[DL] Instruction Driven Cross-layer CNN Accelerator For Fast Detection on FPGA

JINCHENG YU, GUANGJUN GE, YIMING HU, XUEFEI NING, JIANTAO QIU, KAIYUAN GUO, YU WANG AND HUAZHONG YANG, Tsinghua University, China

In recent years, Convolutional Neural Networks (CNNs) have been widely applied in computer vision and have achieved significant improvements in object detection tasks. Although there are many optimizing methods to speed up CNN-based detection algorithms, it is still difficult to deploy detection algorithms on real-time low-power systems. FPGA has been widely explored as a platform for accelerating CNN due to its promising performance, high energy efficiency, and flexibility. Previous works show that the energy consumption of CNN accelerators is dominated by the memory access. By fusing multiple layers in CNN, the intermediate data transfer can be reduced. However, previous accelerators with the cross-layer scheduling are designed for a particular CNN model. In addition to the memory access optimization, the Winograd algorithm can greatly improve the computational performance of convolution.

In this paper, to improve the flexibility of hardware, we design an instruction driven CNN accelerator, supporting the Winograd algorithm and the cross-layer scheduling, for object detection. We modify the loop unrolling order of CNN, so that we can schedule a CNN across different layers with instructions, and eliminate the intermediate data transfer. We propose a hardware architecture to support the instructions with Winograd computation units, and reach the state-of-the-art energy efficiency. In order to deploy image detection algorithms onto the proposed accelerator with fixed-point computation units, we adopt the fixed-point finetune method which can guarantee the accuracy of the detection algorithms.

We evaluate our accelerator and scheduling policy on Xilinx KU115 FPGA platform. The intermediate data transfer can be reduced by more than 90% on the VGG-D CNN model with the cross-layer strategy. Thus, the performance of our hardware accelerator reaches 1700 GOP/s on the classification model VGG-D. We also implement a framework for object detection algorithms, which achieves 2.3× and 50× in energy efficiency compared with GPU and CPU respectively. Compared with floating-point algorithms, the accuracy of the fixed-point detection algorithms only drops by less than 1%.

CCS Concepts: • Computer systems organization → Embedded hardware; Real-time system architecture;

Additional Key Words and Phrases: FPGA, Image Object Detection, CNN

ACM Reference Format:

1 INTRODUCTION

Image object detection is a challenging and widely researched computer vision problem. The object detection task requires that the algorithm tells not only what objects are in the image, but also

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2017 Association for Computing Machinery.
1936-7406/2017/12-ART11 $15.00
https://doi.org/0000001.0000001

ACM Transactions on Reconfigurable Technology and Systems, Vol. 9, No. 4, Article 11. Publication date: December 2017.
the location of each object. Traditional algorithms suffer from low accuracy and long execution time. Recently, CNN-based algorithms like Faster-RCNN (Region CNN) [22], YOLO (You Only Look Once) [21] and SSD (Single Shot multibox Detector) [16] have significantly improved the accuracy of object detection. However, these CNN-based object detection algorithms require high computational complexity and huge storage capacity compared with traditional algorithms. For example, the convolution layers of SSD (based on VGG-D [24]) have 20M parameters and a single forward pass of SSD needs more than 60G operations [24]. Thus, specific hardware architectures are designed to accelerate CNN on ASIC [3–6] and FPGA [1, 14, 17, 18, 20, 29].

Previous works have discovered that the energy consumption of CNN accelerators is dominated by the memory access. The power consumption of transferring a 32-bit number from DDR is three orders of magnitude higher than the power of an add operation [12]. Moreover, the high latency and low bandwidth of DDR access limits the utilization rate of the computation units [11], because the computation units may stay idle waiting for the data transferred from DDR. For CNN, there are two kinds of data that need to be transferred between on-chip and off-chip memory: weights, and featuremaps. The weights need to be transferred to on-chip memory at least once. However, there is no need to transfer every intermediate featuremap between on-chip and off-chip memory. Previous work proposes a cross-layer scheduling method to minimize intermediate data transfer [1]. This work uses on-chip memory instead of off-chip memory, to buffer the intermediate data between different layers, saving the effort of transferring intermediate featuremaps. However, the hardware of this work is designed for a particular CNN model, and the FPGA needs to be reconfigured in order to support another CNN model, so it lacks flexibility [1]. Though FPGA can be reconfigured in a second, the performance will decline in some conditions where the target network changes rapidly and the FPGA is reconfigured frequently. For example, in automatic driving, a CNN firstly tell the environment is complex or not. In a complex environment, another CNN is used to find out the obstacles rapidly. In a simple scene, there are only a few obstacles, and thus a third CNN is used to tell more detail information about the obstacles. In this case, the results of the first CNN decide the following computation flow, and the instruction driven accelerator is much more flexible and effective than reconfiguring the FPGA with different bitstreams.

We modify the loop unrolling order of convolution so that different CNN models can be deployed onto the hardware by different instruction flows, without reconfiguring the FPGA. The platform can thus quickly switch between different tasks to adapt to various applications. Furthermore, we propose a general network dividing method to minimize the data transfer of featuremaps for various networks of different memory sizes.

Moreover, mathematical transformations, such as Winograd transformation [25], can accelerate convolution with the same computation consumption. To leverage the resources on FPGA, we introduce the Winograd algorithm to our instruction set and hardware design.

In order to deploy image detection algorithms on the fixed-point computation unit, we also propose the fixed-point finetune method, to guarantee the accuracy of the detection algorithms on our hardware platform, harnessing FPGA’s great capacity of fixed-point number operation.

To sum up, we firstly optimize the cross-layer strategy in [1] and propose the Transfer Minimum Dividing (TMD) method to minimize the intermediate data transfer for various networks [27]. We extend the TMD method in our previous work [27] to CNN with branches. Furthermore, we propose an instruction driven object detection system with the following contributions:

- We package the basic operations of CNN-based detection methods as instructions and propose an instruction set, with a compiler mapping a CNN model to the instructions for the hardware. The Winograd algorithm is also introduced to our instruction set and hardware.
• We propose the fixed-point fine tune method to guarantee the accuracy of CNN-based object detection algorithms on FPGA. Compared with floating-point algorithms, the accuracy of the fixed-point detection algorithms only drops by less than 1%.
• We implement the object detection system based on our instruction set on a FPGA platform, Xilinx KU115, and achieve 2.3× and 50× in energy efficiency compared with GPU and CPU respectively.

The overall structure of the design flow is shown in Fig 1. First, the fixed-point finetune method converts the original CNN model to an optimized fixed-point model. Second, the TMD method divides the layers into several Layer Blobs to minimize data transfer. At the compilation stage, each Layer Blob is compiled into an instruction flow, which can be executed sequentially. After the instructions and the fixed-point weights are generated, the hardware calculates the results of CNN. The rest of this paper is organized as follows. Section 2 introduces the background of CNN-based object detection and Winograd transformation. Section 3 introduces the previous works. Section 4 proposes the TMD method. Section 5 introduces the accelerator architecture. Section 6 goes deeper into the instruction set. Section 7 introduces the optimization methods. We show the experimental results in Section 8. Section 9 summarizes this paper.

2 BACKGROUND
In this section, we introduce the background of hardware CNN object detection design. Firstly, we give a brief introduction of CNN, and analyze some of the state-of-the-art object detection algorithms based on CNN. Secondly, we introduce the technique to accelerate CNN with Winograd method.

2.1 Background of CNN-based Detection

2.1.1 Basic Operations of CNN. CNN achieves state-of-the-art performance on a wide range of computer vision tasks. To help understanding CNN, the basic operations in CNN are introduced in this section. A typical CNN consists of some layers running in sequence. The data between different layers are called featuremaps. There are four kinds of layers in CNN: convolution layers, non-linear layers, pooling layers and full-connecting (FC) layers.

Convolution layers process 2-D convolution with trained parameters. Convolution layers are usually cascaded to extract high-level features from the input. The basic operation of convolution layers is the 2-D convolution of the input featuremaps and convolution weights. Some fast algorithms, such as FFT and Winograd, can accelerate the calculation of 2-D convolution.
Non-linear layers apply a non-linear activation function to each element in the output feature maps of convolution layers. Non-linear layers enhance the fitting capability of CNN. Rectified Linear Unit (ReLU) is a widely used non-linear activation function in CNN, which keeps the positive elements and eliminates the negative elements.

Pooling layers increase the receptive field of output elements and reduce the computational complexity of CNN. A pooling layer outputs the max or average value of all elements in an area of the input featuremaps.

FC layers fully connect all neurons in the input and output layers. The weights of a FC layer can be denoted as a matrix, and the outputs can be computed from matrix-vector multiplication. Some CNN models, such as SSD, have abandoned FC layers for higher performance.

2.1.2 CNN-based Object Detection Algorithms. In the traditional object detection methods without CNN, the algorithms propose possible regions of objects (called bounding boxes) and send each possible bounding box to a classifier to determine whether there is an object in the region, as well as its category. As CNN has already achieved high performance in object classification tasks, it is natural to use CNN as the classifier for object detection, also this is the idea behind the original RCNN (Region CNN) [10]. In RCNN, a complete CNN forward is computed for each proposed bounding box, and thus RCNN is very slow. Fast-RCNN [9] reuses the feature extraction layers for all proposal boxes, and runs the respective classification layers for each proposal box. Region of interest Pooling (ROI Pooling) is introduced in Fast-RCNN. ROI Pooling resizes an arbitrary region in the output featuremaps to a unified shape. The complicated region proposal algorithms limit the performance of this framework. Furthermore, Faster-RCNN [22] uses a network, called a region proposal network (RPN), to generate possible regions. The RPN can also reuse the feature extraction layers in Fast-RCNN.

However, all of these RCNN-based object detection methods need to calculate the classification layers individually for each proposed region. The repeated computation of the classification layers is still computationally intensive. Some works design one shot framework to accelerate object detection, such as YOLO [21] and SSD [16]. The network structures of these two CNN models are illustrated in Fig 2. These frameworks detect possible objects for each category directly from the output of a CNN model. YOLO uses a simple CNN with FC layers, without any branches, while SSD generates possible objects from branches of intermediate featuremaps, without FC layers. The outputs of CNN-based detection algorithms represent hundreds of probable bounding boxes and many of these bounding boxes represent the same object. The non-maximum suppression (NMS) method is widely used to refine the output bounding boxes by keeping the bounding box with the highest confidence and removing other bounding boxes nearby. Since YOLO and SSD achieve comparable accuracy to the RCNN-based methods and are much faster, we mainly discuss these two object detection frameworks. To evaluate the performance of a detection method, the mean average precision (mAP) is adopted as the final metric of detection. The mAP indicates the mean accuracy of each class considering recall and precision. Generally, higher mAP indicates better performance.

Generally, there are four components in these object detection algorithms: region proposal methods, ROI pooling for each proposal, FC layers, and convolution layers. The components used in each framework are illustrated in Table 1. With the development of algorithms, the object detection frameworks become increasingly simple, with fewer components. The advanced SSD framework only consists of convolution layers, and SSD consumes a relatively less computational complexity with a higher accuracy compared with other detection frameworks. However, the structure of the convolution component itself is increasingly complex, and some more features, such as convolution with dilation/stride and branches, are introduced to SSD.
Table 1. The components used in different detection frameworks and the complexity and accuracy on VOC [7] set of each framework

<table>
<thead>
<tr>
<th>Year</th>
<th>Components</th>
<th>Complexity (GOP)</th>
<th>Accuracy (mAP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fast-RCNN [9] 2014</td>
<td>Proposal ✓ ✓ ✓ ✓</td>
<td>-1</td>
<td>70%</td>
</tr>
<tr>
<td>Faster-RCNN [22] 2015</td>
<td>Proposal ✓ ✓ ✓ ✓</td>
<td>300</td>
<td>73%</td>
</tr>
<tr>
<td>YOLO [21] 2016</td>
<td>Proposal ✓ ✓</td>
<td>40</td>
<td>63%</td>
</tr>
<tr>
<td>SSD [16] 2016</td>
<td>Proposal ✓ ✓</td>
<td>52</td>
<td>74%</td>
</tr>
</tbody>
</table>

1 The region proposal algorithms limit the performance of Fast-RCNN.

2.2 Background of Fast Algorithms for CNN

Convolution layers consume most of the computation in CNN. There are several methods to reduce computation complexity of 2-D convolution. Winograd’s minimal filtering algorithm is a fast method to compute 2-D convolution. The Winograd algorithm uses a transformation to compute convolution [25]. For example, to calculate a convolution for which the image tile is $4 \times 4$ and the kernel tile is $3 \times 3$, the Winograd algorithm uses 16 multiplications. Contrastively, the standard algorithm of the same size uses 36 multiplications. The Winograd algorithm achieves a $2.25 \times$ speedup with the same number of multiplications. Fig 3 illustrates the computation flow for a tile of $4 \times 4$ with the standard method and the Winograd method. The implementation of Winograd on hardware will be detailed in Section 5 and illustrated in Fig 8.

The Winograd algorithm can be written in matrix form as:

$$Y = A^T \left[ \left(GgG^T \right) \otimes \left(B^T dB \right) \right] A$$

(1)

The matrices are:

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

(2)
Fig. 3. Convolutions with (a) the standard method and (b) the Winograd method

The $\otimes$ indicates elementwise multiplication, $d$ is the $4 \times 4$ input tile, and $g$ is the $3 \times 3$ convolution kernel. The multiplications with these transformation matrix elements can all be converted to shift operations (such as $\times \frac{1}{2}$), which are hardware-friendly.

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

With the help of Winograd, fewer operations are needed, but that does not directly translate into acceleration always. We need to keep the PEs busy all the time without pipeline stalling. However, the pipeline stalls when the bandwidth between on-chip and off-chip memories is limited. To achieve higher utilization rate of PEs, we use cross-layer scheduling method to reduce the data transferred between on-chip and off-chip memories and make the bandwidth requirement less than traditional layer-by-layer scheduling methods.

3 RELATED WORKS AND MOTIVATION

In this section, the related works are introduced. Furthermore, the motivation of this paper is given alongside to the related works.

3.1 Related Works

3.1.1 Calculation Units Optimization. Some previous works are focused on optimizing the convolution calculation units. The designed calculation units can reuse the feature maps and weights of CNN. Qiu [20] proposes an architecture consisting of hardware convolvers: each PE is a convolver that includes nine multipliers and an adder tree. Chen [5] divides the convolution into 1-D convolution primitives and uses 1-D convolver PE. Du [6] proposes an architecture using multiply-accumulate units (MAC), which involves inter-PE data flow for data reuse. Diannao and DaDiannao [3, 4] also design accelerators for matrix multiplication and convert the computations of convolver to matrix operations. Zhang [28] presents a uniform representation for convolutional layers and FC layers, which can share the same computing resources. However, the previous accelerators optimizing the convolution calculation units cannot reduce the computation consumption of CNN. Zhang [30] implements FFT on a FPGA platform for CNN, but it shows little reduction of computation complexity with small filters. Lu [18] and Utku [2] implement CNN accelerators with
the fast Winograd algorithm and reach significant speedup. We also use the Winograd algorithm in this paper for higher peak performance.

3.1.2 Memory Access Optimization. Besides optimizing calculation units, recent works have turned to optimizing the complete CNN. Memory transfer has a strong impact on the performance of an accelerator. Previous works [5, 20] that utilize off-chip memory discuss the scheduling or data arrangement strategy to overcome the bandwidth limitation between the external and the on-chip memory. Liu [17] implements a structure for each layer of a CNN, and the structures for different layers run in pipelines for different input images. The distribution of hardware resources in different layers in the pipeline structure results in high latency when running a single image. Alwani [1] introduces a fuse-layer policy and reduces the transfer of intermediate data, but Alwani implements the fuse-layer policy with pipeline structure. We aim to introduce cross-layer parallelism to the accelerator for data transfer reduction, and discard pipeline structure for lower latency.

3.1.3 Flexibility. The network structures of CNN are increasingly complicated. Some works design hardware considering flexibility for complicated structures and the demand for CNN updating. Ma [19] proposes a compilation flow from network description with RTL modules. Aydonat [2], Zhang [31] and Xiao [26] use OpenCL or HLS tools to implement CNNs on hardware. However, these previous works, which translate a CNN model to a fixed hardware structure is not runtime reconfigurable. The performance will decline in some conditions where the target network changes rapidly and the FPGA is reconfigured frequently. NeuFlow [8] is a runtime reconfigurable processor with compilation tools. Cambricon [15] is an instruction set designed for neural networks. It translates all of the calculations, including convolutions, to matrix multiplication and schedules CNN layer by layer. Guo [11] also proposes an instruction-driven accelerator, but the massive data transfer between on-chip and off-chip memory limits the performance. We design an instruction set allowing flexibility for implementing different CNN models on a fixed hardware architecture.

3.2 Motivation
We target processing CNN-based object detection algorithms on hardware in this work. As described in Table 1, the SSD image detection algorithm is much simpler and consumes less computational complexity. Therefore, we target SSD as the object detection algorithm. However, it is still difficult to implement SSD real time on low power equipments. Since memory access becomes the bottleneck of energy efficiency on CNN accelerators [12], we introduce the cross-layer scheduling method to minimize data transfer. Furthermore, we want to implement different algorithms, such as object detection and image classification, in the same hardware, without reconfiguring the FPGA. Therefore, we propose an instruction set to accelerate CNN in the cross-layer scheduling. Moreover, the Winograd transformation can improve the system performance by several times, with little extra resources consuming. Thus, we introduce the Winograd algorithm to accelerate convolution in our instruction set and hardware design. With the help of these techniques, we implement the SSD algorithm real time on Xilinx KU115 platform.

4 SCHEDULING OPTIMIZATION
Tradititional layer-by-layer strategy needs to finish a whole layer before starting the next layer, which needs the off-chip memory when the results of a layer cannot be stored on-chip. Previous works[1, 27] have proposed the cross-layer strategy with or without instructions to reduce the data transfer. In this section, we expand these previous works to the framework for scheduling CNN across different layers with network branches. To minimize the data transfer between on-chip and off-chip memory, we propose a network dividing method (Transfer Minimum Dividing, TMD method) to decide which layers from the original CNN model can be fused. After fusing layers into
4.1 Network Dividing Method to Minimize Data Transfer

We propose a network division method to minimize the data transfer of off-chip memory. We name this method a **Transfer Minimum Division** (TMD) method.

The basic idea of the TMD method is to store intermediate data, as much as possible, on-chip. When it is impossible to cache all intermediate data on-chip, the TMD method will find the best scheduling to minimize intermediate data transfer.

Fig 4 shows the input, output, and weight size of each layer for VGG-D network. For example, layer 1 requires 150KB of input and produces 3.4MB of output featuremaps. Since the on-chip memory is limited on FPGA (2MB for intermediate data in our design), the hardware normally cannot provide such a large on-chip memory for all outputs. Therefore, data transfer occurs, to store outputs to off-chip DDR, and to fetch the outputs from DDR for subsequent computation. However, we note that the output size of layer 2 is only 750KB, which can be stored in the on-chip cache of our target FPGA platform, so we do not need to store all of the outputs of layer 1. We can directly compute the output of layer 2 using the partial output of layer 1. In this way, we consider layer 1 and layer 2 as a single **Layer Blob** and eliminate the data transfer within this Layer Blob.

### 4.1.1 Hardware Constraints

As described above, layers can be merged into a Layer Blob and there is no data transfer within a Layer Blob. The size of on-chip memory decides which layers can be fused into a Layer Blob. If some layers (for which the first layer ID is $j$, and the final layer ID is $i$) can be fused, the weights of each of these layers should be stored on-chip. As shown in Equation (3), the weight of the $k^{th}$ layer is $\text{weight}_k$ and the size of the on-chip weight buffer is $\text{Buffer}_\text{weight}$.

$$\sum_{j \leq k < i} \text{weight}_k \leq \text{Buffer}_\text{weight}$$

There is no intermediate featuremap transfer within a Layer Blob. At the interface of two Layer Blobs, if the outputs ($\text{Data}_{out}$) can be stored on-chip, there is no data transfer. If the output size exceeds the on-chip memory, the data transfer volume will be double $\text{Data}_{out}$, because the data transferred to the off-chip memory should be retrieved back onto chip for the subsequent computation. The data transfer of the Layer Blob ending with layer $i$ is denoted as $BT_i$ in Equation (4).
4.1.2 Dividing Method. We can define the description function of total data transfer ($DT(M)$) under a division method ($M$). It is the sum of data transfer in each Layer Blob.

$$DT(M) = \sum_{i \in \text{blobs}} BT_i$$

Our goal is to find a network dividing method such that the data transfer of the CNN is minimized, i.e., find a solution $M^*$ under the hardware constraints governed by Equation (3), such that:

$$DT(M^*) = \min(DT(M))$$

We convert the CNN model to a Directed Acyclic Graph (DAG). In the DAG, a node represents the output of a certain layer. Each node has a weight, which means the amount of the data transfer ($BT_i$) here, given by Equation (4). We add a property $\text{lastnode}$ to a certain node. $\text{lastnode}$ indicates whether each node is the end layer of a Layer Blob or not, if so, $\text{lastnode}$ also indicates the beginning layer of the same Layer Blob. The $\text{lastnode}$ of all nodes form the dividing method $M$ of a CNN network.

We can find a collection of subproblems of the overall problem in Equation (6). For the $i^{th}$ node, we define the subproblem $P(i)$ as: find the optimal division solution for the graph before the $i^{th}$ node. It can be easily proven that the problem has the optimal substructure. Thus, the overall problem can be solved using dynamic programming. To analyze the problem, we define function $P(i)$ of the subproblem as follows:

$$H(i) = \{x|x \leq i\}$$

$H(i)$ is the set of nodes that come before the $i$th node.

$$P(i) = \min_{M \subset H(i)} (DT(M))$$

It is obvious that for the last node $n$, we have:

$$P(n) = DT(M^*)$$

Hence, our goal is to find $P(n)$ and the corresponding $M$. The Bellman Equation (a basic function in dynamic programming) of the subproblem $P(i)$ can be written as:

$$P(i) = \min_{j} (P(j) + BT_j)$$

where $j$ is a node before $i$, and both $i$ and $j$ satisfy the hardware constraints given in Equation (3). The extra data transfer $BT_j$ is added to the total data transfer at the node $j$.

With the given optimal substructure and Bellman Equation, we use dynamic programming to find the target $P(n)$. For each node in the network, the optimal division solution is one of the optimal division solutions for node $j$ plus the weight at node $j$. Thus, we can describe our process using pseudocodes. This is shown in Algorithm 1.

The Check ($\cdot$) function checks the satisfiability with Equation (3). At each loop, a new layer is added to the search range. Repeating the steps until the last node of the network leads to the optimal division solution.

The network division method shows the minimum data transfer under the constraint of on-chip memory and presents the division of layers to Layer Blobs.
Algorithm 1 TMD: Transfer minimum division method

1: \( // bt[i] \) is the BT in Equation (4), and \( lastnode \) gives a division method \( M \) in Equation (5).
2: \( P[0] = 0 \)
3: \( lastnode[0] = 0 \)
4: \( \textbf{for } i = 1 : n - 1 \textbf{ do} \)
5: \( P[i] = +\infty \)
6: \( \textbf{for } j = 0 : i - 1 \textbf{ do} \)
7: \( \textbf{if } \text{Check}(i, j) \textbf{ then} \)
8: \( \textbf{if } P[j] + bt[j] < P[i] \textbf{ then} \)
9: \( P[i] = P[j] + bt[j] \)
10: \( lastnode[i] = j \)
11: \( \text{Layer Blob List} = \text{divide}(lastnode) \)

4.1.3 Network Branches. The object detection networks contain branches for multi-scale detection. Previous works [1, 27] based on cross-layer cannot handle networks with branches, resulting in failure to deploy SSD on hardware. We propose a depth first search (DFS) network dividing method to split the original network into several subnetworks. Each subnetwork contains no branches, and so can be scheduled using the cross-layer strategy. The detail of the cross-layer strategy will be displayed next to this subsection at Section 4.2.

Our DFS network dividing method searches the original network from the input layer and creates a subnetwork, starting with the input layer. If a layer has not been searched, we search the layer and add this layer to the current subnetwork. If branches occur at a layer, we finish constructing the current subnetwork and start new subnetworks for each branch. Our method carries out the DFS search method recurrently until every layer in the original network has been searched. An example of our DFS network dividing method is illustrated in Fig 5.

4.2 Cross-layer Scheduling

After dividing layers into Layer Blobs, our cross-layer scheduling method calculates the layers within each Layer Blob, without data transfer between the on-chip cache and the off-chip DDR. In this section, we will introduce the basic concept of our cross-layer scheduling. Furthermore, in Section 6.4, an example of scheduling CNN with our ISA will be given to help readers understanding the benefits of this technology.

A typical CNN consists of some layers running in sequence, and previous works usually schedule CNN layer by layer. For this reason, previous works often focus on the acceleration of a single layer,
and all of the featuremaps of an intermediate layer are stored as a whole. However, it is not necessary to maintain the complete featuremaps of intermediate layers. The scheduling flow of the fuse layer in [1] is illustrated in Algorithm 2. The $N_i$ input featuremaps with size $W \times H$ are convolved with $k_x \times k_y$ weights in order to get the $N_o$ output feature maps. [1] modifies the traditional loop unrolling order, which put the loop of the layer inside the loop of row and column (line 3). Because the computation structure of each layer differs from the others, it designs different computation units and data reuse units for different layers. The FPGA accelerator in [1] can only run a specific network and needs to be reprogrammed to run another network.

We modify the loop unrolling strategy in [1] to make the cross-layer scheduling policy more flexible. We propose the cross-layer scheduling strategy in line, which is shown in Algorithm 3.

The row is regarded as the outermost loop variable (line 1). Each time, the computation window moves down by two lines and performs the computation through the whole Layer Blob. Furthermore, convolution operations are done by the fast Winograd computation kernel.

Fig 6 shows our cross-layer scheduling strategy, with an example of considering two convolutional layers as a whole. The size of the input featuremap (featuremap 1) of the 1st layer is $5 \times 8$, and the size of the convolution kernels of both layers is $3 \times 3$. Layer 1 takes featuremap 1 as its input and produces featuremap 2. Layer 2 produces featuremap 3 from featuremap 2. In this example, there is no padding or pooling operation, and the stride of the convolution is 1.

First, layer 1 takes lines 1-4 of featuremap 1 as its input and produces lines 1-2 of featuremap 2 using the Winograd algorithm. Secondly, lines 3-4 of featuremap 2 are calculated. After lines 1-4 of featuremap 2 are produced and stored on-chip, layer 2 calculates lines 1-2 in featuremap 3. As soon as lines 1-2 in featuremap 3 are produced, lines 1-2 in featuremap 2 can be discarded. The result of

---

**Algorithm 2** conventional convolution layer in [1]

1. for (row = 0; row < H; row++) do
2.   for (col = 0; col < W; col++) do //H,W indicate the size of last Layer
3.     for (layer = 0; layer < L; layer++) do
4.       //The following parameters, $N_i, N_o, k_x, k_y$, differ in different layers, i.e. different L.
5.         for (fi = 0; fi < N_i; fi++) do
6.           for (fo = 0; fo < N_o; fo++) do
7.             for (i = 0; i < k_x; i++) do
8.               for (j = 0; j < k_y; j++) do
9.                 out[row][col][fo] += weight[fi][fo][i][j] * in[row+i][col+j][fi]

**Algorithm 3** convolution layer computation with the Winograd

1. // Our Winograd algorithm produces 2 line of outputs simultaneously, so row += 2
2. for (row = 0; row < H; row+= 2) do //H indicate the height of last Layer
3.   for (layer = 0; layer < L; layer++) do
4.     //The following parameters, $N_i, N_o, k_x, k_y, W$, differ in different layers, i.e. different L.
5.       for (fo = 0; fo < N_o; fo++) do
6.         for (fi = 0; fi < N_i; fi++) do
7.           //Our Winograd algorithm produces 2 column of outputs, so col += 2
8.         //Parameter W indicates the width of the current layer.
9.       for (col = 0; col < W; col+= 2) do
10.      // [x+k] means [x:x+k]. For example, [row+1] means [row:row + 1]
11.     out[row+1][col+1][fo] += Winograd (in[row+1][col+1][fi], weight[fi][fo])

...
step 4 (lines 5-6 of featuremap 2) can be stored at the same address as lines 1-2. Finally, line 3-4 of featuremap 3 are produced from lines 3-6 of featuremap 2.

Step 4 is the critical step in this example. By storing the latest outputs (lines 5-6) to the same address as the discarded old lines (lines 1-2), we only need to store four lines of featuremap 2 on-chip. Generally, there are always two lines that can be discarded when two lines of the next featuremap is computed. If there are more than two layers in a Layer Blob, we only need to store four lines of each of the intermediate featuremaps. Because we don’t need to store the entire intermediate featuremap on-chip, we can cache the intermediate data on the memory-limited chip, without data transfer between the on-chip cache and the off-chip DDR.

Each step in Fig 6 is treated as the basic operation in our design. The basic operation to calculate two lines of the output featuremap is packaged into a CAL instruction. The details of our instruction set will be given in Section 6.

5 HARDWARE DESIGN

In this section, we propose a structure to combine the Winograd and the cross-layer scheduling. The control of the data flow will be indicated by the instructions described in Section 6.

5.1 Architecture Overview

Fig 7 presents the architecture overview of our implementation of the Winograd and the cross-layer scheduling on FPGA. To store input data and output data in the same buffer, the hardware parallelism of input channels and output channels should be the same. An input register file is designed to shuffle data in case of zero-padding. Weights are fetched from Weight Buffer in $3 \times 3$. PEs compute Equation (1) with the input tile and weights. We also design an output register buffer data in case of pooling. In our design, there are several Buffer Pools and Weights Buffer for featuremaps and weights in different channels. The controller takes the charge of decoding instructions and manages the memories and PEs on chip.

5.2 Winograd PE Design

Fig 8 shows the structure of PEs. The calculation of the convolution layers can be divided into four stages. The first stage is the transformation of the input tile and weights. Note that the
transformation of the input data and weights can be divided into two constant matrix multiplications; the second multiplication of input tiles is arranged in DSPs. The second stage is the element-wise multiplication of the transformed input tile and weights. We use DSPs to perform this operation. The third stage is the transformation of the result. The fourth stage is to accumulate the output tiles from different input channels.

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

The computation on hardware also parallelizes the loops of input channel and output channel. We define the unroll factors of input channel and output channel as \(P_i\) and \(P_o\). Therefore, there are a total of \(P_i \times P_o\) PEs in the hardware design. Moreover, in order to store input data and output data in the same buffer, \(P_i\) is the same as \(P_o\). Each PE consists of 16 DSPs. Therefore, \(P_o\) and \(P_i\) are constrained by the number of DSPs on the FPGA platform, according to Equation (11). The number of DSPs is denoted as \(n_{DSP}\). As there are 2 DDRs on the FPGA board and the FPGA chip consists of...
2, we set 2 computation kernels in our design. In the experiments, we set \( P_i = P_o = 8 \) in each computation kernel.

\[
P_i \times P_o \times 16 \times 2 \leq n_{DSP}
\]  \hspace{1cm} (11)

In our current design, the kernel size of convolution is fixed to \( 3 \times 3 \). Although the kernel size is fixed, we are still able to support other kernel sizes as shown in Fig 9. For smaller kernels like \( 1 \times 1 \), the kernel is padded to \( 3 \times 3 \). For larger kernels, such as \( 5 \times 5 \), multiple \( 3 \times 3 \) kernels are used to cover it. However, the utilization rate of our hardware will decrease when the kernel size is not \( 3 \times 3 \).

### 5.3 Data Buffer Design

Previous works usually adapt line buffer structure for data reuse [18, 20]. Initially, the line buffer will read the first \( M \) lines from input data. PE will stay idle until the \( M \) lines of input data are fully read. Considering the scheduling of the cross-layer strategy mentioned in Section 4.2, the basic operation is to calculate a line of the output featuremap. After finishing a line of the output featuremap, the next operation will start to compute a new featuremap, and the PE will return to idle, waiting for the \( M \) lines of this new featuremap. In the cross-layer scheduling, the redundancy of reading the first \( M \) lines will keep PEs idle most of the time, resulting in low utilization of PEs, and this redundancy is inevitable when using line buffer to cache data. However, the line buffer structure is suitable for weights buffer, because the lines of weights are much shorter. The redundancy of the two lines of weights can also be covered by pre-loading and registering the weights.

In order to support the cross-layer scheduling, the input data buffer and the output data buffer should be merged into a single buffer, called **buffer pool** in our implementation. Each buffer pool consists of 4 RAMs, and each ram provides 4 input data per clock cycle. The addresses of RAMs are given in the instructions, and instructions also provide the writeback addresses of RAMs to store the result of the convolution layer. All PEs share the input data from the buffer pool. However, the weights are not shared among PEs, so there is one weight buffer corresponding to each PE.

### 5.4 On-chip Memory Allocation

We further improve the utilization rate by presenting a dedicated memory allocation strategy for featuremaps. During the cross-layer computation stage of our system, all the input features are loaded. All of the output feature maps will be written back after computation. A single tile of the intermediate features is reserved for each layer’s computation. In each outermost loop of the cross-layer computation, there are two lines of the input feature that will not be used again. To make use of the released memory, we employ the structure of **circular queue** to the featuremap buffer, for on-chip memory management. Whenever two lines of the input are released, the read pointer will move forwards by the exact size of the data. The released space can be used for later writes.

### 6 INSTRUCTION SET

We propose an Instruction Set Architecture (ISA) for our NN accelerators. Different from other instruction sets, such as Cambricon [15], which aims to support various NN algorithms, such as MLP, CNN and LSTM, our ISA is designed specifically for CNN to support the cross-layer scheduling and the Winograd acceleration. Our proposed ISA is very simple with only three types of instructions, including the save data instruction (SAVE), load data instruction (LOAD) and calculation instruction.
In_addr Out_addr W_addr opcode 4 32 32 32 CAL op_ex 9 L_width 16

0-3: PAD type
4: Last Channel
5-6: POOL type
7: ReLU type
8: Shuffle

Fig. 10. Top: CAL instruction with data type field. Middle: SAVE instruction. Bottom: LOAD instruction.

CAL: This ISA enables our hardware platform to support different CNNs without reprogramming the FPGA. Each instruction is 128-bit long as shown in Fig 10.

6.1 SAVE/LOAD instruction
The SAVE/LOAD instructions are designed for data transfers between on-chip memory and off-chip memory (DDR). As mentioned in Section 2.1, there are two kinds of data that need to be transferred between the on-chip cache and the DDR: intermediate featuremaps and network weights. At the inference stage of a CNN, the network weights only need to be transferred from DDR to cache. However, the data flow of intermediate featuremaps is bi-directional. The intermediate featuremaps may be transferred from cache to DDR when the data size exceeds the on-chip storage capacity. To calculate a new Layer Blob, the intermediate featuremaps need to be transferred from DDR to cache. The SAVE instruction transfers only intermediate featuremaps from cache to DDR, while the LOAD instruction transfers both intermediate featuremaps and weights from DDR to cache.

As shown in Fig 10, a SAVE/LOAD instruction can save/load $D_{len}$ bytes to/from DDR. The base addresses of on-chip cache ($On_addr$) and off-chip DDR ($Off_addr$) are indicated in the instructions. For the LOAD instruction, there is one extra field to define the data type (intermediate featuremaps or weights).

6.2 CAL instruction
A CAL instruction executes a basic operation of the hardware. As described in Section 4.2, the basic operation of our cross-layer scheduling policy is the calculation of two output lines of a single layer, i.e., a calculation instruction may finish line 9 to line 11 in Algorithm 3. Parameters for calculation, like the number of columns $W$ and the on-board addresses of out, in and weight are given in the instructions as $L_{width}$, $In_addr$, $Out_addr$ and $W_addr$. As illustrated in Fig 10, more options for CNN calculations like padding, ReLU and pooling are also set in calculation instructions. These options are set in the 9-bit $op_{ex}$ field.

In our implementation, the 8-bit $op_{ex}$ field consists of five sub-fields: 4-bit PAD type field, 1-bit Last Channel field, 2-bit POOL type field, 1-bit ReLU type field, and 1-bit shuffle field. Each bit in the PAD type field indicates if there are zero-padding elements in the four directions of the input line (Top/Bottom/Left/Right). The Last Channel field indicates if this CAL instruction has finished the input channel and produces a valid output channel. When a CAL instruction produces a valid output channel, the POOL and ReLU field is valid, and the results will be written back to the buffer pool. Otherwise, if a CAL instruction produces the sub-results from partial input channels, the results will be written to the intermediate data fifo inside PEs for future usage, and...
Fig. 11. Left: Cross-layer scheduling without intermediate data transfer. Right: Layer-after-layer scheduling.

the POOL and ReLU fields remain invalid. The POOL type field indicates if the results of this CAL instruction need to be downsampled and the pool type (Max pooling or Average pooling). The ReLU type field indicates if the non-linear operation is valid on the outputs. Shuffle field is designed for the cross-layer scheduling. When shuffle bit is zero, the four lines of input featuremap are stored in the on-chip cache successively, like stage 3 in Fig 6. Otherwise, if the shuffle bit is valid, the four lines are twisted in the on-chip cache, i.e., the two higher lines are stored ahead of the two lower lines, like stage 5 in Fig 6.

6.3 Instruction dependency

Data transfer instructions and calculation instructions can be executed in parallel. There is a dependency field in each kind of instruction. Dependency field sets the flags for inter-instruction dependency and helps to parallelize different kinds of instructions. This enables scheduling before instruction execution. There are three bits in the dependency field: S bit, L bit and C bit. If S bit in an instruction is valid, this instruction should not be executed until the last SAVE instruction finishes. Similarly, L bit and C bit indicate that the instruction should stay invalid until the last LOAD or CAL instruction finishes.

6.4 Code Examples

Our instruction set can schedule a CNN layer after layer, as well as across different layers. To illustrate the usage of our proposed instruction sets, we give two simple examples to show the code flow of scheduling with/without crosslayer. In Fig 11, we have scheduled the simple example with two layers in Fig 6 in the cross-layer scheduling (left) and plain layer-after-layer scheduling (right). Cross-layer scheduling eliminates the intermediate data transfer, but because of the limited on-chip memory for featuremap 2 (only 12 bytes from 0x100 to 0x10b of Buffer Pool), the plain scheduling has to store intermediate data on the DDR and reload it. The featuremap 3 is saved to address 0x300 on the DDR after both of these scheduling methods finish. The featuremap 1 is saved to address 0x200 on DDR when we use the plain scheduling.
In the plain scheduling, each layer is considered as an individual Layer Blob. If we want to save the intermediate data to DDR, or the on-chip cache is too limited, the plain method, which schedules CNN layer after layer, is an effective way. The number of CAL instructions is the same as that number in the cross-layer or plain scheduling, but the plain scheduling consumes more SAVE/LOAD instructions, which reduce the performance of hardware. The comparison between the plain method and the crosslayer method will be given in Section 8.2.

We implement a compiler to transfer a high-level described CNN model (such as prototxt file in Caffe [13]) to the code flow in Fig 11. An assembler translates the code flow to binary code for hardware. The instructions number for some typical CNNs, and the utilization rate of hardware compared with customized accelerators which design a hardware for a particular CNN, will be given in Section 8.

7 OPTIMIZING OBJECT DETECTION ALGORITHMS FOR HARDWARE

Our hardware cannot efficiently support some operations, such as 1 x 1 convolutions. However, when we use our accelerator in applications like object detection, we can design the network structure considering the hardware constrains. To run the object detection algorithms real time on our FPGA platform, the algorithms need to be adjusted.

7.1 Network Structure

The original SSD model adopts convolution with stride. However, due to the Winograd transformation, we cannot efficiently process convolution with stride. So we redesign the CNN model and use an original convolution layer and a pooling layer to replace the stride convolution.

There are also other extended functions of different layers in original SSD, like convolution layers with dilations and pooling layers with 3x3 kernel size. We remove the extended parameters and reconstruct the network with the basic layers introduced in Section 2.1. We also notice that the accuracy increases when the size of input picture goes larger, so we expand the input size of SSD from 300x300 to 320x320.

The branches from the original conv4-3 layer shown in Fig 2 are followed with a normalization layer which is difficult to be implemented on FPGA. We eliminate the branches from conv4-3 layer to NMS detector and fine tune the network. In Section 8, the experiments show that the performance of the reconstructed network is as accurate as the original SSD.

7.2 Low-bit Compression

Many previous works have exploited that high precision is not necessarily required in the feed forward phase because of the redundancy in the parameters and featuremaps in CNN models [20]. Qiu [20] also proposes a quantization method analyzing the dynamic range and finding the optimized quantization position for each layer. Therefore, we adopt fixed-point arithmetic units in the hardware to replace the floating-point number format in GPU and CPU.

Previous works show that 8-bit quantization for weights and featuremaps is enough for classification tasks [12, 32]. However, these previous works do not fine tune the CNN in fixed point, and lead to the precision loss in image detection tasks. In this paper, with the help of fixed-point fine tune method, we adapt the 8-bit quantization techniques to object detection tasks.

The original network and the reconstructed network are trained in floating-point representation. After translating the floating-point system to the 8-bit fixed-point system, the accuracy descends. So we use the fixed-point fine tune method to improve the performance of fixed-point object detection method.

Fig 12 shows the process of our fixed-point fine tune method. The fixed-point fine tune method is based on the conventional method for CNN training, stochastic gradient descent method (SGD).
There are two phases in SGD: feed forward and back propagation. In our fixed-point finetune method, we use the fixed-point number representation in the feed forward phase and keep floating-point number representation at the backpropagation. After updating all the weights in the backpropagation step, we will analyze the dynamic range of the featuremaps and weights. The feed forward step of the next iteration is processed in fixed-point with the updated quantization point positions.

With the help of the fixed-point finetune method, the 8-bit fixed-point algorithms on FPGA achieve the same level of accuracy as original floating-point algorithms with accuracy losses less than 1%. The detail results of our fixed-point finetune method will be described afterward in Section 8.4.

8 EXPERIMENTS

Although our hardware architecture is designed for object detection algorithms, the instructions and hardware can also support classification networks. To compared with previous works on classification networks, we validate our system on both classification and detection tasks.

8.1 FPGA Experiment Setup

We evaluate our hardware architecture on the Xilinx KCU1500 development board with KU115 FPGA. This platform has 5520 DSPs and four 4GB DDRs. However, the hardware resources are distributed on two dies. To make full use of these two dies, we set $P_i$ and $P_o$ to 8 for one CNN accelerator, and we implement two CNN accelerators on the FPGA chip. Each die contains an accelerator. In this way, the Batch of our hardware is 2. The weight buffer on each accelerator is 2MB, and the data buffer (Buffer Pool) on each accelerator is 1MB. We use PCIe interface to initial DDRs and control the CNN accelerators on board. The system is operated at 250Mhz frequency. We use 8-bit number representation for data storage and computation.

It should be noted that this work concentrates on accelerating convolution layers of CNN. The other operations, such as FC and NMS, are completed on CPU.

8.2 Cross-layer vs Plain Scheduling

Table 2 shows the performance comparison between scheduling policy with and without the cross-layer optimization. We generate instructions for these models to run different CNN networks on the same hardware platform. We use 1MB on-chip BRAM for Buffer Pool and 2MB for weight.
Table 2. Performance comparison between strategy with and without TMD

<table>
<thead>
<tr>
<th></th>
<th>YOLO(^1)</th>
<th>VGG-A</th>
<th>VGG-D</th>
<th>VGG-D (small Cache)(^2)</th>
<th>VGG-D (layer 1-5)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Plain/TMD</td>
<td>Plain</td>
<td>TMD</td>
<td>Plain</td>
<td>TMD</td>
</tr>
<tr>
<td>Problem complexity (GOP)</td>
<td>30.60</td>
<td>16.80</td>
<td>30.60</td>
<td>30.60</td>
<td>-</td>
</tr>
<tr>
<td>Parameter transfer (MB)</td>
<td>60.00</td>
<td>9.33</td>
<td>18.23</td>
<td>18.23</td>
<td>0.76</td>
</tr>
<tr>
<td>Intermediate data transfer (MB)</td>
<td>0.80</td>
<td>6.60</td>
<td>0.18</td>
<td>6.60</td>
<td>0.18</td>
</tr>
<tr>
<td>Total transfer (MB)</td>
<td>60.80</td>
<td>15.90</td>
<td>9.51</td>
<td>21.62</td>
<td>18.41</td>
</tr>
<tr>
<td>#Instruction</td>
<td>238132</td>
<td>101160</td>
<td>101133</td>
<td>165863</td>
<td>165839</td>
</tr>
<tr>
<td>Instruction Size (MB)</td>
<td>3.81</td>
<td>1.62</td>
<td>1.62</td>
<td>2.65</td>
<td>2.65</td>
</tr>
<tr>
<td>Total time (ms)</td>
<td>65.40</td>
<td>25.50</td>
<td>22.40</td>
<td>44.80</td>
<td>41.70</td>
</tr>
<tr>
<td>Speed up</td>
<td>-</td>
<td>12%</td>
<td>7%</td>
<td>6%</td>
<td>-</td>
</tr>
</tbody>
</table>

\(^1\) All intermediate data of any YOLO layer can be stored on board, and the scheduling of with and without cross-layer is equal.

\(^2\) Small Cache condition: Weight buffer is 1MB. Data buffer is 0.5MB.

The other experiments: Weight buffer is 2MB. Data buffer is 1MB.

buffer. As is shown in the last raw in Table 2, around 12% of acceleration is achieved by adapting the cross-layer scheduling on a small model VGG-A, and 7% on a larger model VGG-D.

To evaluate the performance of the TMD method on different on-chip memory sizes, we also implement a hardware using less BRAMs (0.5MB for data and 1MB for weight, listed in VGG-D (small Cache) column). From the row Intermediate data transfer, we can conclude that, the data transfer can be reduced by the TMD method on different memory sizes, about 97% for the large on-chip cache and 69% for the small on-chip cache. A larger cache enables the TMD method to fuse more layers and transfer less intermediate data.

To compare with the cross-layer scheduling method in [1], we also implement layers 1-5 of VGG-D model (listed in column VGG-D (layer 1-5)). Our TMD method, as well as the cross-layer method in [1], can both completely eliminate the intermediate data transfer. So we achieve the same performance as [1].

The number of instructions and the data size of instructions are also listed in Table 2 for each network (#Instruction row and Instruction size row). Compared with the size of total data transfer, the instruction size is relatively small (less than 20% of total data transfer).

### 8.3 Comparison With Previous Works

The performances comparison with other FPGA works are shown in Table 3. Our work achieves state-of-the-art performance on FPGA. The previous works, arXiv17 [2] and FPGA17 [31] are based on OpenCL and evaluated on Altera 10 FPGA. Other works are based on Xilinx platforms. Moreover, our design is an instruction driven accelerator, providing flexibility for different CNN models. FCCM17 [18] and arXiv17 [2] both introduce the Winograd, and MICRO16 [1] introduces the cross-layer technique.
Table 3. Performance comparison with state-of-the-art FPGA accelerators

<table>
<thead>
<tr>
<th>Platform</th>
<th>Clock (MHz)</th>
<th>Data Format</th>
<th>Used DSPs</th>
<th>Batch</th>
<th>Network</th>
<th>Top5-Accuracy</th>
<th>Complexity (GOP)</th>
<th>Power (W)</th>
<th>Performance (GOP/s)</th>
<th>Driven Type</th>
<th>Utilization</th>
<th>Power Efficiency (GOP/s/W)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPGA16</td>
<td>150</td>
<td>16-bit</td>
<td>780</td>
<td>1</td>
<td>VGG-D</td>
<td>87.94%</td>
<td>39</td>
<td>9.63</td>
<td>137</td>
<td>Inst</td>
<td>59%</td>
<td>14.2</td>
</tr>
<tr>
<td>FPL16</td>
<td>156</td>
<td>16-bit</td>
<td>2144</td>
<td>1</td>
<td>AlexNet</td>
<td>77.12%</td>
<td>2.66</td>
<td>30.2</td>
<td>30.2</td>
<td>Cust</td>
<td>85%</td>
<td>18.7</td>
</tr>
<tr>
<td>arXiv17</td>
<td>303</td>
<td>float</td>
<td>1476</td>
<td>1</td>
<td>AlexNet</td>
<td>77.80%</td>
<td>2.66</td>
<td>45</td>
<td>45</td>
<td>Cust</td>
<td>77%</td>
<td>30.7</td>
</tr>
<tr>
<td>FPGA17</td>
<td>385</td>
<td>16-bit</td>
<td>2756</td>
<td>1</td>
<td>VGG-D</td>
<td>87.94%</td>
<td>30.6</td>
<td>37</td>
<td>37</td>
<td>Cust</td>
<td>84%</td>
<td>48.4</td>
</tr>
<tr>
<td>MICRO16</td>
<td>200</td>
<td>float</td>
<td>2987</td>
<td>1</td>
<td>VGG-D</td>
<td>-</td>
<td>11.2</td>
<td>-</td>
<td>-</td>
<td>Cust</td>
<td>16%</td>
<td>-</td>
</tr>
<tr>
<td>FCCM17</td>
<td>200</td>
<td>16-bit</td>
<td>2304</td>
<td>2</td>
<td>VGG-D</td>
<td>87.94%</td>
<td>30.6</td>
<td>23.6</td>
<td>23.6</td>
<td>Inst</td>
<td>83%</td>
<td>129.0</td>
</tr>
<tr>
<td>Ours</td>
<td>250</td>
<td>8-bit</td>
<td>2048</td>
<td></td>
<td>VGG-D</td>
<td>87.94%</td>
<td>30.6</td>
<td>38.1</td>
<td>38.1</td>
<td>Inst</td>
<td>74%</td>
<td>44.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VGG-A</td>
<td>84.28%</td>
<td>30.6</td>
<td>38.3</td>
<td>38.3</td>
<td>Inst</td>
<td>76%</td>
<td>45.6</td>
</tr>
</tbody>
</table>

1 The frequency is estimated to calculate performance
2 Driven Type. Inst represents the hardware is driven by instructions. Cust represents the hardware is customized for a particular CNN.

As different experiments use different FPGA platforms, CNN models, and precisions, it is difficult to make a definitely fair comparison. We list the platform and numerical precision of each implementation. We list the top-5 accuracy of the classification task of ILSVRC [23] as the performance indicator of these CNN models. We can conclude that, the 8-bit/16-bit fixed-point CNN models reach the same accuracy as a floating-point model. The accuracy reduction is no more than 1%.

Peak performance indicates the ability of an accelerator when assuming that the DSPs are always busy and can be calculated by $\#DSPs \times frequency$. We use the utilization rate (actual performance/peak performance) to indicate the efficiency and flexibility when we schedule a network. The utilization rate of our work is very close to other works which customize a hardware for a particular CNN [2, 14, 18, 31] and is much higher than the previous work also driven by instructions [20].

Compared with the previous works with the cross-layer technique (MICRO16 [1]), we improve the performance of convolution layers from 566 GOP/s to 1709 GOP/s. It should be noticed that the Winograd kernel of our design is $4 \times 4$ and it accelerates convolution $2.25 \times$. The Winograd kernel used in FCCM17 [18] and arXiv17 [2] is $6 \times 6$ and it accelerates convolution $4 \times$.

8.4 Evaluation of Object Detection

We implement two object detection algorithms on our hardware. To deploy the algorithms on our hardware design, we firstly trim the structure of the networks to fit the hardware and finetune the trimmed networks in floating-point number. Secondly, we finetune the float network with our fixed-point finetune method. There are two major object detection test sets, one is VOC [7] and the other one is ILSVRC [23]. Table 4 shows the mAP of each algorithm on different conditions. The accuracy of YOLO on ILSVRC declines sharply, and it is difficult to compute the mAP because of...
Table 4. Performance of different object detection algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Original accuracy</th>
<th>Trimmed accuracy</th>
<th>Fix-point accuracy</th>
<th>Run time (ms)</th>
<th>Complexity (GOP)</th>
<th>Performance (GOP/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>YOLO on VOC</td>
<td>63.4</td>
<td>62.3</td>
<td>61.8</td>
<td>65</td>
<td>30.6</td>
<td>942</td>
</tr>
<tr>
<td>SSD(^1) on VOC</td>
<td>74.3</td>
<td>73.1</td>
<td>73</td>
<td>92</td>
<td>52</td>
<td>1150</td>
</tr>
<tr>
<td>SSD(^1) on ILSVRC</td>
<td>39.6</td>
<td>37.2</td>
<td>36.2</td>
<td>105</td>
<td>55</td>
<td>1048</td>
</tr>
</tbody>
</table>

\(^1\) SSD models are optimized with the methods in Section 7.

The implemented system is illustrated in Fig 13. We plug the FPGA board into the PCIe slot and detect objects on the upper computer with PCIe driver of the hardware.

8.5 Comparison with CPU and GPU

To compare our hardware accelerator with CPU and GPU, we implement the SSD object detection network on ILSVRC set with a popular deep learning tool Caffe [13].

Caffe can run the network in GPU mode or CPU only mode. The platform is an Intel Core i7-4790K desktop CPU with an NVIDIA TITAN X (Pascal) GPU. Note that the target detection algorithm is SSD, which does not contain any FC layers. For a fair comparison, the batch size is configured to 2 in our experiments.

Table 5 summarizes the accuracy of these three platforms. Compared with CPU and GPU, our FPGA implementation with fixed-point number format achieves the comparable accuracy. The results also show that the power efficiency of our FPGA object detection system is 2.3× of the NVIDIA TITAN X (Pascal) GPU platform (released in Q3’2016) and 50× of the Intel Core i7-4790K desktop CPU platform (released in Q2’2014). Note that this conclusion may change when we target more modern higher-end CPU and GPU. Though SSD algorithm does not need to do FC layers, FPGA needs CPU assistance to perform image resizing and NMS operations to run full object detection. The CPU power in such case is not included when calculating FPGA efficiency number.
Table 5. Comparison With CPU and GPU

<table>
<thead>
<tr>
<th>Platform</th>
<th>Accuracy (%)</th>
<th>Power (W)</th>
<th>Run time (ms)</th>
<th>FPS</th>
<th>Performance (GOP/s)</th>
<th>Power efficiency (GOP/s/W)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU (i7-4790K desktop)</td>
<td>37.2</td>
<td>60</td>
<td>3132</td>
<td>0.6</td>
<td>35</td>
<td>0.59</td>
</tr>
<tr>
<td>GPU (TITAN X)</td>
<td>37.2</td>
<td>198</td>
<td>46</td>
<td>43.5</td>
<td>2391</td>
<td>12.08</td>
</tr>
<tr>
<td>FPGA (KU115)</td>
<td>36.2</td>
<td>37</td>
<td>105</td>
<td>19.0</td>
<td>1048</td>
<td>28.31</td>
</tr>
</tbody>
</table>

9 CONCLUSION

In this paper, we propose an instruction driven CNN accelerator based on FPGA with the optimization of the cross-layer scheduling and the Winograd computing unit. We also propose an instruction set for our cross-layer scheduling. To implement detection algorithms on hardware, we use the fixed-point finetune method to guarantee the accuracy of CNN on our fixed-point numerical hardware system. However, because of the fixed Winograd kernel size, our hardware is not that flexible and efficient on convolution layers with other kernel sizes like $1 \times 1$ and $5 \times 5$. In this works, the convolution with stride cannot be effectively supported and neither do convolution layers with dilation. We believe that future developments will implement hardware supporting different kernel sizes for the Winograd as well as convolution layers with stride and dilation.

ACKNOWLEDGMENTS

This work was supported by National Key R&D Program of China 2018YFB0105005, National Natural Science Foundation of China (No. 61622403, 61621091) and the project of Tsinghua University and Toyota Joint Research Center for AI Technology of Automated Vehicle (TT2018-01). This work is also supported by Beijing Innovation Center for Future Chip, Beijing National Research Center for Information Science and Technology(BNRist), and Xilinx Inc.

REFERENCES


