Kong. He has published about 20 papers in the areas of parallel computing and deep learning. His current research interests include large-scale deep learning and high performance computing.